FPGA Architecture Exploration for DNN Acceleration

ESTHER ROORDA, University of British Columbia, Canada
SEYEDRAMIN RASOULINEZHAD and PHILIP H. W. LEONG, University of Sydney, Australia
STEVEN J. E. WILTON, University of British Columbia, Canada

Recent years have seen an explosion of machine learning applications implemented on **Field-Programmable Gate Arrays (FPGAs)**. FPGA vendors and researchers have responded by updating their fabrics to more efficiently implement machine learning accelerators, including innovations such as enhanced **Digital Signal Processing (DSP)** blocks and hardened systolic arrays. Evaluating architectural proposals is difficult, however, due to the lack of publicly available benchmark circuits.

This paper addresses this problem by presenting an open-source benchmark circuit generator that creates realistic DNN-oriented circuits for use in FPGA architecture studies. Unlike previous generators, which create circuits that are agnostic of the underlying FPGA, our circuits explicitly instantiate embedded blocks, allowing for meaningful comparison of recent architectural proposals without the need for a complete inference **computer-aided design (CAD)** flow. Our circuits are compatible with the VTR CAD suite, allowing for architecture studies that investigate routing congestion and other low-level architectural implications.

In addition to addressing the lack of machine learning benchmark circuits, the architecture exploration flow that we propose allows for a more comprehensive evaluation of FPGA architectures than traditional static benchmark suites. We demonstrate this through three case studies which illustrate how realistic benchmark circuits can be generated to target different heterogeneous FPGAs.

CCS Concepts: • **Hardware → Software tools for EDA**; **Hardware accelerators**; *Arithmetic and datapath circuits*; • **Computer systems organization → Reconfigurable computing**; **Neural networks**; • **Computing methodologies → Vector/streaming algorithms;**

Additional Key Words and Phrases: FPGA architecture, neural networks, benchmarking, hardware acceleration

ACM Reference format:
Esther Roorda, SeyedRamin Rasoulinezhad, Philip H. W. Leong, and Steven J. E. Wilton. 2022. FPGA Architecture Exploration for DNN Acceleration. *ACM Trans. Reconfigurable Technol. Syst.* 15, 3, Article 33 (May 2022), 37 pages.
https://doi.org/10.1145/3503465

Authors' addresses: E. Roorda and S. J. E. Wilton, University of British Columbia, Department of Electrical and Computer Engineering, 2332 Main Mall, Vancouver, British Columbia, Canada, V6T 1Z4; emails: {estherr, stevew}@ece.ubc.ca; S. Rasoulinezhad and P. H. W. Leong, School of Electrical and Information Engineering, University of Sydney, Sydney, Australia, NSW 2006; emails: {seyedramin.rasoulinezhad, philip.leong}@sydney.edu.au.

1936-7406/2022/05-ART33 $15.00
https://doi.org/10.1145/3503465

1 INTRODUCTION

The past two decades have seen major changes in the architecture of commercial Field-Programmable Gate Arrays (FPGAs). Capacity and performance have improved by orders of magnitude, while cost and energy per operation have decreased. These advances can be attributed in part to technology scaling, but there has also been a fundamental shift in the internal architecture of the FPGAs themselves. Traditional FPGAs consisted of a fabric of general purpose logic blocks and routing that could be reprogrammed to implement any digital circuit. Increasingly however, commercial FPGA architectures have evolved to include embedded blocks, specialized routing fabrics, and other structures optimized for implementing specific types of circuits.

In recent years, there has been particular interest in modifying FPGAs to better support machine learning applications. Research has considered enhancing the traditional FPGA building blocks to make them better suited to performing the tensor operations that make up the majority of Deep Neural Network (DNN) layers [6, 8, 11, 20, 30, 31]. Adding composable hard matrix multiplication blocks to the FPGA fabric has also been explored [2, 3, 7, 23] as a way to improve computation efficiency while maintaining a high degree of flexibility. Such embedded tensor blocks have found their way into commercial FPGAs such as Intel's Stratix 10 NX architecture [7, 23].

Architectural proposals such as these need to be evaluated to determine whether the architecture provides the right balance between flexibility (the ability to implement a variety of user circuits) and efficiency (in terms of density, speed, and power). Usually, this evaluation is done using an experimental approach [5]. FPGA vendors typically use an in-house flexible CAD flow to map suites of designs to the FPGA architectures under consideration. A similar experimental flow is used in academia, often using publicly available benchmark suites (e.g., [27, 45]) and open-source experimental flows such as VTR [26]. In either case, detailed area, delay, and power models can be used to evaluate the effectiveness of the proposed architectural enhancements.

Conducting such experiments, however, is especially difficult when evaluating FPGAs optimized for machine learning applications, for at least two reasons. Firstly, existing publicly available benchmark suites such as MCNC [45] and Titan [27] do not include examples of machine learning acceleration. Even if machine learning circuits were added to existing benchmark suites, the rapid evolution of the machine learning field means that any suite of benchmark circuits containing machine learning circuits may become stale if not regularly updated. A second, and more challenging concern arises because many of these architectural enhancements involve adding or optimizing large embedded blocks. Each of these blocks is intended to implement a significant amount of functionality. For example, the AI Tensor blocks in [7, 23] contain an array of multiplication/addition operations, which form the bulk of the processing in many machine learning workloads. Mapping circuits to these devices requires an intelligent inference engine to find the best way to implement each workload for each potential embedded tensor block under consideration. This mapping is critically important, not only to evaluate the effectiveness of embedded tensor blocks, but also to evaluate the routing fabric connected to the blocks, and their interaction with other components (e.g., embedded memories). However, given the flexibility typically included in these blocks, and the number of ways these sorts of applications can be "folded" onto a set of flexible blocks, such a mapping algorithm is not trivial to create. During early architecture investigation, inference engines optimized for each proposed embedded tensor block may not yet exist. Creating such an inference engine for each potential block would be time consuming and would limit the size of the design space that could practically be explored.

In this paper, we describe a framework which addresses both of these issues. Our framework provides for rapid preliminary evaluation of new FPGA architectural ideas aimed at improving the efficiency of the FPGA when implementing machine learning circuits. Rather than relying on

a static set of benchmark circuits, we define a parameterized space that covers a wide variety of machine learning workloads. Given a point in this space, and a description of a potential architecture to be evaluated, our framework generates a benchmark circuit and uses an optimization algorithm to "unroll" the circuit onto the potential architecture. The generated benchmarks are functional circuits designed to implement a given DNN workload while targeting a given FPGA architecture. This requires a fundamentally different approach to benchmark circuit generation than previous benchmark circuit generators, taking a description of a heterogeneous target architecture and generating a circuit that instantiates available embedded blocks. We are not aware of any previous work on DNN circuit generation, or synthetic benchmark generation that would be suitable in this context.

Specifically, the contributions of this paper are as follows:

(1) **Problem Space Definition**: We carefully define the space of architectures and workloads that we target. This includes:
   - A taxonomy of embedded tensor blocks in the context of DNN acceleration on FPGAs
   - A mapping vector definition that encodes the loop-unrolling and tiling factors used to map a DNN workload onto hardware.

(2) **Benchmark Generator**: We have implemented an open-source algorithm that generates synthesizable benchmark circuits suitable for use in FPGA architecture research in conjunction with the commonly used VTR framework [26]. Each generated circuit implements one or more DNN layers, given a mapping vector and information about the underlying architecture. The framework includes:
   - A mapping algorithm to select unrolling factors given a target architecture and DNN layer specification.
   - A set of open source Python scripts and pyMTL models used to generate functional DNN accelerator Register Transfer Level (RTL).
   - A simulation flow that can be used to verify the functional correctness of each accelerator generated, as well as a thorough test suite.

(3) **Case Studies**: We demonstrate that our framework can be used to examine FPGA architectures and design trade-offs in a way that was not previously possible due to limitations of existing benchmark suites and inference engines.
   - We model two existing commercial FPGA architectures and generate circuits targeted to these architectures.
   - We vary several parameters of a baseline FPGA architecture to demonstrate how our framework can be used to explore different architectural trade-offs.
   - We explore different ways of accelerating multi-layer networks (as opposed to single layer workloads) on FPGAs and the routability implications of different alternatives.

This paper is organized as follows. Section 2 provides background on FPGA architecture exploration, as well as summarizing related work on automatically generating machine learning accelerators and more generally, synthetic benchmark circuitry. Section 3 describes our architecture exploration framework, in particular our classification system for DNN workloads and embedded tensor blocks and our benchmark generation tool. Section 4 presents and discusses the results of case studies performed using our architecture exploration flow.

Finally, in Section 5 we address some key limitations of our framework and avenues for improvement. Specifically, limitations include the fact that the framework is integrated with VTR and can't currently be used with typical commercial compilers (meaning timing delays must be estimated and modelled in VTR). It is also important to note that in terms of performance, the generated designs are not intended to be competitive with hand-tuned high performance accelerators, but

rather to assist in early comparisons of architecture tradeoffs. Section 5 also describes how these limitations could be addressed in future work

Our open-source framework is available at http://github.com/eroor8/verilog_ml_benchmark_generator.

2 BACKGROUND

In this section, we set the context for our paper by describing recent work on FPGA architectural enhancement, FPGA architectural exploration frameworks, and synthetic circuit generation methods. We also summarize previous work on automated DNN accelerator generation.

2.1 FPGA Architectures for Machine Learning

Increasing interest in the use of FPGAs for machine learning acceleration has motivated recent research into updating the FPGA fabric to more efficiently perform the computations associated with DNNs. Proposed architecture changes include adjustments to the general purpose FPGA fabric. Boutros et al. have proposed changes to Intel's Logic Array Block (LAB) architectures to allow for denser low precision MAC operations in the FPGA fabric [6]. These changes include additional carry chains between LUTs and a low precision multiplier in each LAB. Eldafrawy et al. [11] have also suggested several changes to logic blocks, including adding shadow multipliers to logic clusters, adding several adders to each LAB, and increasing Look-Up Table (LUT) fracturability. Similarly, the work presented in [20, 30] focuses on improving logic element architecture for implementing ultra-low precision neural networks. Other work proposes updates to embedded Digital Signal Processing Blocks (DSPs) block architecture to improve performance of DNN workloads. In [8], Boutros et al. explore adjustments to the DSP architecture with the goal of improving performance for low precision multiply accumulate operations. PIR-DSP [31] also adjusts DSPs to make them better suited to the lower-precision (or multi-precision) requirements of neural networks, and adds DSP register files to improve local data reuse. In the work described above, evaluations were primarily done using microbenchmarks and small hand-tuned designs, rather than using realistic DNN circuits. In many cases, adjustments to the regular FPGA Computer Assisted Design (CAD) flow were also required to ensure that the new architectures were used efficiently. For example, in [11], the open-source CAD tools used to compile benchmark circuits (VTR and ODIN) were customized to target the proposed logic block changes, while in [6], microbenchmarks were mapped to logic blocks by hand.

Rather than modifying existing FPGA building blocks, another strategy is to add dedicated machine learning blocks to the FPGA fabric to efficiently perform tensor operations. Hamamu [3] adds hardened systolic matrix multipliers to the FPGA fabric, as well as the programmable interconnects required to combine them into larger networks. In [2], Arora et al. also propose adding hardened blocks to perform tensor operations, which can similarly be composed in chains or arrays. Unlike Hamamu [3], these blocks can be configured to perform either matrix multiplications or element-wise computations, and also include a local crossbar and control logic to read and write to connected memories. Hardened tensor blocks have also been introduced to recent commercial FPGA architect 10 NX architecture [7, 23] introduces AI Tensor Blocks, each of which contains 30 INT8 multipliers that can be configured to perform either matrix or vector operations. Intel's AI Tensor Blocks are arranged in columns, and adjacent tensor blocks are connected through direct connections. Each of these works evaluate performance using a small number of hand-written benchmark designs or a set of small matrix multiplication micro-benchmarks, written by hand to target the available hardware.

While the Xilinx Versal architecture similarly provides an array of hardened tensor blocks, "AI Engines", these blocks are not embedded in the FPGA fabric. Instead, they are connected to the rest

of the FPGA fabric through a Network on Chip. This architecture also differs from the work described above in that each ASIC Engine includes a simple RISC processor and instruction memory, and is programmed using a C/C++ paradigm.

2.2 FPGA Architecture Exploration

When FPGA vendors or academic researchers consider architectural ideas such as those discussed in the previous section, they need to evaluate the impact of these enhancements on the flexibility, speed, density, and power of the resulting FPGA. Although some strides have been made towards analytical modeling of FPGA architectures [10], the primary way to perform these studies is to use an experimental approach, such as that espoused in [5]. In such an approach, the FPGA architecture is modeled, and an experimental CAD flow (such as VTR [26]) is used to map benchmark circuits to the devices. Detailed area, delay, and power models are then used to evaluate the efficiency of the mapping, allowing for direct comparisons between architectural alternatives.

The use of suitable benchmark circuits is critical. While the properties of individual FPGA building blocks, including maximum operating speed and power models, can be calculated for a given architecture, these measurements alone do not capture the overall performance of complex circuits implemented on FPGAs. Maximum clock frequency, routability, and power, for example only have meaning when applied to a particular circuit. FPGA vendors have their own large benchmark suites and open source benchmark suites are publicly available for academic research, such as [45] (small ciruits that do not include DSPs or Block Random Access Memory (BRAM)) and [27] which are relatively large heterogeneous benchmark circuits that cover a wide range of different domains. There is currently no existing benchmark suite that consists of deep learning workloads.

The experimental CAD flow used to map the benchmark circuits is critical. Mature commercial CAD flows are optimized and designed for specific architectures, which makes it difficult to use these tools during FPGA architecture research. Flexible experimental flows such as Verilog-To-Routing (VTR) allows for different FPGA architectures to be specified and targeted. Unlike commercial CAD software however, its flexibility means that VTR is not designed to perform performance optimizations that target specific architectural features. In particular, new types of embedded blocks cannot always be automatically inferred from a user's digital logic without tool modifications. This is especially a concern when investigating FPGAs optimized for deep learning applications, since such FPGAs may have complex flexible embedded blocks such as the AI Tensor blocks in [7, 23]. These blocks perform significant computation, and inferring the best use of these blocks from a benchmark circuit is difficult. It is possible to investigate architectures with novel embedded blocks by using benchmark circuits with direct instantiations of these blocks. This means, however, different versions of each benchmark circuit is required for each different type of embedded block to be investigated. As will be described in Section 3, our flow solves this by creating a flexible mapping algorithm to map high-level descriptions of deep learning workloads to a parameterized embedded block description.

2.3 Synthetic Benchmark Circuit Generation

As discussed in Section 2.2, benchmarks are an important aspect of FPGA architecture and CAD tools exploration and evaluation. Ideally a diverse suite of realistic benchmark circuits would be available to evaluate new architectures and CAD flows. In practice these benchmark circuits are time consuming to develop manually, and real customer designs are rarely publicly available. A potential solution is using synthetic benchmark circuit generation frameworks, which generate circuits that mimic the properties of real circuits but are not necessarily functional.

There are several approaches that have been used in previous work. One approach is to build circuits bottom up with the same structural characteristics as existing benchmark suites. Early work

by Hutton [18] generated combinational and sequential circuits given a set of desired parameters, including circuit size and fanout distribution. Follow up work by Kundarewich [21] improved on this solution by generating clustered hierarchical circuits with wire lengths closer to those of real circuits. Stroobandt's GNL generator similarly creates circuits by recursively clustering logic while maintaining a desired Rent parameter [37, 42].

Another technique used in later work [25, 29, 40] is to generate synthetic circuits by combining existing sub-circuits. PartGen [29] combines smaller circuits, including small combinational circuits generated using GEN, with memory block instantiations and interconnects. Similarly, in [40], large circuits are constructed by connecting the inputs and outputs of existing subcircuits. In [25], subcircuits are connected through interconnects into large hierarchical designs, which are shown to match the characteristics of actual system-on-chip designs.

A third approach is to make small modifications to existing circuits on order to generate a family of similar circuits. Mutation-based generators include work by Ghosh [13], which randomly replaces a percentage of netlist edges, and Perturber [15], which uses a similar approach to generate benchmarks for testing incremental place and route tools.

2.4 Automated DNN Accelerator Generation

Functional, realistic DNN accelerators are time consuming to develop manually, particularly if they need to be carefully tuned to take full advantage of available architectural features. There are many ways to map the computation patterns of each DNN network layer onto hardware using different loop unrolling schemes and tiling factors; the space of possible solutions, and the optimal solution both depend on the hardware available. To facilitate this process, several frameworks have been developed to assist in creating functional DNN accelerators for a given network.

Automatically mapping a DNN network to FPGA hardware involves traversing the set of possible unrolling schemes and tiling factors. Early tools [46, 49] identify optimal dataflows and unrolling factors, which users would then implement manually. In [49], this is done using the Roofline model, while [46] creates an analytical model of memory energy and traffic. In other work, after an optimal mapping is identified, synthesizable accelerators can be generated automatically. In some cases [47, 48, 51], this is done by instantiating pre-written parameterized Hardware Description Language (HDL) modules. For example, DNNBuilder [51] includes optimized Processing Engine (PE) templates parameterized with different unrolling factors. Other solutions [38, 41, 50] generate HDL by taking advantage of high level synthesis tools to generate synthesizable Verilog.

A commonality of the DNN accelerator generation frameworks described above is that they generate generic soft logic, technology mapped onto traditional FPGA building blocks by FPGA CAD flows, giving users little fine grained control over exactly how FPGA building blocks are instantiated and used. This kind of implementation detail can have a significant effect on performance. For example, carefully placing and instantiating DSPs to take advantage of cascaded DSP connections has been shown to help maximize clock frequency of CNN accelerator implementations [32]. Some HLS compilation software allows for embedded blocks to be explicitly instantiated. However, the existing HLS-based accelerator generation frameworks do not make use of this capability to explicitly instantiate embedded tensor blocks. Given that it is not possible to target arbitrary embedded blocks using these existing frameworks, they are therefore not suitable for generating benchmark circuits for evaluating heterogeneous architectures.

DNNWeaver [33] and DeepBurning [43] generate accelerators using a set of provided templates or libraries. These templates are customizable and allow users to map accelerator components to specific hardware resources in certain cases. Neither framework can automatically target arbitrary embedded tensor blocks however, and adapting these tools to target a specific embedded block would require significant manual changes to the existing templates.

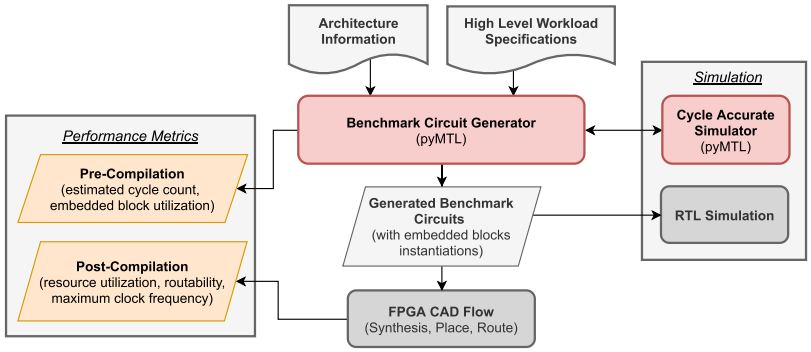

Fig. 1. Overall FPGA architecture exploration framework, including automated benchmark circuit generation and simulation.

3 FPGA ARCHITECTURE EXPLORATION FRAMEWORK

In this section, we describe our architecture exploration framework. We first present an overview of the overall framework and show how it could be used during the process of FPGA architecture exploration. We then outline the inputs to the flow, which include a set of DNN workloads for which a given FPGA architecture is to be evaluated, and a definition of the FPGA architecture under evaluation, including embedded tensor blocks. Section 3.4 describes how DNN accelerators for given DNN workload are then generated to target the available FPGA architecture. We discuss how these generated circuits can be simulated and used as benchmarks to measure FPGA architecture performance in Section 3.6. Section 3.7 describes how the framework handles multi-layer networks as opposed to single DNN layers.

3.1 Overall Flow

Figure 1 shows our overall experimental framework. Given information regarding the FPGA architecture under investigation as well as high-level workload information, our flow generates an appropriate benchmark circuit for use with an experimental synthesis, placement and routing tool (we use VTR due to its widespread use in the community). VTR compiles the benchmark circuit on the target device, and uses detailed area, delay, and power models to estimate the efficiency of the architecture. We also provide the ability to simulate our benchmark circuits at two levels: we provide links to both a cycle-accurate simulator and an RTL simulator.

The heart of our flow, and the main contribution of this paper, is benchmark circuit generation. Figure 2 shows a block diagram of our benchmark circuit generator. Given the high-level workload specification and information about the available embedded blocks, our flow identifies legal mapping vectors which determine how the workload is implemented on the target FPGA. In each case, the given DNN workload is mapped as efficiently as possible to the available DSPs or embedded tensor blocks. This allows for the measurement of low-level FPGA architecture metrics like routability and maximum clock frequency using realistic, targeted benchmark circuits. It also allows for a more high-level evaluation of how efficiently different DNN workloads can be parallelized on a proposed FPGA architecture. Given this mapping, PyMTL models are then generated, from which the benchmark circuit is created. Circuit generation using pyMTL is described in more detail in Section 2.3. Since we wish to target the VPR flow for place and route, and since the synthesis front-end of VTR, ODIN, only supports a subset of modern Verilog syntax, post-processing of the Verilog is required to ensure that it is compatible with the VTR flow.

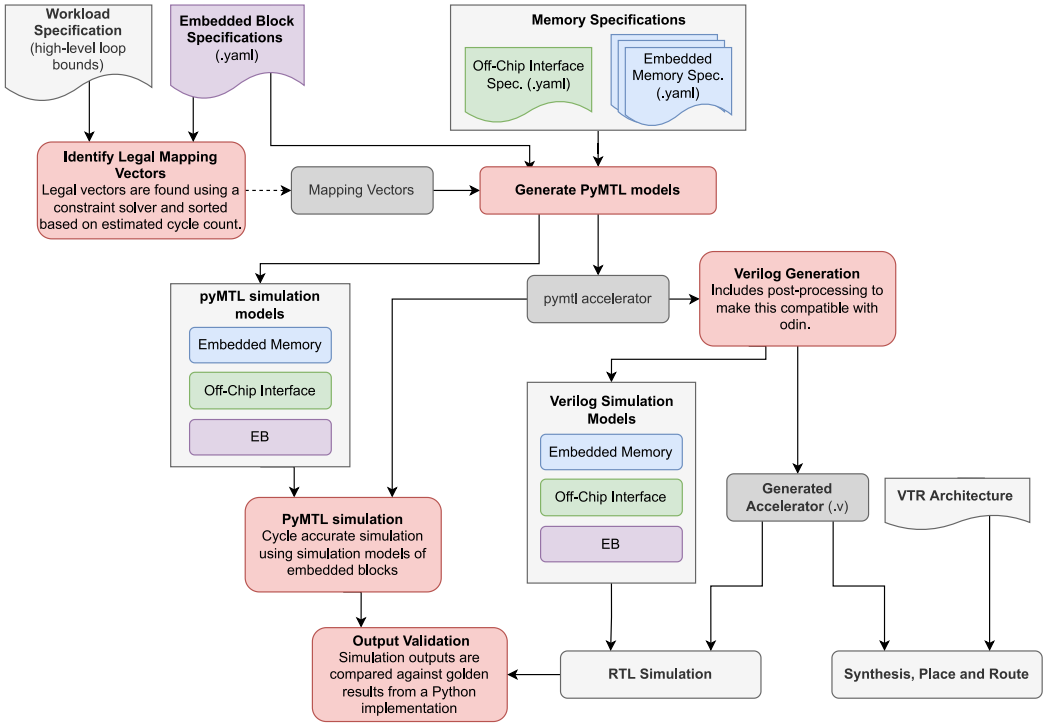

Fig. 2. FPGA benchmark generation framework components, including mapping selection and simulation.

Importantly, the generated circuits make use of embedded blocks by instantiating them directly as components, rather than relying on the CAD flow to infer the use of these blocks. Many deep leaning models can be written as a series of nested loops (as will be discussed in Section 3.2); within our generator, we "unroll" these loops to use the available embedded blocks, and build control circuitry around the blocks to ensure proper sequencing of the arithmetic operations performed by the block. The generated circuits then contain both the embedded block instantiations, the surrounding logic, and the associated interconnect. User memories are also instantiated as necessary. In a mature production CAD flow, this should not be necessary; a synthesis tool should be able to instantiate the available embedded blocks to implement a circuit specified in terms of nested loops. However, during early architecture investigation, inference engines optimized for each block may not yet exist. Creating such an inference engine for each potential block would be time consuming and would limit the size of the design space that could practically be explored.

It is also important to note that using our framework, the same benchmark workload may have different cycle-to-cycle behaviour when implemented on different FPGA architectures. An FPGA containing large embedded tensor blocks may be able to implement much more functionality in a single cycle than an FPGA with only simple DSP blocks. This is different than many other FPGA architecture studies (e.g., such as those in [5]) in which potential architectures all implement the same cycle-by-cycle behaviour for each benchmark circuit, since using our framework both the critical path and the number of cycles vary across different workloads and architectures. Ultimately, both metrics are likely to be different for high-level accelerators optimized using commercial compilation software and hand tuned by researchers. Even so, the proposed framework allows for a more thorough preliminary assessment of design tradeoffs during the early stages of architecture exploration than can be done using static benchmarks.

Adjusting cycle-to-cycle behaviour depending on the available architecture has two implications. Firstly, to provide some confidence in the functionality of the generated circuits, we provide links to a cycle-accurate simulator, as will be discussed in Section 3.6.1. The second implication is that the benchmark circuit generation needs to understand the capabilities of the embedded block, so it can perform the proper "unfolding" and ensure that the controlling state machine is constructed properly. This means a specification method to describe the functionality of these embedded blocks is required; Section 3.3 discusses our method.

More details on this flow is presented in the following subsections.

3.2 Framework Input: DNN Workload Parameterization

As described in the previous section, there are two inputs to our flow: a high level description of the DNN workload, and a description of the embedded blocks available on the target FPGA. In this section we describe the workload specification; in Section 3.3 we describe the manner in which embedded blocks are described.

A DNN may include several different types of layers, including convolutional, fully connected, pooling, normalization, and activation layers. Convolutional, fully connected and depthwise separable layers are based on matrix multiplication (GEMM) operations. These GEMM-based layers are particularly computation intensive, and are the primary targets of many of the embedded blocks described in Section 2.1. However, since realistic DNNs typically include other types of non-GEMM layers (e.g., pooling layers) and functions (e.g., ReLU functions), realistic DNN accelator benchmarks also need to take these types of computation into consideration.

Rather than considering each type of layer individually, we use a generalized loop nest model to encode various different types of layers based on their access patterns. As many layers have similar properties and can be parallelized using similar techniques, this approach requires less custom code. Using a general parameterized model also means that our framework can be more easily extended to new types of layers as they are developed without major changes to the code base. DNN layers can typically be formulated as a set of nested loops, although the innermost operations and the loop bounds differ from layer to layer and network to network. We primarily categorize each DNN layer based on (1) the nested loop bounds, and (2) the operations performed on data within the loop nest.

The generalized nested loop model shown in Algorithm 1 includes the superset of all nested loops required to express various common DNN layers. Each of the eight nested loop dimensions accesses input activations, output activations, and weights using a different access pattern. To define a given layer, loop bounds are encoded in an eight-element vector, $W = \{B, C, E, PX, PY, RX, RY, G\}$. Likewise, strides can also be encoded in an eight-element vector, although strides greater than one are currently supported only in the PX, PY, RX, and RY dimensions.

In GEMM-based layers, the innermost function is a Multiply Accumulate (MAC) operation. In a max-pooling layer this function would be a comparison to find the maximum value of a subset of inputs. Our framework includes a library of functions, including activation functions such as ReLU and sigmoid functions. This library can easily be extended to include more functions as required. Examples of common layer types with corresponding loop bound vectors and innermost operations are listed in Table 1. Activation functions could be considered a separate layer in which a function is applied to each input individually, but they are typically combined with the preceding layer for efficiency. To support this common use case, an additional activation function can optionally be specified for each layer output. Input and activation bitwidth are also specified, allowing for the generation of benchmark circuits representative of quantized and binarized neural networks. Finally, zero-padding can optionally be applied to convolutional layers, which has a small effect on cycle count estimates and generated circuitry.

Table 1. DNN Layer Parameterization Examples

Layer Type	Loop Bound Vector	Inner Operation (f_{inner})
2-D Convolutional	$< 1, C, E, PX, PY, RX, RY, 1 >$	$f(o, i, w) = o + i \times w$
1-D Convolutional	$< 1, C, E, PX, 1, RX, 1, 1 >$	$f(o, i, w) = o + i \times w$
Fully Connected	$< 1, C, E, 1, 1, 1, 1, 1 >$	$f(o, i, w) = o + i \times w$
Depthwise	$< 1, 1, 1, PX, PY, RX, RY, G >$	$f(o, i, w) = o + i \times w$
Pointwise	$< 1, C, E, PX, PY, 1, 1, 1 >$	$f(o, i, w) = o + i \times w$
Max Pooling	$< 1, 1, 1, PX, PY, RX, RY, 1 >$	$f(\mathrm{o}, \mathrm{i}) = MAX(o, i)$
Average Pooling	$< 1, 1, 1, PX, PY, RX, RY, 1 >$	$f(o, i) = o + i/(RX \times RY)$

ALGORITHM 1: Generalized Nested Loop Model

```
for g = 1 to G, stride=SG (groups) do
    for b = 1 to B, stride=SB (batches) do
        for e = 1 to E, stride=SE (output channels) do
            for px = 1 to PX, stride=SPX (filter map x dimension) do
                for py = 1 to PY, stride=SPY (filter map y dimension) do
                    for c = 1 to C, stride=SC (input channels) do
                        for rx = 1 to RX, stride=SRX (filter x dimension) do
                            for ry = 1 to RY, stride=SRY (filter y dim.) do
                                i = I[g][b][c][px + rx][py + ry];
                                w = W[g][e][c][rx][ry];
                                o = O[g][b][e][px][py];
                                O[g][b][e][px][py] = f_inner(i, w, o);
                        O[g][b][e][px][py] = f_act(O[g][b][e][px][py]);
```

The generalized loop nest model described in this section can be used to describe the most, but not all, common types of DNN layers. For example, three dimensional convolution is not supported, as 3-D convolution involves additional dimensions that are not currently included in the loop bound vector. Future work could expand this generalized nested loop model to enable these use cases by adding additional dimensions.

Typically, a DNN consists of multiple layers. A full DNN network can be represented by a collection of DNN layers, each of which is defined separately using the parameters described above.

3.3 Framework Input: Embedded Block Parameterization

A key difference between our circuit generator and previous work is that previous work generates circuits that are agnostic of the underlying FPGA architecture. In our work, benchmark circuits may contain explicit instantiations of embedded blocks. Here we specify the space of the underlying **embedded block (EB)** architectures that our framework supports. It is important to note that the main purpose of our benchmark generation framework is not to generate Verilog for the embedded blocks themselves, but rather to instantiate the existing embedded blocks in benchmark designs. Accordingly, Embedded Block (EB) definitions are primarily used by our framework to ensure that available EBs are instantiated and used appropriately in generated benchmark designs. While the existing embedded blocks proposed for machine learning applications primarily

perform multiply accumulate operations, our embedded block definition is general enough to extend to other operations as well in order to accelerate non-GEMM based layers.

The following parameters can be used to describe the target architecture's embedded blocks. These parameters together describe the "design space" of embedded blocks supported by our framework.

Access Patterns: Embedded blocks can be classified based on their data access patterns, which we have grouped into five categories. The EB's data access pattern limits the ways in which the workload dimensions (as defined in Section 3.2) can be unrolled and mapped onto the EB. For example, if an EB multiplies two inputs by two weights and adds them to produce a single output value, it could be used to process two input channels in parallel (unrolling dimension C in Section 3.2 by a factor of two). A different EB may produce two outputs in parallel by multiplying a single weight by two input streams. In this case, it could be used to compute two batches (unrolling dimension B in Section 3.2 by a factor of two). Access pattern categories are listed below, along with the corresponding workload dimensions from Section 3.2. Examples of circuitry that uses each access pattern are illustrated in Figure 4. The computations performed by these layers are also shown, assuming that the EBs perform MAC operations:

- **AP1: Windowing** *(dimension: RX)*: Using windowing, the EB performs a dot product of AP1 consecutive inputs from a single input stream, with AP1 pre-loaded weights to produce a single output on each clock cycle. This access pattern requires for weights to be pre-loaded. Figure 4 shows an example of how this can be achieved, by passing a a sequence of input activations through a shift register.

$$o = \sum_{t=1}^{AP1} W[t] \times i_t \tag{1}$$

- **AP2: Dot-Product** *(dimensions: RY, C)*: AP2 parallel inputs are multiplied by AP2 weights, and added to produce a single output on each clock cycle.

$$o = W[1 : AP2] \cdot I[1 : AP2] \tag{2}$$

- **AP3: Single Input** *(dimension: E)*: AP3 weights are multiplied by the same input, producing AP3 outputs per clock cycle.

$$O[1 : AP3] = W[1 : AP3] \times i \tag{3}$$

- **AP4: Single Weight** *(dimensions: B, PX, PY)*: AP4 inputs are multiplied by the same weight, producing AP4 outputs per clock cycle.

$$O[1 : AP4] = w \times I[1 : AP4] \tag{4}$$

- **AP5: Element-Wise Multiplication** *(dimensions: G)*: AP5 different inputs are multiplied by AP5 weights to produce AP5 outputs per clock cycle.

$$O[1 : AP5] = W[1 : AP5] \odot I[1 : AP5] \tag{5}$$

In more complex EBs, multiple access patterns can be combined, as shown in Figure 3, meaning that multiple workload dimensions may be unrolled within each block. A wide variety of embedded blocks, from a single MAC or DSP to the embedded matrix multiplication blocks proposed in [23],[7], and [3], can be defined using these five access patterns. Table 2 lists examples of how several of the proposed embedded blocks and DSPs discussed in Section 2 could be modelled.

Intra-EB storage: We assume that each EB operates in either weight-stationary mode or output-stationary mode [39]. Weight stationary blocks stream through input activations and produce one

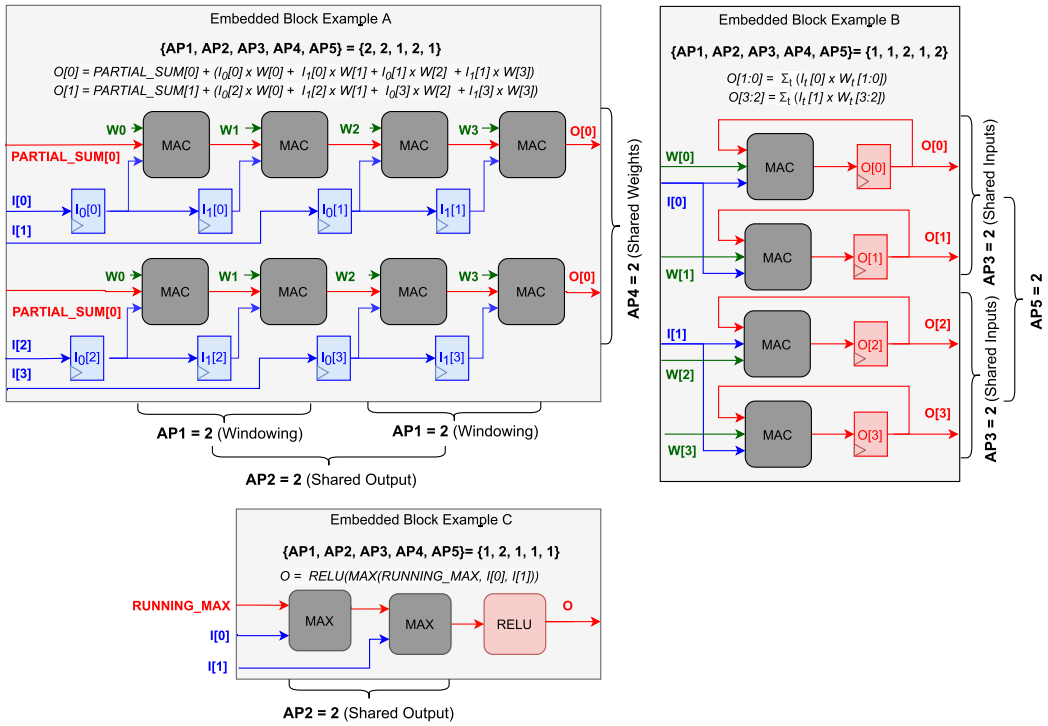

Fig. 3. Three EBs with different combinations of access patterns {AP1, AP2, AP3, AP4, AP5}. Access patterns are {2, 2, 1, 2, 1}, {1, 1, 2, 1, 2}, and {1, 2, 1, 1, 1} for block A, block B, and block C, respectively.

or more output activations on each clock cycle. In this case weights are stored in registers within the EB, which can be either be all loaded into the EB in parallel or pre-loaded serially. Output-stationary blocks stream multiple weights and inputs in parallel and accumulate partial sums in internal registers. How a workload can be mapped onto set of EBs depends on whether they can operate in weight and/or output stationary modes, and therefore whether they have the capability to store outputs and weights in internal registers.

Computation Type: Most embedded blocks proposed for DNN acceleration [3, 7, 23] consist primarily of MAC units. However, to accelerate non-GEMM based layers and implement activation functions it may also be useful to also harden other functions. To support these use cases, we include an extensible library of operations, including common activation functions, any of which can be performed by the EB.

Data Representation: EBs also differ in terms of data bitwidth and data representation of inputs, outputs, and weights. We assume a fixed point representation, with data width specified by the user.

Timing: The number of cycles required by a generated benchmark circuit depends on the cycle-level timing of embedded tensor blocks. By default we make assumptions about the capabilities and structure of embedded tensor blocks and the clock cycles they require to operate. For example, we assume that each multiply accumulate operation takes a single clock cycle. Alternatively, the user can specify these embedded block properties to help ensure the accuracy of high level clock cycle estimates.

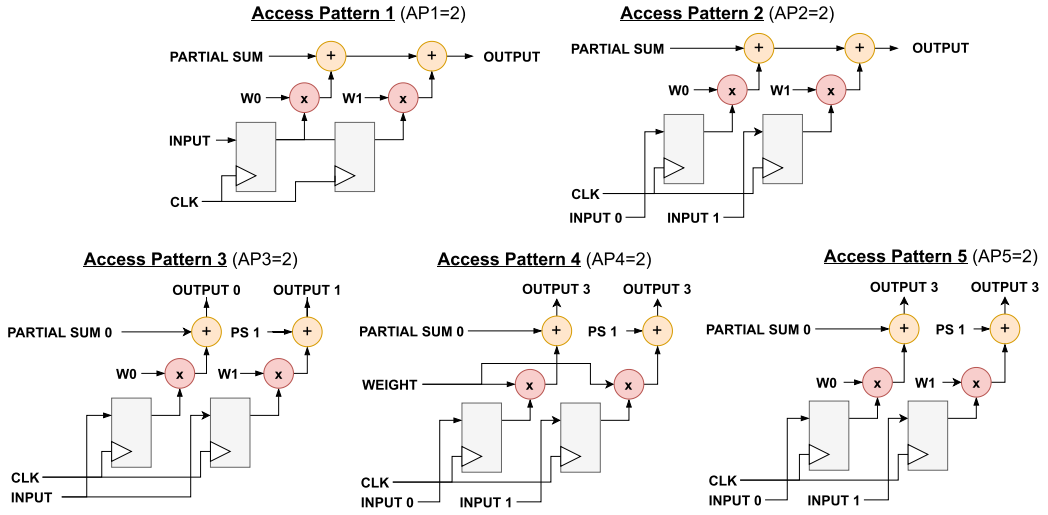

Fig. 4. Examples of circuitry illustrating access patterns AP1 to AP5.

Table 2. Example Definitions for Existing and Proposed Embedded Blocks

FPGA Architecture	Mode	Bitwidth	Access Patterns	Additional Notes
Boutros et al. [8]	MAC Mode	9-bit	{1, 1, 1, 1, 4}	Additional modes include 27×27 and 18×18 multiplication
		4-bit	{1, 1, 1, 1, 8}	
	Independent	9-bit	{1, 2, 1, 1, 2}	
		4-bit	{1, 4, 1, 1, 2}	
Hamamu [3]	$4 \times 4 \times 4$ Multiplier	Various	{1, 1, 4, 4, 1}	Various other multiplier dimensions also explored.
PIR-DSP [31]	i = 1, j = 1, k = 1	27×18-bit	{1, 1, 1, 1, 1}	
	i = 1, j = 3, k = 1	9-bit	{1, 3, 1, 1, 2}	
	i = 1, j = 3, k = 2	4-bit	{1, 3, 1, 1, 4}	
	i = 1, j = 3, k = 4	2-bit	{1, 3, 1, 1, 8}	
Tensor Slices [2]	Tensor Mode	16-bit	{1, 1, 4, 4, 1}	Note that these blocks also contain logic that can't be fully modelled in our framework, as discussed in Section 5.1
		8-bit	{1, 1, 16, 4, 1}	
	Individual PE Mode	16-bit	{1, 1, 1, 1, 16}	
		8-bit	{1, 1, 1, 1, 64}	
Intel AI Blocks [7, 23]	Tensor Mode	8-bit	{1, 10, 3, 1, 1}	See Section 4.2.1
Xilinx UltraScale DSP48E	8-bit multiplication	8-bit	{1, 10, 3, 1, 1}, {1, 10, 3, 1, 1}	See Section 4.2.1

Architectures without Embedded Blocks: In some cases it may be necessary to target FPGA architectures that do not include embedded blocks or DSPs at all. For example, this may be useful in order to measure baseline performance during an architecture study. When mapping to an architecture without embedded blocks, we define a *logical embedded block*. In this case, the user can define an embedded block as before, and the framework generates a Verilog description of the EB for synthesis to soft logic. Alternatively, the user can provide a soft logic implementation of an embedded block, which is instantiated in the generated benchmark circuits and compiled to soft logic by the FPGA CAD flow.

3.4 Circuit Generation and Mapping

This section describes our benchmark circuit generator, which automatically produces benchmark circuitry given a DNN workload definition and a FPGA architecture specification. We first define mapping vectors that describe how a DNN workload is mapped to hardware in Section 3.4.1. We then discuss how efficient mapping vectors can be selected automatically using a constraint solver in Section 3.4.2. Section 3.4.3 discusses how given a mapping vector, Verilog circuits are generated using pyMTL. Finally, Section 3.5 describes the generated circuits themselves in more detail.

3.4.1 Mapping Vector Definition. A given DNN layer can be mapped to a set of EBs in many ways, each resulting in different routing requirements and a different maximum clock speed. Our framework uses a "mapping vector" to define how the nested loops shown in Algorithm 1 are unrolled, tiled, and scheduled on hardware.

To implement a given workload on a set of EBs, each of the eight nested loops may be unrolled within each EB (an EB performs multiple operations in parallel) and across EBs (multiple EBs operate in parallel). These spatial unrolling factors are limited by the capabilities of each EB and the number of EBs available. Operations are also typically scheduled in time, with each EB performing several operations over the course of several clock cycles. We can formulate each possible solution as a set of three nested 8-dimensional loops, represented by three 8-element vectors: the *Intra-EB Unrolling Factors* vector ($U_i = \{u_{i0}, \ldots, u_{i7}\}$), describes unrolling that occurs *within* an EB. *Inter-EB Unrolling Factors* ($U_o = \{u_{o0}, \ldots, u_{o7}\}$), describe how unrolling is performed between EBs. *Temporal Tiling Factors*, ($U_t = \{u_{t0}, \ldots, u_{t7}\}$), describe how the loops are unrolled in time.

An example mapping is given in Algorithm 2. In this simple example, there are two embedded blocks, which each performing four MAC operations. A convolutional layer is mapped onto these two embedded blocks such that each block convolves a different two-dimensional filter over the same input feature map. This is repeated over time for multiple pairs of filters and batches of inputs. In this way, the computations are parallelized across a total of eight MAC units. Realistically, FPGAs would have many more than two embedded blocks, allowing for higher degrees of parallelism.

These vectors can be specified directly by the user, or our framework can automatically select values for these vectors based on what is supported by the EB architecture. The EB access patterns determine which dimensions can be unrolled spatially within each EB, as discussed in Section 3.3.

3.4.2 Mapping Vector Selection. As discussed in Section 3.4.1, the way in which a DNN layer is parallelized on hardware can be encoded using a set of unrolling vectors, $U_t = \{u_{t0}, \ldots, u_{t7}\}$, $U_o = \{u_{o0}, \ldots, u_{o7}\}$ and $U_i = \{u_{i0}, \ldots, u_{i7}\}$. A given DNN workload may be mapped to available FPGA resources in many different ways, each of which would result in different routing patterns, resource utilization, and overall performance. End-users would realistically decide on a parallelization strategy depending on the available hardware and the workload to be accelerated; always assuming a single parallelization scheme would result in sub-optimal performance.

Our framework can be used in two modes. In the first mode, the user specifies all unrolling vectors (U_t, U_o, U_i) directly; this manual approach enables architectural sensitivity analysis [44]. These unrolling factors must still be compatible with the available FPGA resources however, and our framework performs validation to ensure that this is the case. In the second mode, we aim to emulate the real process of DNN accelerator design. The user specifies only the high-level loop dimensions of the workload (W = $\{B, C, E, PX, PY, RX, RY, G\}$), and an appropriate set of unrolling factors is found automatically using a constraint solver. At a high level, this process determines which operations are performed in parallel and how the parallel operations are distributed across embedded blocks.

ALGORITHM 2: CNN Nested Loop Pseudo-code, unrolled across a total of eight MAC units in two embedded blocks

Layer Dimensions $(B, C, E, PX, PY, RX, RY, G) = < 4, 6, 6, 50, 50, 2, 2, 1 >$
Temporal tiling factors, $U_t = < 4, 5, 3, 50, 50, 1, 1, 1 >$;
Inter-EB unrolling factors, $U_o = < 1, 1, 2, 1, 1, 1, 1, 1 >$;
Intra-EB unrolling factors, $U_i = < 1, 1, 1, 1, 1, 2, 2, 1 >$;
for $b_t = 0$ *to 3* **do**
  for $c_t = 0$ *to 5* **do**
    for $e_t = 0$ *to 2* **do**
      for $px_t = 0$ *to 49* **do**
        for $py_t = 0$ *to 49* **do**
          $i_0 = I[b_t][c_t][px_t + 0][py_t + 0]$
          $i_1 = I[b_t][c_t][px_t + 1][py_t + 0]$
          $i_2 = I[b_t][c_t][px_t + 0][py_t + 1]$
          $i_3 = I[b_t][c_t][px_t + 1][py_t + 1]$
          EB 0:
          $o = i_0 \times W[2 * e_t + 0][c_t][0][0]$
          $o \mathrel{+}= i_1 \times W[2 * e_t + 0][c_t][1][0]$
          $o \mathrel{+}= i_2 \times W[2 * e_t + 0][c_t][0][1]$
          $o \mathrel{+}= i_3 \times W[2 * e_t + 0][c_t][1][1]$
          $O[b_t][e_t + 0][px_t][py_t] = o$
          EB 1:
          $o = i_0 \times W[2 * e_t + 1][c_t][0][0]$
          $o \mathrel{+}= i_1 \times W[2 * e_t + 1][c_t][1][0]$
          $o \mathrel{+}= i_2 \times W[2 * e_t + 1][c_t][0][1]$
          $o \mathrel{+}= i_3 \times W[2 * e_t + 1][c_t][1][1]$
          $O[b_t][2 * e_t + 0][px_t][py_t] = o$

The constraint satisfaction problem is formulated as a list of variables V, the domain of each variable D, and a set of constraints C. For this particular problem:

- V: The variables to solve for are the mapping vectors outlined in Section 3.4.1. These are the spatial and temporal unrolling factors that describe how the workload is distributed across embedded blocks and scheduled in time.

$$V = U_t \cup U_o \cup U_i \tag{6}$$

- D: D is the domain of each unrolling factor, where each variable can take on one value from the corresponding domain. In this case, each unrolling factor must be an integer between one and the corresponding high-level loop dimension (w_n). For example, a loop with bound of 100 could be parallelized across one to 100 embedded blocks. To improve solver speed, domains are further reduced to exclude unrolling factors that would lead to inefficient utilization of embedded blocks. In the example above, it would not be efficient to unroll the loop over 51 embedded blocks since this would use more resources than an unrolling factor of 50 with the same cycle count.

$$D_n = \{x \epsilon \mathbb{N} | x \leq w_n, ceil(w_n/(x-1)) > ceil(w_n/x)\} \tag{7}$$

- C: Constraints ensure that the spatial unrolling factors are legal given the number of EBs (C_{outer}), and the EB structure (C_{inner}) as discussed in Section 3.3. For example, if an embedded block multiplies a single input by two different weights (AP3 = 2), then the computation

of two output channels can be performed within the same embedded block, but not two separate batches. Additionally, the value of each high-level loop dimension must be the product of corresponding unrolling factors ($C_{product}$). Each of these constraints are listed below.

$$C_{outer} : \prod_{n=0}^{7} u_{o\_n} \leq \#\text{available embedded blocks} \tag{8}$$

$$C_{inner1} : u_{i\_5} \leq AP1^{1} \tag{9}$$

$$C_{inner1} : u_{i\_5} \leq AP1^{1} \tag{10}$$

$$C_{inner2} : u_{i\_1} \times u_{i\_6} \leq AP2^{1} \tag{11}$$

$$C_{inner3} : u_{i\_0} \times u_{i\_3} \times u_{i\_4} \leq AP3^{1} \tag{12}$$

$$C_{inner4} : u_{i\_2} \leq AP4^{1} \tag{13}$$

$$C_{inner5} : u_{i\_7} \leq AP5^{1} \tag{14}$$

$$C_{product} : u_{o\_n} \times u_{i\_n} \times u_{t\_n} \geq w_n, \forall n \epsilon [0, 7] \tag{15}$$

Finally, the resulting dataflow needs to be supported by the embedded block hardware; for example, if an embedded block can not accumulate outputs locally, then the input channel dimension cannot be unrolled temporally. This last class of constraints is applied on a case-by-case basis.

A Python CSP solver (python-constraint) is used to find all legal values of sets of values for U_t, U_o, and U_i. Solutions are ordered based on an estimate of the number of clock cycles that each one requires, to generate a shortlist of solutions that make efficient use of the available embedded blocks. By default, this cycle estimate is calculated using temporal tiling factors as well as the number of cycles required to pre-load weights into the embedded blocks and to perform each multiply accumulate operation. These cycle counts depend in turn on the latency across each embedded block, and specific timing relationships between embedded block inputs and outputs, which can be specified by the user to describe different embedded block implementations or to approximate the effect of adding pipeline registers into the datapath:

$$\text{Estimated Cycle Count} \approx \text{Weight\_Preload\_Cycles} + \prod_{n=0}^{7} u_{t\_n} * \text{Cycles\_per\_MAC} \tag{16}$$

This does not account for off-chip memory accesses. However, we also provide hooks allowing users to provide their own cost functions with which to order solutions.

In each of the case studies included in Section 4, this process could be completed within a reasonable timeframe. For instance, on a five core, 32GB computer, a mapping could be found for each individual layer within four minutes. As the number of layers, the sizes of layers and the size of the FPGAs grow however, the size of the search space also increases, which could make solving this problem using a constraint solver less feasible. Future work could include using heuristics to narrow the search space, or exploring different strategies such as using dynamic programming and other optimization algorithms.

[1] AP1-AP5 refer to the access patterns that describe the structure of available embedded blocks, as defined in Section 3.3.

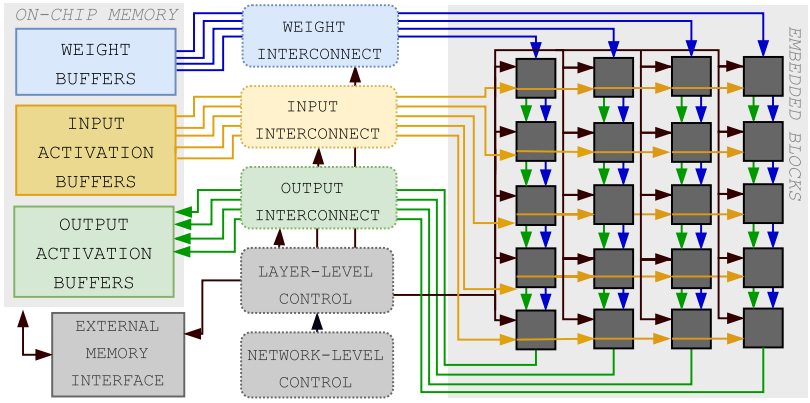

Fig. 5. Generated accelerator diagram.

3.4.3 RTL Generation Using pyMTL. The hierarchical structure of each circuit is modelled in parameterizable python pyMTL classes. Accelerators are modelled in Python using the pyMTL3 [19] infrastructure. Using pyMTL, the structure of each circuit component (corresponding to a Verilog module) is described using a python pyMTL class. PyMTL classes instantiate each other, creating hierarchical circuit structures. Connections between sub-modules as well as combinational and sequential logic are specified using pyMTL constructs. In this way, each circuit component described in Section 3.5 is described using a set of flexible parameterized pyMTL classes. Verilog circuits can then be automatically generated by instantiating these pyMTL models with the appropriate input parameters and generating Verilog through a call to the pyMTL API.

The "modelling towards layout" methodology of pyMTL is not yet widespread compared to either the traditional approach of writing Verilog directly, or using HLS techniques, but the pyMTL methodology and framework suited our purposes well for several reasons. First, compared with using System Verilog templates, specifying the structure and functionality of the circuit in python allows for much more complex generation logic to be integrated throughout the circuit model. pyMTL makes it possible to adjust generated circuits based on calls to python libraries for instance, or using other logic that would be extremely difficult using System Verilog parameters. Furthermore, ODIN does not support all System Verilog syntax, so typical System Verilog templates would not be synthesizable using VTR. Using pyMTL also facilitated the development of a comprehensive suite of unit tests in a unified framework. Our infrastructure has 97% statement coverage through unit tests and regression tests as a result, providing confidence in its reliability and in the functionality of the generated circuits. Finally, pyMTL allows the simulation flow described in Section 3.6.1 to be done in a unified framework rather than through a patchwork of scripts and proprietary simulation software.

3.5 Generated Circuit Structure

Figure 5 shows the general structure of the generated circuits. An array of interconnected embedded blocks is connected to banks of memory blocks containing weights and activations. Embedded tensor blocks, memory blocks and an external memory interface are coordinated by control logic, consisting of several state machines. Each component is described individually in more detail in the sections below.

3.5.1 On-chip Memory Banks. On-chip memory banks store input activations, filter weights, and output activations in on-chip buffers. The datawidth and the length of these buffers are

provided by the user based on the available FPGA architecture. In each generated circuit, a bank of on-chip buffers is instantiated and connected to the external memory interface. The number of buffers required is determined based on the number of input streams, output streams and weight streams connected to the embedded blocks, which in turn is governed by the mapping vector.

The bank of memories containing weights can optionally be double buffered, a technique used in many existing accelerators [1, 14, 16, 28, 36] to reduce bottlenecks associated with off-chip memory accesses. In this case, two sets of buffers are instantiated and additional logic is inserted to mux between them, allowing for weights to be read from one set of buffers while the second set is populated from off-chip.

3.5.2 Embedded Tensor Block Array. The total number of embedded tensor blocks instantiated in each generated accelerator is the product of Inter-EB unrolling factors, which specify how computations are unrolled spatially across embedded blocks. As discussed in Section 3.3, each embedded block could be a large embedded matrix multiplication array or as simple as a single DSP. If no embedded blocks are available in the target architecture, *logical embedded blocks* are automatically generated by our framework. Alternatively, the user can provide an equivalent soft-logic module themselves.

In other instances, existing embedded blocks must be supplemented with additional soft logic in order to perform the required functionality. For example, typical DNN layers include activation functions that are applied to the output activations. On hardware, these activation functions can be applied either in soft-logic modules, or within embedded blocks. If the available embedded blocks do not include hardware to perform activation functions, the benchmark generation framework generates additional soft-logic modules and connects them to embedded tensor block outputs, to implement the required activation functions.

3.5.3 External Memory Interface. An **external memory interface block (EMIF)** is connected to the on-chip memories through an Avalon Interface [9]. This external memory interface populates on-chip buffers with data from off-chip memory, and writes outputs to off-chip memory from on-chip output activation buffers. The data-width and address space are specified by the user, as well as the set of I/O connections between the external memory interface block and external memory. Typically EMIF IP cores are provided by the FPGA vendors, and may include both soft-logic and internal embedded block instantiations. We assume that such a block is available, and can be instantiated in generated benchmark circuits.

3.5.4 Interconnects. Three types of interconnect connect embedded tensor blocks with each other and with on-chip memory: a weight interconnect, an input activation interconnect, and an output activation interconnect. In each case, the required connections between embedded blocks depend on how computations are distributed spatially across EBs.

Input Activation Interconnect. The input activation interconnect connects on-chip buffers containing input activations with EBs. Inputs may also be cascaded between adjacent EBs, in cases where the same input channel feeds both adjacent blocks. Connections between input buffers and EBs are slightly more complex in the case of weight stationary 2-D convolution. In this situation, in order to convolve the stationary weight filter over different rows of the input feature map it may be necessary to select between multiple input streams. This is achieved by adding logic to multiplex between input activation buffers within the input activation interconnect. Figure 6 illustrates two examples in which convolutional DNN layers are distributed across a set of eight embedded blocks with different Inter-EB unrolling factors.

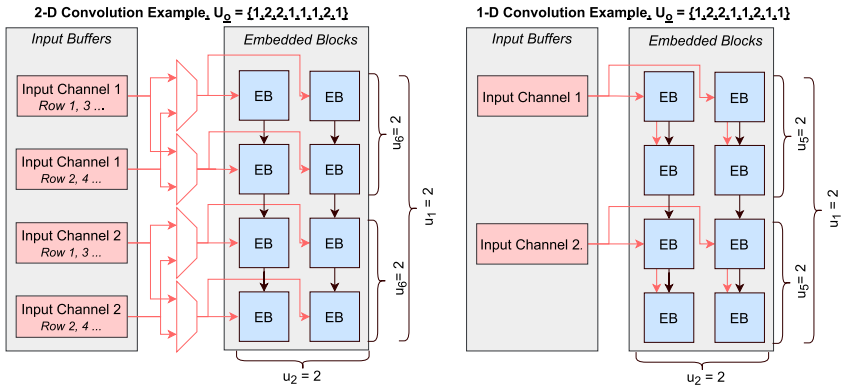

Fig. 6. Input interconnect examples.

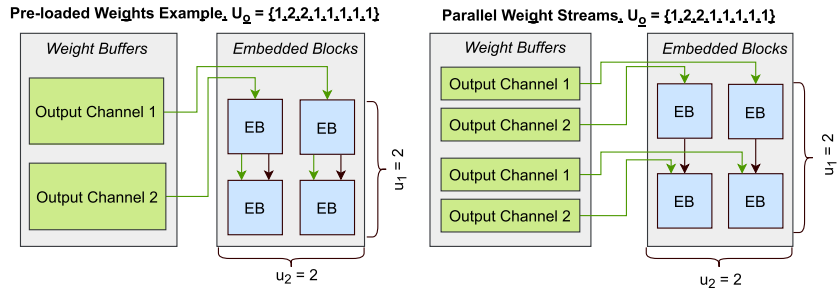

Fig. 7. Weight interconnect examples. On the left, weights are loaded into embedded blocks serially while on the right they are loaded into all embedded blocks in parallel.

Weight Interconnect. Proposed embedded blocks differ in terms of how weights are pre-populated into embedded blocks in the case of weight-stationary operation. In some cases [7, 23], local weights can be pre-populated through a single input port over the course of multiple cycles. Multiple embedded blocks may also be connected in chains through direct connections, allowing weights to be loaded serially through chains of multiple embedded blocks from a single weight buffer. This increases the total time required to pre-load weights into embedded blocks, but reduces the number of connections that must be routed between memory blocks and embedded blocks, potentially improving routability and critical path length. Alternatively, separate weight buffers can be connected to each embedded block individually to load all weights into embedded blocks in parallel. We support both use cases by allowing weights to be either pre-loaded serially through chains of embedded blocks or loaded in parallel to each embedded block. Both alternatives are illustrated in Figure 7.

Output Activation Interconnect. Output activation interconnects connect the outputs of embedded blocks to on-chip output activation buffers. In weight stationary dataflows, this interconnect also routes partial sums between embedded blocks. Figure 8 illustrates the required output interconnect for a set of four embedded blocks that compute two output channels in parallel.

3.5.5 State Machines. Control circuitry is generated to coordinate memory transfers between the off-chip memory interface and embedded memories, and to drive embedded block control signals. The layer-level control circuitry shown in Figure 5 is composed of several state machines,

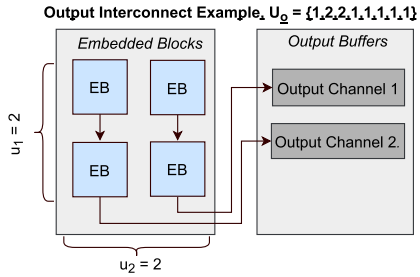

Fig. 8. Output interconnect example.

Table 3. Generated State Machines and Their Functions

Function	Description
Read off-chip weights	Populate on-chip weight buffers from off-chip memory through the External Memory Interface (EMIF). The location of weights in off-chip memory is user specified.
Read off-chip inputs	Populate input activation buffers from off-chip memory through the EMIF. The location of input activations in off-chip memory is user specified.
Load Weights into EBs	Stream weights from weight buffers into embedded tensor blocks. This process may be repeated with multiple batches of weights depending on the mapping vector U_t. In weight-stationary dataflows, this process is performed before streaming through input activations, but in output-stationary dataflows both processes run in parallel.
Load Inputs into EBs	Stream input activations from input buffers into embedded tensor blocks.
Retrieve outputs from EBs	Write output activations to on-chip output activation buffers. In output-stationary dataflows this is performed after streaming through multiple weights and inputs.
Write outputs off-chip	Write output activation off-chip through the EMIF. The location of output activations in off-chip memory is user specified.

parameterized and composed depending on the mapping vectors specified in Section 3.4.1. The functions of these state machines are summarized in Table 3.

Limitations. The main purpose of the framework presented in this paper is to facilitate the comparison of throughput, routability, and other performance metrics for different FPGA architectures. In this context, our priority is generating realistic circuits to reasonably evaluate metrics like resource usage, rather than generating circuits that will be reliably functional and highest possible performance.

In particular, we make assumptions about the input control signals of embedded blocks. Actual embedded blocks might require additional control signals that we do not take into consideration; for example, Intel's tensor slices may require control signals to select one of the several different operating modes that these blocks support. In comparison with the combined widths of the input activation, weight and output activation ports however, only a small percentage of an embedded block's ports are typically control signals. For example, in Xilinx's Ultrascale DSP48E2 block, for

which there is detailed publicly available documentation, out of a total of 399 input and output signals, 28 (7%) are control signals, not all of which would necessarily be required. For this reason, we assume that control signals play a smaller role in overall routability and congestion and focus on ensuring that the generated circuits are functional given a set of simplifying assumptions about the embedded blocks' control signals.

The generated state machine logic is also not highly pipelined or optimized. Realistically, there are many different ways to implement these types of state machines. State machines implementation differences could affect resource utilization or maximum clock frequency if the critical path is within the state machine logic (which would typically be mapped to logic elements rather than embedded blocks). Supplementing the benchmarks generated by our tool with existing benchmark suites, which contain many different examples of soft-logic and state machines, could help to provide a full picture of the performance of this state machine circuitry. The focus our framework is on comparing performance for circuits that primarily perform the highly-parallel tensor operations that characterize DNNs, potentially making use of embedded tensor blocks, rather than the more generic soft logic that is already well represented in existing benchmark suites.

3.6 Processing

3.6.1 Simulation Interfaces. Simulation of benchmark circuits is often not part of architectural studies, and even static benchmark suites are not necessarily extensively validated through simulation. Simulating the generated circuits provides several advantages however. Primarily, simulation demonstrates functionality and provides an extra level of confidence in the generated circuits. Secondly, it allows for simulation-based power estimation: power usage depends on switching behavior, which can be estimated through simulation. Finally, it could enable an analysis of both DNN accelerator accuracy (which may depend on data representation and precision) and low level performance metrics within a unified framework. Although simulation-based power estimation and accuracy evaluation are not currently part of our flow, having functional simulation makes these future extensions to the framework possible. For these reasons, we provide two levels of simulation:

- *PyMTL Cycle-Accurate Simulation:* Cycle-accurate simulation can be performed on pyMTL models directly, prior to Verilog generation. In this way, small circuits can be validated in python without requiring RTL simulation software. This method of simulation is not feasible for large designs but is useful for unit testing and validation.
- *RTL Simulation:* Generated Verilog can be simulated using a commercial RTL simulator, a more practical choice for larger designs. To validate the framework using large realistic networks, we simulate generated Verilog using Modelsim. A parameterized Verilog testbench and scripts to run Modelsim and extract output activations are provided to facilitate RTL simulation.

Both levels of simulation require simulation models of embedded blocks: BRAMs, EMIFs, and embedded tensor blocks. These Verilog models can be automatically generated based on the architectural descriptions provided by the user. Alternatively, the user could replace the generated simulation models with their own Verilog models, or with more realistic simulation models provided by vendors.

Prior to simulation, off-chip memory is populated with random input activations and filter weights. The expected DNN outputs are computed for this set of random inputs using functional python models (separate from pyMTL structural models) that are included as part of our framework. Simulation results are compared with these expected outputs to ensure that the generated accelerators produce the correct results.

3.6.2 Compiler Interfaces. Academic FPGA architecture research often uses the VTR CAD suite to synthesize, place and route designs during architecture research. VTR uses the ODIN compiler for synthesis, which only supports a subset of modern Verilog syntax. pyMTL generates synthesizable System Verilog however, including by default many System Verilog constructs that are not currently supported by ODIN. To resolve this, generated circuits are automatically post-processed to replace System Verilog constructs with ODIN-compatible Verilog. For example, the "logic" keyword is replaced with "wire" or "reg" as appropriate and parameters are removed and replaced with their values. Several other similar post-processing steps are applied to produce two versions of the circuit, one written in System Verilog and the other in an ODIN-compatible form.

In the release version of VTR, blocks are placed by VPR using a simulated annealing algorithm. While this placement strategy often places connected embedded blocks in nearby locations, they are not consistently placed in adjacent positions connected through direct connections. To ensure that embedded blocks are consistently placed in adjacent locations, taking best advantage of direct connections, we updated VTR to take as an input a set of placement constraints mapping blocks to locations. These placement constraints can be applied to a subset of the netlist, allowing us to manually place embedded blocks but use the regular VTR placement algorithms to place logic blocks. During benchmark generation, a set of placement constraints is also generated to place connected embedded blocks in adjacent locations whenever possible. While it is also possible to use our framework without these placement constraints and changes to VTR, they improve routability and routing utilization significantly.

3.7 Multi-Layer Networks

Different DNN network layers have different computational requirements, and therefore different optimal mapping vectors. A single accelerator (characterized by a single mapping vector) may therefore not be an efficient choice for all layers of a network. Existing accelerators typically use one of two strategies for accelerating multiple DNN layers:

- *Multiple Layers Executed Concurrently on Separate Hardware:* Using this strategy, different hardware accelerators are written (or generated) and optimized on a layer-by-layer basis. Multiple batches can then be pipelined and executed in parallel. This strategy maximizes efficiency of hardware utilization but is not suitable for applications that require low latency.
- *Layers Executed Sequentially on Single Accelerator:* Executing layers sequentially on the same hardware resources reduces latency but can lead to resource under-utilization when the accelerator structure is not well suited to all layers. Several papers [35, 36, 52] discuss and quantify this inefficiency. In [36], using the same unrolling scheme across all layers is found to cause a 1.7× performance slowdown, while in [35], utilization of DSPs is found to be only 24% on average for any given layer due to the variation in computational requirements across layers.

Recent work has explored flexible accelerators that execute layers sequentially on the same hardware resources while avoiding the hardware under-utilization that can be associated with this approach. On Application Specific Integrated Circuits (ASICs), several flexible interconnects have been proposed, allowing the accelerator dataflow to change dynamically from layer to layer [12, 16, 22, 24, 34]. Recent work [35, 36, 52] has also begun to address this issue for FPGA-based accelerators, proposing flexible accelerators that can adapt to the differing computational requirements of different DNN layers.

In the context of FPGAs with embedded tensor blocks, it would be ideal to optimize the mapping of DNN computations to available embedded blocks on a layer-by-layer basis, and to dynamically change the connections between blocks to implement different dataflows. This type of flexible

interconnect requires significantly more logic however, increasing resource utilization and lengthening timing delays. To help assess the feasibility of this solution on different FPGAs architectures, our infrastructure has the ability to generate accelerators that switch dynamically (temporally) between multiple mappings, allowing for different network layers to be mapped differently to the same resources. Each layer shares the same set of on-chip memories and embedded tensor blocks, but with different interconnects and state machine logic. An additional state machine is generated to switch between interconnects and layer-level control circuitry from layer to layer, as shown in Figure 5.

This strategy allows for different layers to reuse the same hardware resources efficiently, but requires additional soft logic and routing between the EBs and buffers. This is a particular problem in our current implementation, which does not take advantage of the similarities between different layers' connectivity requirements. There are several ways in which this could be improved to minimize the additional routing and logic required, which could be the subject of future work. In Section 4.5 we present data comparing the routability, congestion, and estimated latency of circuits generated using flexible interconnects.

4 CASE STUDIES AND RESULTS

To illustrate how our benchmark generation framework can be used to draw conclusions about FPGA architectures in the context of DNN acceleration, we present a set of case studies that use our framework. In the first, we demonstrate how real blocks from commercial FPGAs can be modelled and compared using our classification system. In the second, we vary different aspects of a set of embedded blocks and discuss the effects on overall performance. Finally, we explore different ways of mapping full DNN networks to the same FPGA resources and analyze the tradeoffs of each method.

4.1 Methodology

Baseline Architecture: Our baseline FPGA architecture includes an island-style fabric of logic blocks, memory blocks, and I/O pins arranged in columns, with unidirectional routing. The architecture grid dimensions are 149 × 177 and channel width is 250 unless otherwise specified. Timing delay estimates, switch-blocks, and connection blocks are based on the Stratix IV architecture. Although this is not one of the more recent commercial FPGA devices, the Stratix IV device can be captured accurately using VTR [26]. In comparison with more recent devices, Stratix IV timing delays are longer and overall accelerator performance is significantly slower. We also do not have timing models for new embedded tensor blocks that we model, and for these reasons, the maximum frequencies and critical path lengths listed in our case studies are not representative of expected results on more recent FPGAs, and do not necessarily reflect the performance benefits of exploiting embedded tensor blocks. Realistically, FPGA architects would have access to accurate timing estimates and this would not be a limitation.

Compilation: VTR 8.0.0 [26] was used to model FPGA architecture variants and to synthesize generated circuits. As outlined in Section 3.6.2, small adjustments were made to VTR to enable embedded block placement constraints.

DNN Workloads: In each case study, we base the DNN workloads on three layers of MobileNet [17]. The first is a fully connected layer, the second performs a 1 × 1 point wise convolution, and the third is a convolutional layer with a 3 × 3 filter. Unrolling factors are selected automatically to minimize estimated cycle counts. Loop bound vectors (W = {B, C, E, PX, PY, RX, RY, G}) of each layer are listed in Table 4.

Table 4. DNN Workloads, Based on Three MobileNet Layers

Layer	MobileNet Layer Index	Layer Type	Loop Bound Vector (W)
L1	20	Fully Connected	{1, 1024, 1000, 1, 1, 1, 1, 1}
L2	5	Pointwise Convolution	{1, 64, 128, 56, 56, 1, 1, 1}
L3	1	3×3 Convolution	{1, 3, 32, 224, 224, 3, 3, 1}

4.2 Case Study 1: Commercial Embedded Blocks

To demonstrate how real embedded machine learning blocks can be defined using our classification system, this case study compares a more traditional DSP block with an embedded tensor engine, based on Xilinx's Ultrascale DSP slices and Intel's Stratix 10 NX AI Tensor Blocks, respectively. We show how our framework can be used to generate accelerators for three layers of MobileNet, and compare performance results. We do not have access to the actual timing delays of these blocks, their area footprints, or connectivity to surrounding fabric, so it is important to note that this case study is an example of how such a comparison might be performed rather than an accurate evaluation of real architectures.

4.2.1 *Candidate Architectures.*

Tensor Block Model: Intel's recent AI Tensor Blocks can operate in several modes. We model these blocks operating in "tensor mode" according to our understanding of publicly available documentation [7, 23]. Assuming 8-bit precision is used, each block computes three dot products of 10-element input vectors with three 10-element weight vectors. The access patterns are therefore {AP1, AP2, AP3, AP4, AP5} = {1, 10, 3, 1, 1}.

In this case study, weights are pre-loaded into all tensor blocks in parallel through 16-bit input ports. Accelerators are constructed of chains of tensor blocks, with partial sums cascaded through direct connections between adjacent blocks. We assume that input activations can also cascade between adjacent tensor blocks through direct connections, facilitating input reuse during convolution (as is the case for DSPs [32]). Otherwise, additional logic could be automatically inserted, allowing for input reuse, but increasing logic utilization.

The structure of the resulting accelerators is very similar to the example of the intended use of Stratix 10 NX AI tensor blocks provided in [23].

DSP Model: We base our DSP blocks on Xilinx UltraScale DSP48E2 blocks, each of which contains a 27×18 multiplier and a 48-bit accumulator. For 8-bit inputs, each DSP can be used to multiply one weight by two input activations. All inputs can be registered, and the outputs can be accumulated. The access patterns ({AP1, AP2, AP3, AP4, AP5}) are therefore {1, 1, 1, 2, 1}.

Generated circuits connect adjacent DSPs in an array of chains. The resulting accelerators are similar to the chains of DSPs used in [32], an accelerator that aims to make use of the dedicated interconnects between DSPs to maximize frequency using more traditional FPGA building blocks.

VTR Architecture: In the VTR architecture models used in this case study, we aim for roughly the same resource mix as the Stratix 10 NX architecture (albeit at a quarter of the size to allow for faster synthesis), which has 70,272 logic blocks, 6,847 M20Ks, and 3,960 AI tensor blocks. We compare two architectures. The first has 17,657 logic blocks, 1,720 RAM blocks, and 989 AI tensor blocks. In the second, each tensor block is replaced with two DSPs. The resulting DSP count is comparable to other Stratix 10 devices with compute intensive resource mixes, but the total number of multiply accumulate units is much lower than in the architecture that includes tensor blocks, since each tensor block has 15× more multipliers than a single DSP. As shown in Figure 9, RAM blocks, DSPs and AI Tensor Blocks are arranged in columns, which are distributed evenly across the device.

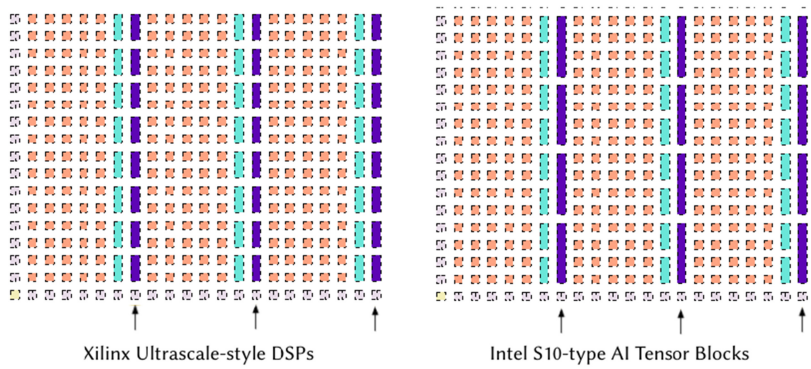

Fig. 9. VTR architecture models with Intel S1-0-type AI tensor blocks and Xilinx Ultrascale-style DSPs.

Table 5. Implemented Accelerator Mapping Vectors and Pre-Compilation Metrics

EB type	Layer	U_i	U_o	U_t	MAC count	MAC utilization	Estimated Cycle Count
Tensor Block	1	{1, 10, 3, 1, 1, 1, 1, 1}	{1, 26, 38, 1, 1, 1, 1, 1}	{1, 4, 9, 1, 1, 1, 1, 1}	29640	99.9%	566
	2	{1, 10, 3, 1, 1, 1, 1, 1}	{3, 7, 43, 1, 1, 1, 1, 1}	{1, 1, 1, 19, 56, 1, 1, 1}	27090	91.3%	1086
	3	{1, 3, 3, 1, 1, 1, 3, 1}	{1, 1, 11, 1, 28, 3, 1, 1}	{1, 1, 1, 224, 8, 1, 1, 1}	24948	84.1%	1810
DSP	1	{1, 1, 2, 1, 1, 1, 1, 1}	{1, 79, 25, 1, 1, 1, 1, 1}	{1, 13, 20, 1, 1, 1, 1, 1}	3950	99.8%	1524
	2	{1, 1, 2, 1, 1, 1, 1, 1}	{1, 64, 5, 3, 2, 1, 1, 1}	{1, 1, 13, 19, 28, 1, 1, 1}	3840	97.1%	6916
	3	{1, 1, 2, 1, 1, 1, 1, 1}	{1, 3, 8, 1, 9, 3, 3, 1}	{1, 1, 2, 224, 25, 1, 1, 1}	3888	98.3%	11200

Table 6. Implemented Accelerator Post Compilation Metrics (Resource Utilization)

EB type	Layer	M20K Usage	Tensor Block Usage	LAB Usage	Short Wire utilization	Long Wire utilization	Average Wire Length
Tensor Block	1	1175 (68.3%)	988 (99.9%)	1333 (7.5%)	20.9%	9.8%	22.8
	2	1199 (69.7%)	903 (91.3%)	1758 (9.9%)	19.5%	9.1%	26.1
	3	1242 (72.2%)	924 (93.4%)	2967 (16.8%)	14.5%	7.5%	13.0
DSP	1	1053 (61.2%)	1975 (99.8%)	1171 (6.6%)	9.6%	10.6%	22.4
	2	1084 (63%)	1920 (97.1%)	1214 (6.9%)	19.1%	9.4%	33.5
	3	657 (38.2%)	1944 (98.3%)	950 (5.4%)	14.7%	6.4%	22.7

4.2.2 Results. Table 5 includes the mapping vectors selected for each workload, and estimated cycle counts, as defined in Section 3.4.2. Table 6 lists resource utilization of each design and lower level implementation metrics. In this example, timing results (included in the Appendix in Table 13) do not reflect the actual performance of either block, since timing delays are estimated based on Stratix IV. For this reason, rather than including critical path delays, which are dependent on accurate timing delays, we focus instead on wire utilization metrics and average wire length. Average wire length is reported in terms of routing grid units.

4.2.3 Analysis. All generated accelerators have relatively high utilization of available MAC units, with utilization over 80% for all workloads. The MAC units of DSPs are used slightly more efficiently, since the hardened structure of the larger AI tensor blocks limits the possible spatial unrolling factors. Despite the fact that they are less efficiently utilized, AI tensor blocks provide a much higher number of MAC units in total (7.5×), allowing for higher spatial unrolling factors

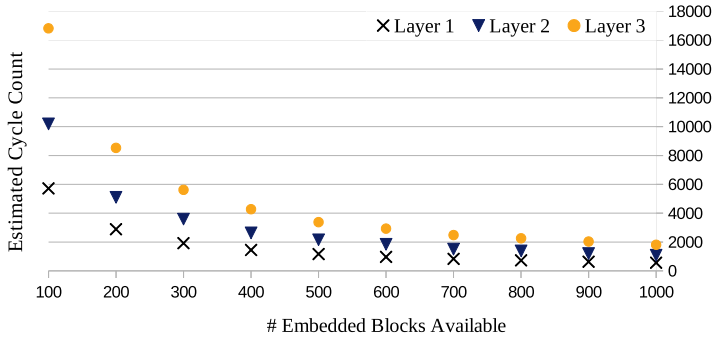

Fig. 10. Impact of embedded block count on total estimated clock cycles.

overall. This results in a significantly lower total estimated cycle count when AI tensor blocks are used as opposed to DSPs.

All generated circuits are routeable, with average routing utilization under 20% for all circuits.

4.3 Case Study 2: Architectural Trade-off Analysis

This case study demonstrates the use of our framework to evaluate several different design trade-offs for a heterogeneous FPGA architecture with embedded tensor blocks. The number of embedded blocks available, input activation quantization, and embedded block layout are varied to evaluate the impact of these parameters on overall performance. In each case, we use the same baseline architecture and workloads described in Section 4.1. Embedded blocks are based on Intel's ML Tensor Blocks, as outlined in Section 4.2.1.

4.3.1 Resource Mix Analysis. An important architectural consideration is the mix of resources included on the chip. Increasing the number of embedded tensor blocks could allow for more spatial parallelism but decreases the amount of general purpose FPGA building blocks available.

We first evaluate the effect of varying the number of embedded blocks available on estimated total cycle count and resource usage, assuming that all MAC operations are performed in embedded blocks rather than in the FPGA fabric. As shown in Figure 10, on FPGA architectures with more embedded blocks, workloads can be parallelized with larger unrolling factors, resulting in lower total estimated cycle counts. Circuits using fewer embedded blocks generally use fewer memory blocks and less general purpose logic, although the number of resources required is also dependent on which of the DNN layer dimensions are unrolled. Resource utilization, cycle counts, and routing metrics for each workload and architecture variant are listed in Appendix A, in Table 14, Table 15 and 16, respectively.

On an FPGA architecture with embedded tensor blocks, the FPGA user could still opt to implement some tensor operations using soft logic (in *logical embedded blocks*) instead of exclusively in hardened logic. We generate five sets of circuits with 1500 MAC units but with different proportions of tensor operations performed in soft logic. As shown in Figure 11, implementing computations in general purpose logic significantly increases utilization of logic blocks and of routing resources, even for relatively small circuits. For larger circuits, implementing a significant percentage of tensor operations in soft logic could quickly become impossible for this reason. We expect the high logic and routing utilization in these cases would also further decrease maximum clock frequency, but it becomes unfeasibly time consuming to compile such large soft-logic designs using ODIN, which is why in this case study we use relatively small designs.

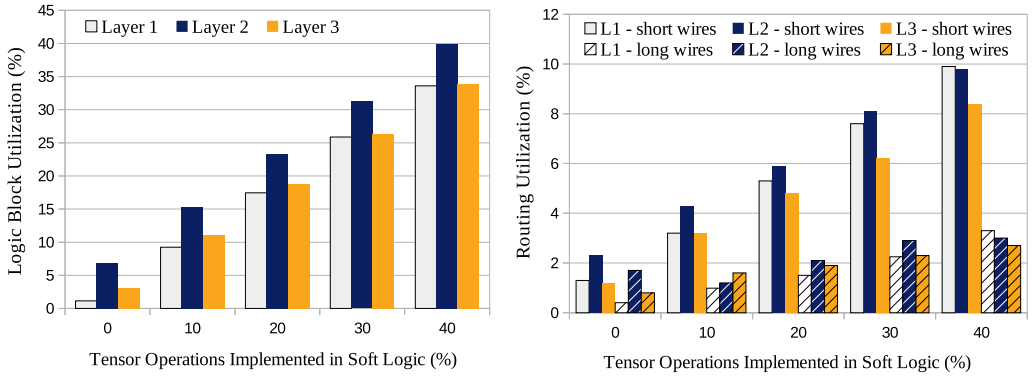

Fig. 11. Routing utilization and logic block utilization with different proportions of tensor operations implemented in soft-logic.

Table 7. Maximum Clock Frequency with Logical Embedded Blocks

Layer	Embedded Blocks Only	10% Soft Logic	20% Soft Logic	30% Soft Logic	40% Soft Logic
Layer 1	148.6 MHz	139.4 MHz	139.2 MHz	139.4 MHz	139.4 MHz
Layer 2	149.7 MHz	139.2 MHz	139.2 MHz	143.5 MHz	143.7 MHz
Layer 3	150.5 MHz	139.2 MHz	145.2 MHz	142.1 MHz	143.7 MHz

Implementing tensor operations in soft logic also decreases the maximum clock frequency (listed in Table 7) although as noted previously, in our case studies the maximum clock frequency is calculated using timing delays based on Stratix IV. Timing results would realistically be quite different for FPGA implemented on the latest process nodes.

4.3.2 Quantization Analysis. Decreasing the precision of activations and weights can allow for more operations to be performed with the same area and power footprint. Here, we investigate adjusting the structure of the embedded tensor blocks to operate on input activations with different bitwidths, including binarized inputs (1-bit quantization). The baseline embedded blocks are based on Intel's AI Tensor blocks, which perform three dot products on the same ten-element input vector and three different sets of weights ($\{AP1, AP2, AP3, AP4\} = \{10, 3, 1, 1, 1\}$). For these tensor blocks, each input activation is an eight-bit value. The input port of each embedded block is therefore 80 bits in total. In this case study, we maintain the same input port width but change the bit-width of individual activation values. For example, instead of operating on ten 8-bit input activations, the embedded blocks could operate on twenty 4-bit activations without changing the width of the input or output ports ($\{AP1, AP2, AP3, AP4\} = \{20, 3, 1, 1, 1\}$).

Resource and routing utilization with different input activation precisions is listed in Tables 17 and 18 of Appendix A, while Figure 12 shows estimated cycle count. In principle, decreasing bitwidth and increasing the number of MAC units within each embedded block should allow for more spatial parallelism and lower estimated cycle counts. As shown in Figure 12 however, this is not always the case for Layer 1 or Layer 3.

In Layer 3, decreasing input activation bitwidth below eight does not improve performance. In this layer, each output is produced based on only nine input activations, so changing the embedded blocks to operate on more than nine input activations does not result in any performance improvement. In Layer 1, changing the input activation has little effect on total cycle count for a different reason. This is a fully connected layer without local weight reuse, so loading the embedded blocks with weights consumes a large percentage of the total estimated cycle count. In this particular case,

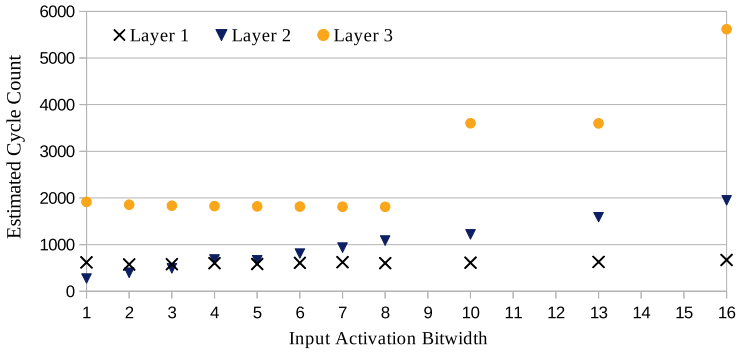

Fig. 12. Impact of precision on estimated cycle count.

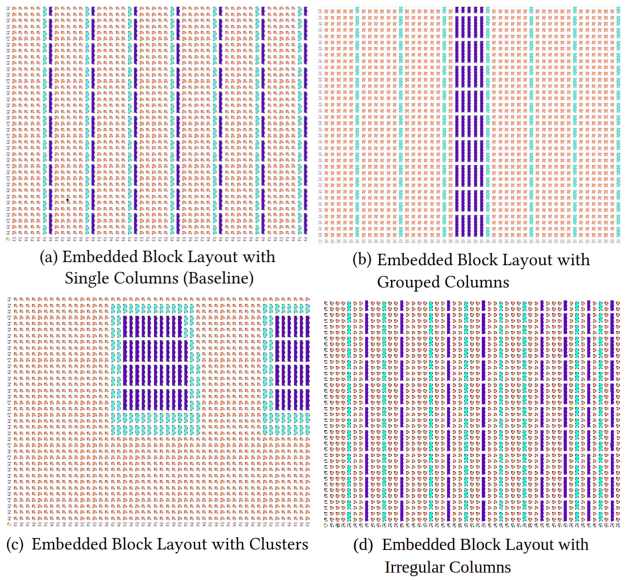

Fig. 13. Four candidate embedded block layouts. Only the bottom-left corner of each architecture is pictured.

increasing the number of operations performed within each embedded block increases the number of clock cycles required to load weights, offsetting the benefits of the increased MAC count.

This analysis demonstrates that quantization can have a significant effect on performance (particularly for Layer 2 and for Layer 3) but that simply decreasing bitwidth is not sufficient to improve performance for all workloads. To fully understand the effect of these types of changes on overall performance it is necessary to closely consider different workloads and how they could realistically be mapped to hardware.

4.3.3 Layout Analysis. In this example, we vary how embedded tensor blocks are arranged across the FPGA device. Since embedded tensor blocks are typically interconnected and may be driven by the same inputs and control signals, one strategy could be to group embedded tensor blocks as close together as possible. Routing congestion also needs to be taken into account however. In this example, we consider four different high-level layouts with the same total number of embedded blocks (989), each of which is illustrated in Figure 13.

Table 8. Routing Results for Different Embedded Block Layouts

		Single Columns	Grouped Columns	Clusters	Irregular Columns
Layer 1	Short Wire Utilization	16%	16%	20%	16%
	Long Wire Utilization	8%	7%	10%	9%
	Max Routing Utilization	50%	63%	70%	49%
	Average Wire Length	21.0	20.6	26.5	21.1
Layer 2	Short Wire Utilization	15%	15%	20%	15%
	Long Wire Utilization	7%	7%	8%	7%
	Max Routing Utilization	46%	59%	69%	43%
	Average Wire Length	22.2	22.2	28.3	22.1
Layer 3	Short Wire Utilization	11%	12%	20%	11%
	Long Wire Utilization	6%	5%	10%	6%
	Max Routing Utilization	45%	69%	70%	44%
	Average Wire Length	10.8	20.6	26.5	10.6

In the baseline layout, embedded blocks are arranged in 23 columns, which are spaced evenly across the device in the x-dimension. In the second, columns are combined together into groups of four, while in the third, embedded blocks are arranged in evenly spaced clusters. In the last architecture variant, single columns are spread irregularly across the device, similarly to how DSP columns are arranged in Xilinx Ultrascale+. The internal structure of the embedded tensor blocks is the same for each layout, with each tensor block performing up to thirty multiplications in parallel. In each case, the activations of adjacent embedded blocks are passed between embedded tensor blocks through direct connections, bypassing the segmented routing network. Placement constraints are generated to ensure that blocks are placed appropriately to make use of these connections.

Routing utilization and average path length are listed in Table 8 for each proposed layout, with a channel width of 300. On average, maximum routing utilization is significantly lower for the baseline architecture and irregular architecture, in which embedded blocks are laid out in single columns. In the clustered and grouped layouts, while the connected tensor blocks are in close proximity, general purpose routing is still required to transmit activations, partial sums, weights and control signals in cases where connected embedded blocks are not directly adjacent to each other. In the clustered and grouped layouts, this routing is concentrated in areas around embedded blocks, leading to high congestion and a significantly higher maximum routing utilization.

4.4 Case Study 3: Workload Comparison

While we do not support all possible workloads and layer types, as discussed further in Section 5.1, our benchmark generation framework supports many different DNN layer types that are not represented in the previous case studies. The purpose of this case study is to demonstrate the automatic generation of a wider variety of different workloads, and to show how both high level cycle count and lower level metrics like routing utilization are dependent on both the available hardware and the workload. Specifically, in addition to the workloads outlined in Section 4.1, we include depthwise layers, different filter dimensions, input and output channels, strides, dilation and activation functions. Each layer workload is defined in Table 9. In each case, we use the same baseline architecture described in Section 4.1. Embedded blocks are based on Intel's ML Tensor Blocks, as outlined in Section 4.2.1.

4.4.1 Results. Tables 10 lists both pre-compilation and post-compilation results for each workload described in Table 9. These results include the estimated cycle counts, as defined in

Table 9. Additional Workload Definitions

Layer Name	Layer Type	Workload Dimensions	Additional Parameters	Description
C1		{1, 3, 32, 224, 224, 2, 2, 1}	stride = 1; dilation = 1	2 × 2 Filter
C2	Convolutional	{1, 3, 32, 224, 224, 3, 3, 1}	stride = 1; dilation = 1	3 × 3 Filter
C3		{1, 3, 32, 224, 224, 3, 3, 1}	stride = 2; dilation = 1	3 × 3 Filter, with stride
C4		{1, 3, 32, 224, 224, 3, 3, 1}	stride = 1; dilation = 2	3 × 3 Filter, with dilation
C5		{1, 3, 32, 224, 224, 4, 4, 1}	stride = 1; dilation = 1	4 × 4 Filter
C6		{1, 3, 32, 224, 224, 5, 5, 1}	stride = 1; dilation = 1	5 × 5 Filter
C7		{1, 3, 32, 224, 224, 6, 6, 1}	stride = 1; dilation = 1	6 × 6 Filter
P1		{1, 64, 124, 56, 56, 1, 1, 1}	activation = NONE	No activation function
P2	Pointwise	{1, 64, 124, 56, 56, 1, 1, 1}	activation = RELU	ReLU activation
P3		{1, 64, 124, 56, 56, 1, 1, 1}	activation = CLIPPED	Clipped activation
FC1		{1, 1024, 1000, 1, 1, 1, 1, 1}	stride = 1; dilation = 1	Fully Connected
FC2	Fully Connected	{1, 1024, 2000, 1, 1, 1, 1, 1}	stride = 1; dilation = 1	FC, more output channels
FC3		{1, 1024, 500, 1, 1, 1, 1, 1}	stride = 1; dilation = 1	FC, fewer output channels
FC4		{1, 2048, 1000, 1, 1, 1, 1, 1}	stride = 1; dilation = 1	FC, more input channels
FC5		{1, 512, 1000, 1, 1, 1, 1, 1}	stride = 1; dilation = 1	FC, fewer input channels
D	Depthwise	{1, 3, 32, 224, 224, 1, 1, 4}	stride = 1; dilation = 1	Depthwise

Section 3.4.2 as well as resource utilization of each design and lower level implementation metrics such as average wire length, which is reported in terms of routing grid units.

Embedded tensor block utilization and MAC utilization is listed in Table 10, and while typically high, it varies significantly between different layers depending on how efficiently the workload can be implemented on the available resources. For example, the MAC utilization of workloads C1, which performs convolution with a 2 × 2 filter, is particularly low, since in this instance it is not possible to use more than 12 of the 30 MAC units within each embedded block.

Estimated cycle count is also highly dependent on the workload, in part because of the different rates of tensor block utilization, but primarily because different layers involve differing numbers of computations. Metrics like LAB Usage are also dependent on the workload. For example, applying different activation functions to otherwise identical layers results in different rates of LAB utilization, since the activation functions are implemented using soft logic.

4.5 Case Study 4: Multi-Layer Accelerators

As discussed in Section 3.7, accelerating dissimilar DNN layers using the same mapping to hardware can result in under-utilization of resources. In this case study, we illustrate this under-utilization using Layer 1 (a fully connected layer) and Layer 3 (a convolutional layer), two particularly dissimilar workloads. Table 11 lists the optimal mappings for each individual layer, the optimal mapping for the combined layers, and resulting estimated cycle counts in each case.

As described in Section 3.7, our framework can be used to generate circuits that reuse the same embedded blocks between layers but with different parallelization schemes and interconnects. We generate one accelerator that uses this strategy to dynamically switch between the two optimal mapping vectors listed in Table 11, and one which uses the same static mapping for both layers. The FPGA architecture used is the the same baseline architecture described in Section 4.1, with 500 embedded tensor blocks based on Intel's Stratix 10 NX AI Tensor Blocks and a channel width of 400. The resulting resource utilization, routing utilization, and difference in average path delay are listed in Table 12. Dynamically changing interconnects across layers significantly increases the soft logic required, since the generated accelerator includes more state machine logic and additional soft logic in the input, weight, and output interconnects. Routing requirements also increase dramatically since inserting additional logic between embedded blocks means that the direct

Table 10. Implemented Accelerator Pre and Post-Compilation Measurements

Layer	Tensor Block Usage	MAC Usage	LAB Usage	Estimated Cycle Count	Short Wire utilization	Long Wire utilization	Average Wire Length
C1	924 (93%)	56%	4461 (25%)	1252	16.5%	8.7%	13.1
C2	924 (93%)	84.1%	2970 (17%)	1810	14.5%	7.5%	13.0
C3	924 (93%)	84.1%	2948 (17%)	1810	14.1%	7.4%	10.9
C5	968 (98%)	88.1%	1796 (10%)	4762	15.4%	6.6%	11.5
C6	880 (89%)	80.1%	1406 (8%)	6332	8.6%	4.0%	7.0
C7	924 (93%)	84.1%	1306 (7%)	7230	11.9%	5.9%	9.2
P1	903 (91%)	91.3%	1667 (9%)	1086	18.0%	8.6%	25.2
P2	903 (91%)	91.3%	1758 (10%)	1086	19.5%	9.1%	26.1
P3	903 (91%)	91.3%	2295 (13%)	1086	18.4%	8.6%	22.8
FC1	988 (100%)	99.9%	1333 (8%)	566	20.9%	9.8%	22.8
FC2	975 (99%)	98.6%	1544 (9%)	1165	17.4%	7.1%	18.4
FC3	980 (99%)	99.1%	1606 (9%)	323	19.5%	9.9%	21.0
FC4	984 (99%)	99.5%	1673 (9%)	1161	17.7%	9.6%	19.0
FC5	988 (100%)	99.9%	1574 (9%)	314	20.4%	10.2%	21.9
D	903 (91%)	91.3%	1658 (9%)	4313	18.0%	8.7%	25.2

Table 11. Mappings and Estimated Cycle Count for Two-Layer Workload

	Dynamic Layer-by-Layer Mappings			Optimal Static Mapping		
	U_o	U_i	Estimated Cycle Count	U_o	U_i	Estimated Cycle Count
Layer 1	{1, 13, 38, 1, 1, 1, 1, 1}	{1, 10, 3, 1, 1, 1, 1, 1}	1165	{1, 1, 26, 2, 1, 3, 3, 1}	{1, 10, 3, 1, 1, 1, 1, 1}	21433
Layer 3	{1, 1, 11, 15, 1, 3, 1, 1}	{1, 3, 3, 1, 1, 1, 3, 1}	3378			25112
			Total: 4543			Total: 46545

Table 12. Implemented Accelerator Post Compilation Metrics (Resource Utilization)

Multi-layer accelerator type	M20K Usage	Tensor Block Usage	LAB Usage	Short Wire Utilization	Long Wire Utilization	Maximum Channel Utilization
Dynamic	817 (47%)	495 (99%)	5067 (29%)	20%	18%	82%
Static	342 (20%)	468 (93.6%)	618 (4%)	3%	2%	34%

connections between adjacent blocks can not be used. Future work using our benchmark generation framework could explore different routing architecture alternatives to improve the feasibility of these kinds of dynamic interconnects.

5 CONCLUSIONS AND FUTURE WORK

Deep Neural Networks are machine learning applications with computation demands that cannot be met efficiently by conventional Central Processing Unit (CPU)-based platforms. FPGAs-based hardware accelerators present an attractive solution; FPGAs provide high-performance and power efficiency as well as reconfigurability, an important consideration in the quickly evolving field of deep learning. For this reason, DNN acceleration is increasingly a focus of FPGA research, including architectural studies that aim to design FPGAs that are better suited to these workloads.

In this paper, we have discussed how the evaluation of FPGA architectures in the context of DNN acceleration is challenging, due in part to a lack of suitable benchmark circuits. Not only do

publicly available DNN benchmark circuits not exist, the types of static benchmark suites typically used in FPGA research are not sufficient in this context. Static benchmark circuitry and existing benchmark circuit generators do not target the types of embedded blocks that have been proposed for improving performance of FPGA-based DNN accelerators. This is a practical impediment for FPGA architects, who need to hand write benchmark circuits as a result, which can be very time consuming. This also makes it difficult to evaluate architecture tradeoffs and compare different proposals in a unified framework.

We propose an architectural exploration flow in which benchmark circuits are generated to target the architecture under consideration, which we presented in Section 3. In Section 4 we also provide case studies that use our FPGA architecture exploration flow to evaluate design tradeoffs and to explore the relationships between FPGA architecture, DNN workload, and resulting performance.

5.1 Limitations and Future Work

Through our case studies we demonstrate how our framework can be used to explore several different aspects of FPGA architecture design. However, the current framework has limitations that currently restrict its applicability in some areas, and which could be improved to make this work more broadly useful. These limitations and potential improvements are summarized below.

Flexibility and Support for Different Use Cases: Our taxonomy of embedded blocks covers a wide range of different tensor operations and our framework supports the majority of embedded tensor blocks proposed to date. There are types of embedded blocks that cannot be modelled using our parameterized definition however. In [2] for example, the proposed embedded blocks include logic to read and write to on-chip memory, whereas in our benchmark generation flow, we assume that this type of logic is implemented using the reconfigurable FPGA fabric instead of within embedded blocks. Future extensions to the framework could expand the embedded block definition to better handle this particular case.

Similarly, while we consider the most common types of DNN layers, there are activation functions and DNN layer types that are not currently supported. New activation functions can be easily added to the existing library of activation functions. In other cases however, such as for 3-D convolutions and transposed convolution, our workload definition is not sufficient and support would require more extensive changes to the workload definition presented in Section 3.2. Another current limitation is the ability of our framework to efficiently handle sparse networks. Many high performance accelerators exploit the sparsity of weights and activations values using various different optimization strategies, including several different ways of compressing weights and/or activations and eliding zero-value multiplications, each of which would require different adjustments to either the generation framework or manual changes to the benchmarks themselves. Ongoing work aims to improve the modularity of the generated benchmarks and the framework itself to facilitate adding additional custom logic to generated circuits, facilitating the implementation of sparsity optimizations.

Generated Accelerator Quality and Performance: Circuits generated through the proposed benchmark generation tool are designed to make realistic use of embedded blocks in functional circuits, to estimate the performance of real user designs. The focus was not to generate circuits that compete with hand-written accelerators in terms of performance. In particular, while we pay particular attention to different dataflows and unrolling factors, since these determine how the embedded blocks are connected and the overall structure of the circuit, we focus less on other aspects of DNN accelerator design, such as efficiently reading and writing from off-chip memory. For example, selectively keeping some inter-layer activations on-chip between layers has been shown to improve

performance [4], but this kind of optimization does not affect the connections between embedded blocks and memory. Further, the state machines generated are functional but are not optimized to minimize clock frequency through hyper-pipelining, as discussed in Section 3.5.5.

Existing DNN circuit generators [41, 47, 48, 51] typically focus more on the performance and practical reliability of the generated circuits. These generators are intended to reduce development time of FPGA users, and are not intended for FPGA architecture exploration. Improving the framework proposed in this work to take into consideration other common performance optimizations would allow for it to be used for DNN circuit generation outside of the context of FPGA architecture research. Significant practical improvements to the code base would be required, but our conceptual framework and problem space definition could be a good starting point for this work. In particular, using HLS instead of pyMTL to generate circuits could allow us to take advantage of performance optimizations performed in HLS software, provided it is possible to model and instantiate embedded blocks with sufficient low-level control. For example, using HLS for circuit generation would allow us to take advantage of existing HLS pipelining optimizations.

Case Study Limitations: While our case studies effectively demonstrate how our framework could be used by FPGA designers to compare different architecture candidates and evaluate design tradeoffs, the results themselves are highly dependent on assumptions about timing delays and the baseline architecture. Realistically, during FPGA architecture exploration, vendors and researchers would calculate timing delays of each FPGA building block based on detailed models. We do not have access to these types of timing models, or detailed information about all components of the latest commercial FPGA architectures and frequency measurements are therefore not representative of modern devices. On these grounds, the provided case studies focus primarily on resource utilization, routability and average wire length, although frequency can also be measured if accurate timing delays are available. VTR also does not support all architectural features of the latest commercial devices, such as pipeline registers in the routing fabric, so it is not possible to fully and accurately model the latest commercial architectures in VTR. For these reasons, our case studies are intended to provide a blueprint for how FPGA vendors could evaluate architectural proposals, rather than to draw concrete conclusions about which architecture performs best.

Ideally it would be possible to compare performance of generated circuits against existing benchmark circuit suits and circuit generation frameworks. This would allow us to check the accuracy of low level metrics like clock frequency as well as clock cycle counts. At this time however, existing benchmark suites (and circuits produced by existing generators) do not explicitly instantiate embedded blocks, and compilers cannot automatically infer these blocks.

APPENDIX

A CASE STUDY RESULTS - EXTENDED

Table 13. Case Study 1 – Maximum Frequency with Different Embedded Tensor Blocks

Workload	Tensor Block	DSP
Layer 1	146.0 MHz	146.7 MHz
Layer 2	139.2 MHz	147.7 MHz
Layer 3	150.5 MHz	146.0 MHz

Table 14. Case Study 2.1 – Resource Utilization with Different Numbers of Embedded Blocks

Available Embedded Blocks		1000	900	800	700	600	500	400	300	200	100
Layer 1	Tensor Block Usage	988	896	784	668	588	494	392	300	196	100
	LAB usage	1333	1561	1442	2250	1109	888	790	631	456	297
	M20K usage	1175	1104	987	1179	749	616	511	405	273	155
Layer 2	Tensor Block Usage	903	882	770	693	539	462	378	280	196	98
	LAB usage	1758	2240	1818	2244	1788	1233	1461	1648	1388	1270
	M20K usage	1199	1351	988	1081	875	716	708	750	630	560
Layer 3	Tensor Block Usage	924	825	759	693	594	495	396	297	198	99
	LAB usage	2967	2663	2451	2249	1954	1643	1331	1024	722	412
	M20K usage	1242	1116	1032	948	822	696	570	444	318	192

Table 15. Case Study 2.1 – Cycle Count with Different Numbers of Embedded Blocks

Available Blocks	Embedded	1000	900	800	700	600	500	400	300	200	100
Layer 1		566	632	727	834	967	1165	1447	1919	2887	5717
Layer 2		1086	1202	1381	1511	1844	2158	2636	3592	5100	10193
Layer 3		1810	2034	2258	2482	2930	3378	4274	5618	8530	16818

Table 16. Case Study 2.1 – Routing Utilization with Different Numbers of Embedded Blocks

Available Embedded Blocks		1000	900	800	700	600	500	400	300	200	100
Layer 1	Short Wire Util	20.9%	15.4%	13.5%	14.1%	10.3%	8.6%	7.3%	5.4%	3.9%	2.1%
	Long Wire Util	9.8%	5.2%	4.3%	4.4%	3.6%	3.5%	2.4%	2.7%	1.5%	0.9%
	Avg Wire Length	22.8	17.3	17.1	18.1	17.4	17.7	18.2	18.3	19.1	19.5
Layer 2	Short Wire Util	19.5%	42.6%	28.1%	24.8%	21.9%	10.8%	21.6%	12.1%	8.9%	4.5%
	Long Wire Util	9.1%	35.1%	11.8%	12.7%	10.4%	4.7%	16%	6.8%	3.8%	1.4%
	Avg Wire Length	26.1	54.2	39.8	38.7	43.4	24.9	54.6	57.1	50	35.1
Layer 3	Short Wire Util	14.5%	18.6%	16.8%	13.1%	12.8%	9.03%	7.8%	5.7%	3.6%	1.8%
	Long Wire Util	7.5%	8.9%	7.8%	6.72%	6.1%	3.95%	3.1%	2.4%	1.5%	0.9%
	Avg Wire Length	13	16	15.6	13.5	15.2	12.82	13.6	13.4	12.4	12.2

Table 17. Case Study 2.2 – Resource Utilization with Different Input Activation Bitwidth

Input Bitwidth		1	2	3	4	5	6	7	8	10	13	16
Inputs Per EB	80	40	26	20	16	13	11	10	8	6	5	
Layer 1	Tensor Blocks	611	988	960	988	896	960	987	988	988	986	984
	Lab Usage	928	1492	1421	1523	1460	1521	1733	1569	1772	1772	1747
	M20K Usage	689	1133	1089	1147	1048	1132	1254	1175	1232	1233	1261
Layer 2	Tensor Blocs	903	860	902	860	860	860	924	903	960	946	975
	Lab Usage	3357	2468	2604	2390	2614	2007	2191	1940	2261	2133	1861
	M20K Usage	593	675	899	847	1189	1146	1146	1199	920	1314	940
Layer 3	Tensor Blocks	924	924	924	924	924	924	924	924	924	924	924
	Lab Usage	2303	2474	2557	2622	3068	3118	2879	2968	2023	2131	1881
	M20K Usage	266	381	640	668	1242	1242	1242	1242	762	762	576

Table 18. Case Study 2.2 – Routing Utilization with Different Input Activation Bitwidth

Input Bitwidth		1	2	3	4	5	6	7	8	10	13	16
Inputs Per EB		80	40	26	20	16	13	11	10	8	6	5
Layer 1	Short Wire Util (%)	10.7	20.1	16.3	20.2	14.9	16.7	28.5	20.4	20.8	19.5	17.9
	Long Wire Util (%)	6.1	9.1	8.4	10.3	5.4	8.3	25.8	10.4	10.8	10.1	10.2
	Avg Wire Length	19.2	21.7	18.2	21.9	17.1	18.4	32.2	21.9	22	20.7	19.1
Layer 2	Short Wire Util (%)	19.8	21.0	17.6	16.8	19.0	18.7	18.5	19.5	39.5	25.6	38.1
	Long Wire Util (%)	8.9	8.2	8.1	8.0	8.6	8.4	10.6	8.6	25.5	10.7	22.3
	Avg Wire Length	28.2	27.7	22.0	22.2	24	23.8	22.4	23.7	45.1	28.6	43
Layer 3	Short Wire Util (%)	5.8	7.3	8.8	9.4	10.8	11.6	12.7	13.7	11	12.7	14
	Long Wire Util (%)	4.5	5.5	5.7	6.1	6.4	6.9	7.4	7.6	5.2	5.5	6.5
	Avg Wire Length	5.4	6.5	7.5	7.9	8.9	9.4	10.1	10.6	8.8	9.8	11.6

REFERENCES

[1] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783725

[2] Aman Arora, Samidh Mehta, Vaughn Betz, and Lizy John. 2020. Tensor slices to the rescue: Supercharging ML acceleration on FPGAs. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*.

[3] A. Arora, Z. Wei, and L. K. John. 2020. Hamamu: Specializing FPGAs for ML applications by adding hard matrix multiplier blocks. In *2020 IEEE 31st International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. 53–60. https://doi.org/10.1109/ASAP49362.2020.00018

[4] Arash Azizimazreah and Lizhong Chen. 2019. Shortcut mining: Exploiting cross-layer shortcut reuse in DCNN accelerators. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 94–105. https://doi.org/10.1109/HPCA.2019.00030

[5] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. 1999. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers.

[6] Andrew Boutros, Mohamed Eldafrawy, Sadegh Yazdanshenas, and Vaughn Betz. 2019. Math doesn't have to be hard: Logic block architectures to enhance low-precision multiply-accumulate on FPGAs. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'19)*. Association for Computing Machinery, New York, NY, USA, 94–103. https://doi.org/10.1145/3289602.3293912

[7] Andrew Boutros, Eriko Nurvitadhi, Rui Ma, Sergey Gribok, Zhipeng Zhao, James Hoe, Vaughn Betz, and Martin Langhammer. 2020. Beyond peak performance: Comparing the real performance of AI-optimized FPGAs and GPUs *(FPT'20)*.

[8] A. Boutros, S. Yazdanshenas, and V. Betz. 2018. Embracing diversity: Enhanced DSP blocks for low-precision deep learning on FPGAs. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8. https://doi.org/10.1109/FPL.2018.00014

[9] Intel Corporation. 2020. Avalon Interface Specifications.

[10] J. Das and S. J. E. Wilton. 2011. An analytical model relating FPGA architecture parameters to routability. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 181–184.

[11] Mohamed Eldafrawy, Andrew Boutros, Sadegh Yazdanshenas, and Vaughn Betz. 2020. FPGA logic block architectures for efficient deep learning inference. *ACM Trans. Reconfigurable Technol. Syst.* 13, 3, Article 12 (June 2020), 34 pages. https://doi.org/10.1145/3393668

[12] A. Firuzan, M. Modarressi, M. Daneshtalab, and M. Reshadi. 2018. Reconfigurable network-on-chip for 3D neural network accelerators. In *2018 Twelfth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*. 1–8. https://doi.org/10.1109/NOCS.2018.8512170

[13] Debabrata Ghosh, Nevin Kapur, Franc Brglez, and Justin E. Harlow. 1998. Synthesis of wiring signature-invariant equivalence class circuit mutants and applications to benchmarking. In *1998 Design, Automation and Test in Europe*. 656–663.

[14] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. 2019. SparTen: A sparse tensor accelerator for convolutional neural networks. In *Proceedings of the 52nd Annual IEEE/ACM International*

Symposium on Microarchitecture (MICRO'52). Association for Computing Machinery, New York, NY, USA, 151–165. https://doi.org/10.1145/3352460.3358291

[15] David Grant and Guy Lemieux. 2009. Perturb+mutate: Semi-synthetic circuit generation for incremental placement and routing. *ACM Transactions on Reconfigurable Technology and Systems* 1, 3 (2009), 1–24.

[16] Kartik Hegde, Rohit Agrawal, Yulun Yao, and Christopher W. Fletcher. 2018. Morph: Flexible Acceleration for 3D CNN-based Video Understanding. (2018). arXiv:cs.LG/1810.06807

[17] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. (2017). arXiv:cs.CV/1704.04861

[18] Michael D. Hutton, Jonathan Rose, and Derek G. Corneil. 2002. Automatic generation of synthetic sequential benchmark circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems* 21, 8 (2002), 928–940.

[19] S. Jiang, P. Pan, Y. Ou, and C. Batten. 2020. PyMTL3: A Python framework for open-source hardware modeling, generation, simulation, and verification. *IEEE Micro* 40, 4 (2020), 58–66. https://doi.org/10.1109/MM.2020.2997638

[20] J. H. Kim, J. Lee, and J. H. Anderson. 2018. FPGA architecture enhancements for efficient BNN implementation. In *2018 International Conference on Field-Programmable Technology (FPT)*. 214–221. https://doi.org/10.1109/FPT.2018.00039

[21] Paul D. Kundarewich and Jonathan Rose. 2004. Synthetic circuit generation using clustering and iteration. *IEEE Trans. on CAD of Integrated Circuits and Systems* 23, 6 (2004), 869–887.

[22] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects. *SIGPLAN Not.* 53, 2 (March 2018), 461–475. https://doi.org/10.1145/3296957.3173176

[23] Martin Langhammer, Eriko Nurvitadhi, Bogdan Pasca, and Sergey Gribok. 2020. Stratix 10 NX architecture and applications. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'20)*.

[24] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li. 2017. FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 553–564. https://doi.org/10.1109/HPCA.2017.29

[25] Cindy Mark, Scott Chin, Lesley Shannon, and Steven Wilton. 2012. Hierarchical benchmark circuit generation for FPGA architecture evaluation. *ACM Transactions on Embedded Computing Systems* 11, S2 (2012), 42:1–42.25.

[26] Kevin E. Murray, Oleg Petelin, Sheng Zhong, Jia Min Wang, Mohamed Eldafrawy, Jean-Philippe Legault, Eugene Sha, Aaron G. Graham, Jean Wu, Matthew J. P. Walker, Hanqing Zeng, Panagiotis Patros, Jason Luu, Kenneth B. Kent, and Vaughn Betz. 2020. VTR 8: High-Performance CAD and customizable FPGA architecture modelling. *ACM Trans. Reconfigurable Technol. Syst.* 13, 2, Article 9 (May 2020), 55 pages. https://doi.org/10.1145/3388617

[27] Kevin E. Murray, Scott Whitty, Suya Liu, Jason Luu, and Vaughn Betz. 2015. Timing-Driven titan: Enabling large benchmarks and exploring the gap between academic and commercial CAD. *ACM Trans. Reconfigurable Technol. Syst.* 8, 2, Article 10 (March 2015), 18 pages. https://doi.org/10.1145/2629579

[28] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. (2017). arXiv:cs.NE/1708.04485

[29] J. Pistorius, E. Legai, and M. Minoux. 2000. PartGen: A generator of very large circuits to benchmark the partitioning of FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 19, 11 (2000), 1314–1321.

[30] Seyedramin Rasoulinezhad, Siddhartha, Hao Zhou, Lingli Wang, David Boland, and Philip H. W. Leong. 2020. LUXOR: An FPGA logic cell architecture for efficient compressor tree implementations. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'20)*. Association for Computing Machinery, New York, NY, USA, 161–171. https://doi.org/10.1145/3373087.3375303

[31] S. Rasoulinezhad, H. Zhou, L. Wang, and P. H. W. Leong. 2019. PIR-DSP: An FPGA DSP block architecture for multi-precision deep neural networks. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 35–44. https://doi.org/10.1109/FCCM.2019.00015

[32] A. Samajdar, T. Garg, T. Krishna, and N. Kapre. 2019. Scaling the cascades: Interconnect-Aware FPGA implementation of machine learning problems. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 342–349. https://doi.org/10.1109/FPL.2019.00061

[33] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783720

[34] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783720

[35] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Maximizing CNN accelerator efficiency through resource partitioning. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. Association for Computing Machinery, New York, NY, USA, 535–547. https://doi.org/10.1145/3079856.3080221

[36] Atefeh Sohrabizadeh, Jie Wang, and Jason Cong. 2020. End-to-end optimization of deep learning applications. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'20)*. Association for Computing Machinery, New York, NY, USA, 133—139. https://doi.org/10.1145/3373087.3375321

[37] D. Stroobandt, J. Depreitre, and J. VanCampenhout. 1999. Generating new benchmark designs using a multi-terminal net model. *Integration: The VLSI Journal* 27, 2 (1999), 113–129.

[38] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. 2016. Throughput-Optimized OpenCL-Based FPGA accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'16)*. Association for Computing Machinery, New York, NY, USA, 16–25. https://doi.org/10.1145/2847263.2847276

[39] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. (2017). arXiv:cs.CV/1703.09039

[40] Martin Tom and Guy Lemieux. 2005. Logic block clustering of large designs for channel-width constrained FPGAs. In *Design Automation Conference*.

[41] S. I. Venieris and C. Bouganis. 2016. fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 40–47. https://doi.org/10.1109/FCCM.2016.22

[42] P. Verplaetse, D. Stroobandt, and J. VanCampenhout. 2002. Synthetic benchmark circuits for timing-driven physical design applications. In *International Conference on VLSI*. 31–37.

[43] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li. 2016. DeepBurning: Automatic generation of FPGA-based learning accelerators for the Neural Network family. In *2016 53rd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2897937.2898002

[44] Andy Yan, Rebecca Cheng, and Steven J. E. Wilton. 2002. On the sensitivity of FPGA architectural conclusions to experimental assumptions, tools, and techniques. In *International Symposium on FPGAs*. 147–156.

[45] S. Yang. 1991. Logic Synthesis and Optimization Benchmarks User Guide 3.0. Technical Report. In *MCNC*. 1–6.

[46] Xuan Yang, Jing Pu, Blaine Burton Rister, Nikhil Bhagdikar, Stephen Richardson, Shahar Kvatinsky, Jonathan Ragan-Kelley, Ardavan Pedram, and Mark Horowitz. 2016. A systematic approach to blocking convolutional neural networks. *CoRR* abs/1606.04209 (2016). arXiv:1606.04209 http://arxiv.org/abs/1606.04209

[47] Yufei Ma, N. Suda, Yu Cao, J. Seo, and S. Vrudhula. 2016. Scalable and modularized RTL compilation of convolutional neural networks onto FPGA. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8. https://doi.org/10.1109/FPL.2016.7577356

[48] Hanqing Zeng, Ren Chen, Chi Zhang, and Viktor Prasanna. 2018. A framework for generating high throughput CNN implementations on FPGAs. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'18)*. Association for Computing Machinery, New York, NY, USA, 117–126. https://doi.org/10.1145/3174243.3174265

[49] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-Based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'15)*. Association for Computing Machinery, New York, NY, USA, 161–170. https://doi.org/10.1145/2684746.2689060

[50] C. Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2016. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1145/2966986.2967011

[51] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W. Hwu, and D. Chen. 2018. DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1145/3240765.3240801

[52] Xiaofan Zhang, Hanchen Ye, Junsong Wang, Yonghua Lin, Jinjun Xiong, Wen Mei Hwu, and Deming Chen. 2021. DNNExplorer: A Framework for Modeling and Exploring a Novel Paradigm of FPGA-based DNN Accelerator. (2021). arXiv:cs.AR/2008.12745

Received June 2021; revised October 2021; accepted November 2021