Mitigating Memory Wall Effects in CNN Engines with On-the-Fly Weights Generation

STYLIANOS I. VENIERIS*, Samsung AI Center, Cambridge, UK
JAVIER FERNANDEZ-MARQUES*, Samsung AI Center, Cambridge, UK
NICHOLAS D. LANE, Samsung AI Center, Cambridge & University of Cambridge, UK

The unprecedented accuracy of convolutional neural networks (CNNs) across a broad range of AI tasks has led to their widespread deployment in mobile and embedded settings. In a pursuit for high-performance and energy-efficient inference, significant research effort has been invested in the design of FPGA-based CNN accelerators. In this context, single computation engines constitute a popular design approach that enables the deployment of diverse models without the overhead of fabric reconfiguration. Nevertheless, this flexibility often comes with significantly degraded performance on memory-bound layers and resource underutilisation due to the suboptimal mapping of certain layers on the engine's fixed configuration. In this work, we investigate the implications in terms of CNN engine design for a class of models that introduce a pre-convolution stage to decompress the weights at run time. We refer to these approaches as *on-the-fly*. This paper presents unzipFPGA, a novel CNN inference system that counteracts the limitations of existing CNN engines. The proposed framework comprises a novel CNN hardware architecture that introduces a weights generator module that enables the on-chip on-the-fly generation of weights, alleviating the negative impact of limited bandwidth on memory-bound layers. We further enhance unzipFPGA with an automated hardware-aware methodology that tailors the weights generation mechanism to the target CNN-device pair, leading to an improved accuracy-performance balance. Finally, we introduce an input selective processing element (PE) design that balances the load between PEs in suboptimally mapped layers. Quantitative evaluation shows that the proposed framework yields hardware designs that achieve an average of 2.57× performance efficiency gain over highly optimised GPU designs for the same power constraints and up to 3.94× higher performance density over a diverse range of state-of-the-art FPGA-based CNN accelerators.

CCS Concepts: • **Computer systems organization → Reconfigurable computing**; • **Computing methodologies → Neural networks**.

Additional Key Words and Phrases: neural networks, hardware accelerator, weights generation

ACM Reference Format:
Stylianos I. Venieris, Javier Fernandez-Marques, and Nicholas D. Lane. 2022. Mitigating Memory Wall Effects in CNN Engines with On-the-Fly Weights Generation. *ACM Trans. Des. Autom. Electron. Syst.* 37, 4, Article 111 (August 2022), 31 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The unparalleled accuracy of convolutional neural networks (CNNs) across a broad range of AI inference tasks has led to the development of novel applications. With a large portion deployed

*Both authors contributed equally to this research.

Authors' addresses: Stylianos I. Venieris, s.venieris@samsung.com, Samsung AI Center, Cambridge, UK; Javier Fernandez-Marques, j1.fernandez@samsung.com, Samsung AI Center, Cambridge, UK; Nicholas D. Lane, Samsung AI Center, Cambridge & University of Cambridge, UK, nic.lane@samsung.com.

1084-4309/2022/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

across mobile and embedded devices, there is a need for high-performance, energy-efficient implementations that can deliver responsiveness and prolonged battery life. Currently, the conventional computing platforms for CNN inference comprise either CPUs and GPUs [8, 89, 94] or custom application-specific integrated circuits (ASICs), such as neural processing units (NPUs) [8, 43, 45]. On the one hand, CPU- and GPU-based systems can support diverse CNN models through their programmability, but penalise performance in order to provide this generality [8]. On the other hand, ASICs provide significant acceleration under a minimal power envelope [43]. Nevertheless, the benefits of ASICs typically require the functionality to remain fixed after fabrication, leaving no room for applying model-specific optimisations [63, 77] or mapping newer CNN models [85].

To provide a balance between flexibility and high performance, numerous CNN accelerators target reconfigurable hardware platforms, such as field-programmable gate arrays (FPGAs). Currently, the FPGA-based accelerator landscape spans a wide spectrum, from flexible CNN-specific processors [31, 100, 101] to highly customised streaming architectures [41, 83, 85, 90]. One of the most widely adopted paradigms that lies in the midpoint of the flexibility-customisation spectrum is the single computation engine (SCE) [1, 2, 33, 47, 51, 52, 62, 67, 96, 103]. Under this paradigm, a powerful processing engine is time-shared to sequentially execute the layers of a CNN. This allows for accelerator's resources to be reused across both layers and CNN models, without the need to reconfigure the fabric.

Despite the flexibility of SCEs, their attainable performance is often bounded by two primary factors: *1)* layers with low computation-to-communication ratio that become memory-bound [51, 62, 67] and *2)* the suboptimal mapping of diverse layers on the fixed configuration of the SCE that leads to underutilised processing elements (PEs) [62, 84, 102]. These two obstacles set a hard limit to the actual sustained performance that this family of accelerators can achieve, indicating an emerging need for novel solutions to counteract their impact.

Concurrently with the continuing hardware advances, a growing body of work focuses on compressing CNNs through a class of lossy non-structural methods [26, 27, 35, 71, 80, 98]. Orthogonally to other model simplification techniques such as pruning or quantisation, this group of methods dictates that the weights of a model are deployed in a compact form and are "inflated" only at run time. Given that several CNN layers are constrained by the limited off-chip memory bandwidth of the target computing platforms [51, 62, 67, 84], storing the compressed weights on-chip and reconstructing them on-the-fly can play a decisive role in alleviating the memory-boundedness and enabling the better utilisation of the computational resources.

Nevertheless, the novel dataflow and execution scheme of such models brings up a new challenge regarding their optimised mapping. Existing accelerators have been designed for conventional deep models, adopting either a streaming or layer-by-layer execution [88]. Hence, despite the significant potential of on-the-fly models, their different execution paradigm renders conventional architectures futile in serving them.

This paper presents a novel CNN system that overcomes the withstanding limitations of single computation engines and enables the efficient and high-performance execution of on-the-fly models. At the core of the proposed system lies a novel CNN hardware architecture. To alleviate the impact of the memory wall, we introduce a hardware-based weights generator that is responsible for efficiently generating the CNN weights on-the-fly. Comprising a custom memory organisation and a highly optimised datapath, the weights generator is scalable, with tunable parameters that allow it to be tailored to the needs of the target application, the workload characteristics of the CNN and, the capabilities of the FPGA device. To counteract the underutilisation of computational resources due to the suboptimal mapping of diverse layers, we further propose a novel CNN engine design comprising input selective PEs. Under this design, a subset of PEs is enhanced with efficient switches that enable neighbouring PEs to perform load-balancing through seamless work-stealing.

Finally, we present a framework for deriving on-the-fly models from pre-trained CNNs and mapping them on a given FPGA.

The initial work in [87] provided a high-level overview of the proposed framework and its on-the-fly weights formulation. Moreover, the evaluation solely focused on end-to-end performance compared to other FPGA works. In this paper, we first present the complete framework, encompassing both the algorithmic and hardware aspects (Section 3). As such, we provide a detailed description of the on-the-fly weights generation process and its algorithmic underpinning on OVSF codes (Section 2.2), and we position the proposed hardware architecture with respect to the status-quo CNN engines (Section 4).

Moreover, in the initial work, the compression ratios of each CNN layer were manually selected based on heuristics. In this work, we make steps towards automation by introducing a hardware-aware scheme for tuning the per-layer compression ratios of OVSF models (Section 6.2). The proposed method exploits the bottleneck characteristics of each layer in order to generate more accurately its weights while sustaining high throughput, leading to a better accuracy-performance trade-off (Section 7.5).

Finally, we present for the first time an in-depth evaluation of various critical aspects of the proposed framework. These include the evaluation of two techniques to derive on-the-fly models from vanilla CNNs (Section 7.1.2), a thorough study of the impact of the proposed input selective PEs (Section 7.4), an extensive comparison with highly optimised designs on embedded GPUs (Section 7.6) and further comparisons with the most recent state-of-the-art FPGA-based accelerators (Section 7.2.2).

2 BACKGROUND AND RELATED WORK

This section first positions the proposed on-the-fly formulation in context with existing optimisation techniques for CNNs. We continue with a brief historical context for OVSF codes and how to construct them. Then, we show how orthogonal variable spreading factor (OVSF) codes are used to construct filters in CNNs and, how they are integrated into the training process. This section concludes with an overview of the challenges and opportunities of CNN-targeting single computation engines, which is our paradigm of choice when implementing on-the-fly models on FPGAs.

2.1 Designing Lightweight Convolutional Neural Networks

The plethora of existing techniques to modify CNNs for faster inference can be categorised into: pruning [15, 39, 61], which removes redundant parameters; quantisation [7, 24, 28, 44, 53], which results in compact low-precision models; or, sparsification [14, 30, 93], which leverages compressed data formats. In addition, a number of frameworks combine several of these techniques. Most notably: Deep Compression [37] which, given an over-parametrised model, applies pruning, quantisation and Huffman encoding; RedCNN [92] which prunes channels based on an activation overlap metric; and, more recently, APQ [91], which designs a CNN that meets given computational, memory and latency constraints through a joint optimisation formulation.

2.2 On-the-Fly Convolutional Neural Networks

Orthogonal to these methods, various works have explored ways of obtaining extremely compact model representations by factorising the filters in CNNs or by passing these through a multi-stage compression pipeline. Common to these methods is the need for a *decompressing* stage that generates the filters at run time during inference. A selection of such techniques include: [35] that uses an auxiliary NN to generate each layer's weights in the main network given an embedding of the weights. In [71], weight filters are constructed as a dense combination of a set of Fourier Bessels

bases that are generated deterministically at run time. Another technique exploiting deterministic bases was presented in [26, 27, 80], where bases are formed from OVSF binary codes. It enables the construction of model weights by learning a linear combination of OVSF bases during training.

We label these techniques as *on-the-fly* since they *1)* require a single-step decompression stage to obtain the filters, *2)* this process can be done on-demand, *i.e.* at each layer and for every input, and *3)* such decompression is lightweight, *i.e.* it does not require multiple inferences to amortise its associated costs. In this work, we focus on on-the-fly methods that use compression as a means to reduce data-movement overheads, balancing the costs of decompressing the model parameters with the latency savings due to reduced off-chip or main memory accesses.

To this end, this work makes use of OVSF codes to compress filters in a CNN and reconstruct them during inference in a lightweight manner. Three additional reasons motivated the choice of this class of codes: *1)* OVSF codes are *binary* and thus can be efficiently stored on-chip [48]; *2)* their theoretical properties are well studied by the wireless community [10, 72]; *3)* they offer good compression-accuracy trade-off (i.e. lossy compression) in various AI tasks [26, 80]. Nonetheless, [70] is the only existing FPGA-based OVSF design, presenting solely a direct implementation specific for communication systems. To effectively use OVSF with CNNs, the underlying hardware design needs to be tailored and optimised for the CNN dataflow.

2.3 On-the-Fly OVSF CNN Layers

The chosen OVSF codes are a set of mutually orthogonal binary codes, originally designed to split in the frequency domain signals from different users in W-CDMA-based 3G cellular systems [3]. Using them as channelisation codes allowed for communication channels to remain orthogonal in multi-user access scenarios, reducing signal interference while dramatically increasing the system capacity.

These codes can be obtained using Sylvester's construction algorithm for Hadamard matrices. In this way, given $H_0 = [1]$ and H_2, subsequent H_{2^k} expansions are defined as

$$H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad H_{2^k} = \begin{bmatrix} H_{2^{k-1}} & H_{2^{k-1}} \\ H_{2^{k-1}} & -H_{2^{k-1}} \end{bmatrix} = H_2 \otimes H_{2^{k-1}} \tag{1}$$

where H_{2^k} is an $L \times L$ Hadamard matrix, with $L = 2^k, k \in \mathbb{N}$ and $\otimes$ is the Kronecker product. For $k > 1$, each row is an OVSF code fulfilling the properties of being binary and orthogonal to each other. This will enable us to use them as basis for $\mathbb{R}^L$. An alternative formulation [4] allows for the construction of OVSF codes as a recursive expansion of a perfect binary tree.

By treating the set of L OVSF codes as a basis spanning $\mathbb{R}^L$, we can define the construction process of an arbitrary real-valued vector $\boldsymbol{v}'_i$ as the linear combination of such codes

$$\boldsymbol{v}'_i = \sum_{j=0}^{\lfloor \rho \cdot L \rceil} \alpha_i^j \boldsymbol{b}_i^j, \quad E_i = \left\| \boldsymbol{v}'_i - \boldsymbol{v}_i \right\|_2^2 = \left\| \sum_{j=0}^{\lfloor \rho \cdot L \rceil} \alpha_i^j \boldsymbol{b}_i^j - \boldsymbol{v}_i \right\|_2^2 < \epsilon \tag{2}$$

where $\boldsymbol{\alpha}_i = \{\alpha_i^0, \alpha_i^1, \alpha_i^2, ..., \alpha_i^{L-1}\}$ are weighting coefficients, $\boldsymbol{b}_i^j$ is the j-th OVSF binary code of length L and, $\rho \in [0, 1]$ is the ratio of codes to use in order to construct $\boldsymbol{v}_i$. The expression on the right measures the difference between $\boldsymbol{v}'_i$ and a real-valued standard vector of length L, $\boldsymbol{v}_i$. Intuitively, $\epsilon \to 0$ as we increase the ratio of binary codes used.

When constructing matrices from OVSF codes or higher-dimensional tensors, a reshaping stage follows the linear combination shown in Eq. (2). In this way, if the weights tensor of a given convolutional layer is of shape $N_{\text{out}} \times N_{\text{in}} \times K \times K$, the construction process using OVSF could be framed as the concatenation of N_{out} $N_{\text{in}} \times K \times K$ filters using codes of length $L = N_{\text{in}}KK$ and up to L of such codes. Here, N_{in} and N_{out} stands for the number of input and output channels in the

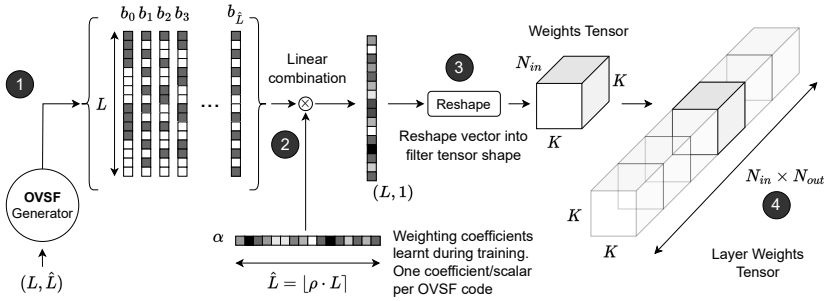


Fig. 1. Constructing filters of a convolutional layer using OVSF codes. With an OVSF code generator outputing binary codes of length L (1), a $K{\times}K$ filter with N_{in} channels (*i.e.* of shape $N_{\text{in}}{\times}K{\times}K$) is obtained by performing a linear combination (2) of $\hat{L} = \lfloor \rho \cdot L \rceil$ codes of length $L = N_{\text{in}}KK$, with $\rho \in [0, 1]$. Then, the resulting $L{\times}1$ vector is reshaped to match the target filter's shape (3). For a convolutional layer with N_{out} output channels, this process is repeated N_{out} times, concatenating the results (4).

layer respectively, and $K{\times}K$ are the spatial dimensions of the convolutional filter. This scenario is illustrated in Fig. 1. The set of scalars $\{\alpha\}_{k=1}^{L}$ in each layer are learnt via standard backpropagation and represent the only learnable parameters in an OVSF layer since the OVSF binary codes themselves are fixed. A compressed representation of f_i is obtained when $\rho{<}1$. Upon deployment, the filters are first generated and then the main inference computation proceeds as normal.

Due to the construction process of OVSF codes (Eq. 1), K in OVSF convolutional layers is limited to be a power-of-two number. We address this in Sec. 6.1, where we present two methods to enable the construction of filters with $K \in \mathcal{N}$, for example with the ubiquitous $K = 3$.

2.4 Challenges of FPGA-based CNN Engines

Until now, a wide array of FPGA-based CNN accelerators have been proposed. In the customisation-programmability spectrum, existing designs span from custom streaming architectures [41, 85] and accelerators for quantised [16, 69, 82, 83, 90, 105] and sparse CNNs [32, 40, 59, 68, 79, 104, 106], up to instruction-based processors [31, 100, 101]. One of the most well adopted paradigms are the single computation engines [1, 2, 33, 51, 52, 62, 67, 96, 103], due to their balanced trade-off of programmability and performance. Currently, despite the progress in processing unit design, further gain in the attainable performance of such engines is hindered by two main factors: i) memory-bound layers that are dominated by the communication with the external memory [51, 62, 67, 78]. While embedded platforms provide limited bandwidth [21, 64, 86], *e.g.* less than 4.5 GB/s for Ultra96 and ZC706, sustaining peak bandwidth even on larger devices, such as ZCU104, is nontrivial [64]. This is aggravated as multiple applications are collocated on a single device [13, 86, 97]; and ii) underutilised PEs due to the mismatch of diverse layer shapes [1, 47, 54, 62, 84, 102].

Memory-centric Designs. The memory bandwidth problem faced by CNN engines has been studied in previous work. EIE [36] uses the Deep Compression method [37] to compress the weights of fully connected (FC) layers. However, as FC layers have been mostly abandoned in modern CNNs, its applicability is limited. Angel-Eye [34] compresses all layers through precision quantisation. Cambricon-X [104] transfers only the non-zero weights, while Cambricon-S [106] and Scalpel [99] apply coarse weight pruning, but with significant accuracy drop. CircCNN [23] uses block-circulant matrices for weights, but requires complex FFT hardware for efficient execution. [22] converts sparse weights to permuted diagonal matrices, but only targets FC layers. [74] exploits large batch

sizes to increase weights reuse and, thus, is not suitable for latency-critical applications that cannot tolerate batching [84].

Focusing on activations, [9] fuses adjacent layers to cache intermediate activations, while [68] and Eyeriss [20], [66] employ encoding schemes to minimise their bandwidth footprint. Other solutions have either relied on large devices [11] and multiple FPGAs [29] to fit all weights on-chip, or utilised highly customised designs to exploit multi-precision cascades [52] or fine-grained pruning [59] at the cost of notable accuracy drop.

Tackling PE Underutilisation. So far, a limited number of designs have focused on ii). [75] addresses underutilisation by grouping CONV layers based on the compatibility of their shapes. [41] maps each layer to a dedicated compute stage, which can be used only for shallower networks, but does not scale to the deeper models of today. Furthermore, a limited number of works rely on FFT-based designs with flexible dataflow [67] and costly ASIC solutions [12, 55, 60, 81] with highly flexible PE interconnect.

In contrast to these works, we propose an approach that is independent of the CNN engine by not requiring any modification to the engine architecture itself. unzipFPGA can benefit any existing single computation engine by augmenting it with its hardware weights generator and enhancing its PE array with lightweight switches, without affecting the PE's internal processing units. As such, unzipFPGA is orthogonal and complementary to quantisation [34], activations' encoding [20, 68], fusion [9] and zero-skipping PEs [5, 6, 32, 56, 106].

3 unzipFPGA'S DESIGN FLOW

Our framework aims to enhance the performance of hardware CNN engines, while maintaining a high level of abstraction for deep learning developers. Fig. 2 shows a high-level view of unzipFPGA's design flow, comprising two software components: *1)* the *OVSF Model Converter* and *2)* the *Optimiser*.

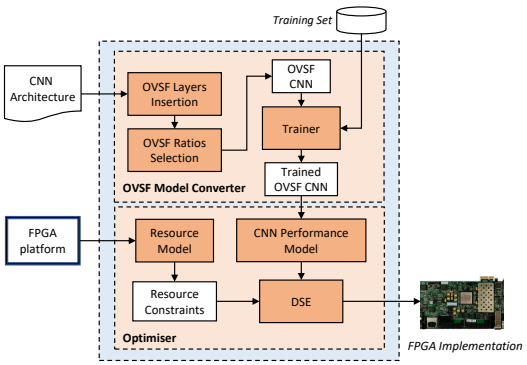


Fig. 2. Overview of unzipFPGA's design flow.

As a starting point, the deep learning expert provides the CNN model, expressed in PyTorch, and the target FPGA platform. The *Converter* processes the supplied CNN architecture and derives an OVSF variant, by transforming the conventional convolutional layers into OVSF convolutional (OVSF-CONV) layers. This step entails *i)* the replacement of weight filters with a trainable linear combination of OVSF bases, followed by *ii)* the selection of each layer's compression ratio ρ. Next, the OVSF model is passed to the *Trainer*, where the model gets trained using the supplied training set.

The *Optimiser* accepts the trained OVSF CNN and a given FPGA platform and, uses them to populate the CNN *Performance Model* and the *Resource Constraints*, respectively. Importantly, the *Optimiser* navigates the hardware configuration space considering resource allocations between the CNN engine and the weights generator. Upon completion, the design space exploration (*DSE*) stage yields the highest performing configuration of unzipFPGA's architecture for the given CNN-device pair and the system is deployed on the FPGA.

4 CNN ENGINE DESIGN FOR ON-THE-FLY WEIGHTS

This section starts by reviewing the hardware architecture of a conventional CNN engine, similar to the ones in [20, 25, 51]. Then, it provides a series of design requirements to enable on-the-fly weights generation for CNNs and presents our techniques for achieving them.

4.1 Conventional CNN Engine Design

Fig. 3 illustrates a typical CNN engine design. The accelerator consists of an array of processing elements (PEs) to perform matrix multiplications and convolutions, one input and one output activations buffer, and a weights buffer. From an operational perspective, the CNN layers are scheduled sequentially, with pipelining applied between I/O communication and computation to hide the off-chip memory transfer latency.

Processing Engine: To execute layers of various shapes and types, the core processing engine comprises an array of PEs for the execution of block matrix multiply (GEMM). Each PE contains a scalable dot-product circuit with configurable number of multiply-accumulate (MAC) units. By translating convolutions into matrix multiplication, the engine can process both CONV and FC layers. To this end, a CONV layer with N_{in} $H{\times}W$ input activations, N_{out} output channels, $K{\times}K$ filters, p padding and S stride involves the multiplication between an $R{\times}P$ activations matrix and a $P{\times}C$ weights matrix to produce an $R{\times}C$ output matrix, with $R{=}\left\lceil\frac{H+2p-K}{S}+1\right\rceil\left\lceil\frac{W+2p-K}{S}+1\right\rceil$, $P{=}N_{\text{in}}K^2$ and $C{=}N_{\text{out}}$.

Design-time Parametrisation: The CNN engine can be scaled based on the workload characteristics and the resources of the target FPGA. As such, it is parametrised with respect to the parameter tuple $\langle T_R, T_P, T_C\rangle$. Each parameter determines the tile sizes for each matrix dimension $\langle R, P, C\rangle$, the number of PEs (T_C) and the MAC units within each PE (T_P).

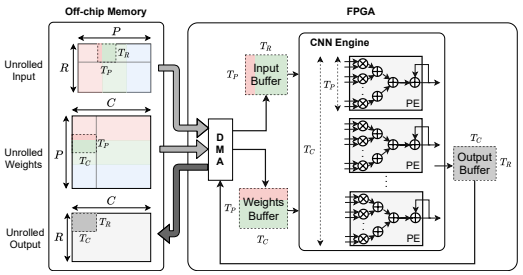


Fig. 3. A conventional CNN engine.

Operational Flow: To produce a $T_R{\times}T_C$ output tile, $\left\lceil\frac{P}{T_P}\right\rceil$ tiles from the activations and weights matrices are processed and accumulated sequentially. A common mapping strategy (Fig. 3) ties T_P to the MACs per PE to exploit the parallelism within each T_P-wide dot product, and T_C to PEs to parallelise the dot products at each output column. Overall, the rows of the $T_R{\times}T_P$ activations tile are processed in a pipelined manner to maximise throughput. This is equivalent to an output stationary dataflow [20, 25, 51], which minimises the memory accesses for the output activations by caching partial sums on-chip. Nonetheless, unzipFPGA is adaptable to other dataflows with minimal modifications.

The Data Movement Bottleneck: From a data movement perspective, this approach requires the transfer of $\left\lceil\frac{P}{T_P}\right\rceil$ tiles of size $T_R{\times}T_P$ for the inputs, $\left\lceil\frac{P}{T_P}\right\rceil$ tiles of size $T_P{\times}T_C$ for the weights, and one tile of size $T_R{\times}T_C$ for the outputs. To produce all the output tiles, all the data movements are performed $\left\lceil\frac{R}{T_R}\right\rceil\left\lceil\frac{C}{T_C}\right\rceil$ times. In spite of the compute-bound CONV layers, the external memory bandwidth often becomes the bottleneck in CNN inference. This is primarily manifested in cases where: *i)* a resource-rich FPGA device is targeted. In this case, a large and powerful processing engine is instantiated and the speed of feeding it with new data constrains the performance; *ii)* the CNN layers have a large amount of weights, either due to large kernel sizes or number of filters. This case often occurs in deeper CNN layers, which are typically of significant width. As such, the

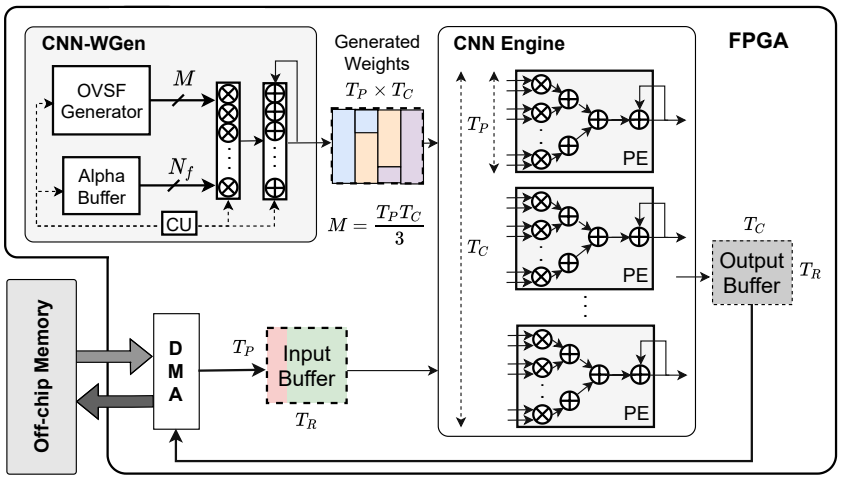


Fig. 4. CNN engine with on-the-fly weights generation. The weights are dynamically generated and the full bandwidth is available for inputs and outputs.

weights cannot be stored on-chip and multiple memory transactions have to be issued, putting pressure on the available bandwidth; *iii)* high-dimensional input and output activations have to be transferred, increasing excessively the bandwidth requirements. This typically occurs in earlier layers of a CNN, where the feature maps dimensions are still large.

As such, there is an emerging need for relieving the data movement burden and address the impact of hitting the memory wall. In this context, *on-the-fly weights generation* can be an enabling factor in extracting higher performance and making more cost-effective use of hardware CNN engines.

4.2 Devising a Hardware CNN Weights Generator

The objective is to minimise the data movement of CNNs by dynamically constructing the model weights using only on-chip resources. Importantly, on-the-fly weights generation needs to take place at run time, in a timely and per-layer fashion, since each CNN layer is a standalone independent schedulable unit. Moreover, the amount of computational and memory resources assigned to the weights generator ought to be balanced with the rest of the CNN engine to maximise the throughput of the accelerator while sustaining high resource utilisation. To that end, bringing forth on-the-fly weights generation requires devising two major components:

4.2.1 Tiled Weights Generation. Our novel insight is that, to be able to generate weights for layers of various dimensions, there is a need for a tiling method on top of the weights generation process. We denote our proposed tiling method by TiWGen. As shown in Fig. 4, TiWGen divides each $T_P \times T_C$ weights tile into subtiles of size M, with M being uniform across the CNN's layers. Tiling on top of the weights generation method makes the dataflow of diverse layers identical to each other. With this approach, the value of M becomes independent of the CNN architecture and is solely bound by the resources allocated to the weights generator. As such, M exposes a tunable trade-off between weights generation speed and resource consumption.

Alg. 1 describes the internal workings of TiWGen. Initially, the $P \times C$ weights matrix is partitioned into $\left\lceil \frac{P}{T_P} \right\rceil$ tiles of size $T_P \times T_C$, with each tile processed sequentially (line 1). Next, each tile is divided

Algorithm 1: Generation of a layer's weights using TiWGen

Input: Layer's weights matrix shape: $P \times C = N_{\text{in}}K^2 \times N_{\text{out}}$
Row and column tile sizes T_P and T_C
α values with $\alpha \in \mathbb{R}^{N_{\text{in}}N_{\text{out}}\lceil \rho K^2 \rceil}$
Output: Weights matrix $\boldsymbol{W}$

1 **for** $t \leftarrow 1$ **to** $\left\lceil \frac{P}{T_P} \right\rceil \cdot \left\lceil \frac{C}{T_C} \right\rceil$ **do** ▷ *tiles loop* - # **PIPELINE**
2 **for** $i \leftarrow 1$ **to** $\left\lceil \frac{T_P T_C}{M} \right\rceil$ **do** ▷ *subtiles loop* - # **PIPELINE**
3 $\text{subtile}_i^t \leftarrow \mathbf{0}$
4 **for** $j \leftarrow 1$ **to** ρK^2 **do** ▷ *basis vectors loop* - # **PIPELINE**
5 **for** $k \leftarrow 1$ **to** M **do** ▷ # **UNROLL**
6 $incr_k \leftarrow \text{vec}_j(k) \cdot \alpha_k$ ▷ *Multiplier array*
7 $\text{subtile}_i^t(k) \leftarrow \text{subtile}_i^t(k) + incr_k$ ▷ *Adder array*
8 **end**
9 **end**
10 $\text{tile}^t \leftarrow \text{UpdateTile}(\text{tile}^t, \text{subtile}_i^t)$
11 **end**
12 $\boldsymbol{W} \leftarrow \text{UpdateMatrix}(\boldsymbol{W}, \text{tile}^t)$
13 **end**

into $\left\lceil \frac{T_P T_C}{M} \right\rceil$ subtiles (line 2). After all basis vectors of the current subtile have been processed (lines 4-9), the associated part of the output tile is updated (line 10) and the algorithm proceeds to the next subtile. When all subtiles of a tile have been generated, the weights matrix is updated (line 12) and the algorithm continues to the next iteration until all weights tiles have been constructed.

Applicability to Other Dataflows. Although the presented instance of TiWGen focuses on output stationary dataflows, our method can be applied to hardware designs that employ other dataflows. The main modifications comprise *i)* the order the generated weights and *ii)* the required generation rate. For instance, considering Google's TPU [46] which is the most widely used systolic array for CNN inference, the accelerator adopts a weight-stationary dataflow. In this case, as the tile of the weight matrix is reused for several cycles, the OVSF generator would have to generate weights in longer periods compared to output stationary dataflow and the resource allocation would be automatically adjusted at the DSE stage accordingly.

4.2.2 Weights Generator Microarchitecture. With the design objectives and constraints of Section 3 in mind, we propose a microarchitectural unit, called `CNN-WGen`, which is placed within the CNN engine (Fig. 4) and is responsible for generating the weights in an orderly manner and feeding them to the processing engine. Fig. 5 illustrates the design. As shown, the unit consists of: *i)* a *vector compute datapath* comprising two vector units (multiplier and adder arrays), *ii)* the *Alpha buffer* storing the α values, and *iii)* the *OVSF generator* that is responsible for outputting the M-sized basis vector subtiles as dictated by the TiWGen scheme.

Mapping Strategy. To efficiently map and perform the TiWGen loops, `CNN-WGen` employs loop optimisation techniques, annotated in Alg. 1. Namely, loop pipelining and unrolling are employed to customise the computation patterns and on-chip memory reuse of the weights generator. Pipelining is applied on the three outer loops over tiles (line 1), subtiles (line 2) and basis vectors (line 4), and unrolling on the inner loop that processes the M-sized subtile (line 5). To unroll the innermost loop, `CNN-WGen` employs two M-wide vector units that perform M-parallel multiplications and additions, respectively. In this manner, tuning M can balance the parallelism-resource usage trade-off of the module.

Parametrised Vector Compute Datapath. As shown in Fig. 5, the vector arithmetic units must have a fixed size that complies with the resource constraints of the target FPGA and namely the

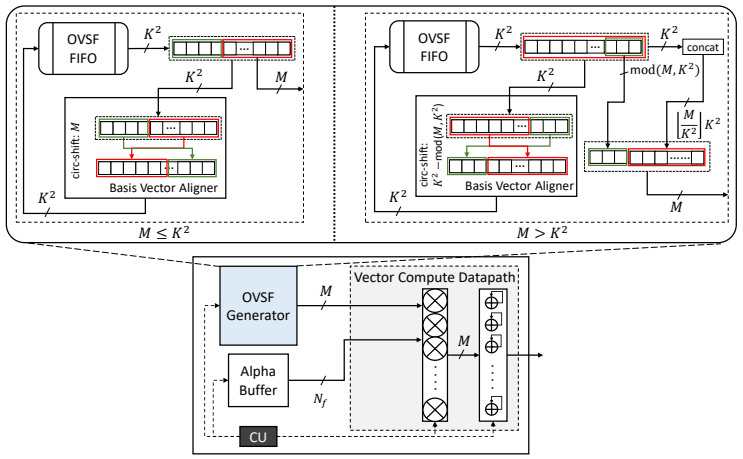


Fig. 5. Microarchitecture of the CNN-WGen module.

available DSP blocks. The multiplier array is connected to the *OVSF generator* and the *Alpha buffer*. For the i-th subtile (line 3 in Alg. 1), the former produces ρK^2 basis vectors of size M, while the latter outputs the associated α coefficients, both of which are forwarded to the multiplier array in a pipelined manner. All M elements are processed in parallel by the M-wide vector units, leading to the vectorised unrolling of the inner loop on line 5 of Alg. 1. The adder array processes the output of the multiplier array by accumulating the ρK^2 partial results. Finally, when TiWGen proceeds to the next subtile (*i.e.* next iteration of the loop on line 2), the control unit (CU in Fig. 5) resets the accumulators' state. Overall, the vector compute datapath is design-time configurable with respect to parameter M which controls the sizing of the vectors units and balances in this way the performance-resource usage trade-off of CNN-WGen. The design space exploration of M is discussed in Section 5.

Memory Customisation in Alpha Buffer. TiWGen dictates that each subtile contains weights from N_f distinct $K \times K$ filters. To sustain CNN-WGen's throughput, an equal number of αs have to be fetched in parallel from the *Alpha buffer*. To accomplish this, we design the *Alpha buffer* as a unified buffer with customised memory organisation and addressing. Each layer contains $N_{\text{in}} N_{\text{out}} \left\lceil \rho_l K_l^2 \right\rceil$ distinct α values. As such, the *Alpha buffer* is broken down to $N_P^{\text{Alpha}} = N_f$ independent multi-bank sub-buffers, with a depth of D^{Alpha} (Eq. (4)) to accommodate N_L layers.

$$N_f = \left\lceil \frac{\min(T_P, M)}{K_{\max}^2} \right\rceil \left\lfloor \frac{M}{T_P} \right\rfloor + \text{mod}(M, T_P) \left\lceil \frac{M}{K_{\max}^2} \right\rceil \tag{3}$$

$$D^{\text{Alpha}} = \overbrace{\sum_{l=1}^{N_L}}^{\text{for each layer}} \overbrace{\frac{N_{\text{in}}^l N_{\text{out}}^l \left\lceil \rho_l K_l^2 \right\rceil}{N_P^{\text{Alpha}}}}^{\text{no. of } \alpha \text{ values}} \quad (\textit{Buffer depth}) \tag{4}$$

where N_L is the number of layers, $N_{\{\text{in,out}\}}^l$ the l-th layer's number of input/output channels and ρ_l the compression ratio. Finally, the outputs of the sub-buffers are concatenated and connected to the multiplier array to provide concurrent access to N_f coefficients. Finally, if the number of α coefficients exceeds the available on-chip memory, the remaining coefficients are transferred from the off-chip memory.

Rate Matching in OVSF Generator. Following TiWGen, the basis vectors are processed in a blocked manner with a tile size of M. This approach leads to two pipelined loops over the $\left\lceil \frac{T_P T_C}{M} \right\rceil$ subtiles (line 2) and the ρK^2 basis vectors (line 4) and the unrolled loop of processing the M-element

subtile with the vector units (line 5). To produce the i-th subtile, the *OVSF generator* feeds the compute datapath with ρK^2 basis vectors that are tiled as dictated by TiWGen's parameter M.

In order not to straggle the operation of `CNN-WGen`, the *OVSF generator* has to match the rate of the vector units by feeding them with M bits/cycle. A conventional design involves statically laying out the tiled vectors into a single buffer, with M ports and a depth equal to the number of reads per tile (*i.e.* #basis vectors×#subtiles). However, such a monolithic design would impose significant overheads as the basis vectors would have to be replicated either in the same address (*e.g.* when $M{>}K^2$) or in multiple addresses (*e.g.* storing rotated versions as required by different subtiles). This leads to inefficient utilisaiton of the on-chip memory due to excessive replication.

An alternative approach that would avoid the basis vector replication involves the instantiation of a K^2-deep OVSF memory with each location storing one K^2-bit vector. Such a design requires significantly lower amount of storage and provides an access rate of 1 vector/cycle by reading the appropriate address. Nonetheless, to obtain the M-bit subtile from the K^2-bit vectoc, complex multiplexer selection circuitry has to be instantiated. This approach can affect the maximum clock rate or add latency cycles to such an extent that any throughput gains would be outweighed.

To alleviate these limitations when mapping TiWGen's tiling scheme, a custom *OVSF generator* was developed. The top-level diagram of the *OVSF generator* is shown in Fig. 5. It is composed of three main components: the *OVSF FIFO*, a *basis vector aligner* and the *output register*. By introducing a FIFO for the OVSF vectors in combination with a *basis vector aligner*, the *OVSF generator* introduces a rate-matching mechanism that sustains the processing rate of the *vector compute datapath* while efficiently utilising the on-chip memory. The generator performs a different operation depending on the values of M and K^2 of layer i (Fig. 5).

Initially, the *OVSF FIFO* stores the $(K_i^2 K_i^2)$-bit basis vectors. The current vector is read from the FIFO into the top register. If $M{\leq}K_i^2$, the M least significant bits (LSBs) are outputted to the *vector compute datapath*. At the same time, the basis vector is processed by the *basis vector aligner*, which performs an M-bit left circular shift and writes the rotated vector to the *OVSF FIFO*. If $M{>}K_i^2$, the basis vector is self-concatenated $\left\lfloor \frac{M}{K_i^2} \right\rfloor$ times and written to the output's LS part. Simultaneously, the $\text{mod}(M, K_i^2)$ LSBs of the basis vector are written to the output's MSBs and the constructed vector is passed to the compute datapath. Here, the aligner performs a left circ-shift of K_i^2-$\text{mod}(M, K_i^2)$ bits and writes the result to the FIFO.

With this approach, when the basis vectors are read again out of the OVSF FIFO after ρK_i^2 cycles (*i.e.* in the next iteration of the loop on line 2), they are correctly aligned to directly match TiWGen's tiling pattern. For instance, after the generation of the red-striped subtile in Fig. 4, the FIFO-read basis vectors will be correctly aligned in order to generate the blue-striped subtile without the need for costly selection logic or redundant storage. For CNNs with multiple filter sizes, the *basis vector aligner* is instantiated with as many circ-shift options. As the distinct filter sizes are known *a priori*, only the required shifting logic is inserted, avoiding expensive generic multiplexers, and the appropriate per-layer bit-shift is selected at run time.

Overall, the proposed design offers two main benefits. First, it alleviates the redundant replicated storage of basis vectors and avoids the hardware cost of partitioning multiplexers that would require excessive LUTs usage. Second, it provides the necessary bandwidth to the *vector compute datapath* while efficiently utilising the on-chip memory through the *OVSF FIFO* and the resource-efficient aligner design. As the values of K_i^2 for each layer and M are known at design time and after the DSE phase respectively, the *OVSF generator* can be statically instantiated at compile time.

4.3 Input Selective PEs for Counteracting Underutilisation

One key limitation of existing CNN engines is that, when processing compute-bound layers, the layer dimensions often do not match the fixed processing engine configuration, leading to underutilisation of the computational resources and severe performance penalties [54, 62, 84, 103]. Such a scenario can be observed when mapping a layer with N_{out}=64 channels (*i.e.* C=64) on an engine with 128 PEs (*i.e.* T_C=128). In this case, the PEs would remain idle 50% of the time, halving the attainable performance.

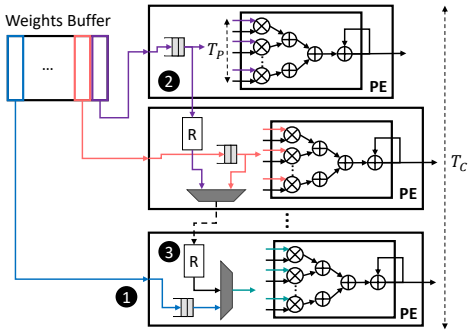


Fig. 6. unzipFPGA's input selective PE array for CNNs.

To alleviate this, we propose *input selective PEs*, a design that enables existing PEs to perform load-balancing through inter-PE work-stealing in a resource-efficient manner. Fig. 6 shows unzipFPGA's input selective PEs. The initial PE is augmented with registers and switches. However, not all PEs have the same components; only the PEs that remain *underutilised* even for a single layer are further equipped with a compact switch that selects the inputs to the dot-product circuit. In addition to the normal flow of data, these switches enable each PE to send its weight to its bottom neighbour. As highlighted in dark blue at the bottom PE of Fig. 6, the switch on the left of the PE selects its input from two options: *i)* under normal operation, the PE is fed with the weight written by `CNN-WGen` in the weights buffer (❶); *ii)* in the absence of this weight (*e.g.* due to a mismatch between C and T_C), the PE is fed with the weight passed by the adjacent PE (❸). In the second case, the weights are propagated along the PE array (❸) so that a different weight is used by each augmented PE in each cycle. Moreover, the Input Buffer (Fig. 4) is reorganised accordingly to provide parallel access to multiple rows.

Effectively, this design works as a load-balancing mechanism that partially unrolls the T_R dimension and thus distributes the work more evenly among the PEs. By restricting connectivity to adjacent units and enhancing only the underutilised PEs, the additional circuitry is low-overhead and delivers up to 20% higher performance on compute-bound layers.

5 DESIGN SPACE EXPLORATION

Based on its parametrisation of the processing engine, buffer sizes and weights generator, unzipFPGA defines a particular architectural design space. To estimate the performance and resource usage of different configurations, an analytical modelling framework has been developed. At a high-level, the key decisions for yielding a high-performance configuration of the system are: the allocation of the on-chip resources between the CNN engine and the weights generator and, the sizes of the activations buffers. The design-time tunable parameters comprise *1)* M that determines the TiWGen's tile size and the size of `CNN-WGen`'s vector units, *2)* tile sizes T_C and T_P that determine the number of PEs and MACs per PE, respectively, and T_R affecting the size of the activations buffers.

5.1 Performance Model

The workload of a CNN with N_L layers is represented as a sequence of W_i=$\langle R_i, P_i, C_i\rangle$ *workload tuples* with $i \in \{1, ..., N_L\}$. Given a design point σ=$\langle M, T_R, T_P, T_C\rangle$, the `CNN-WGen`'s runtime for generating the i-th layer's weights required to compute a $(T_R \times T_C)$ output tile is given by

$$t^i_{\texttt{CNN-WGen}}(\sigma, W_i) = \lfloor \rho \cdot l \rfloor \cdot \left\lceil \frac{T_P \cdot T_C}{M} \right\rceil \cdot \left\lceil \frac{P_i}{T_P} \right\rceil \tag{5}$$

where ρ and l are the OVSF ratio and basis length, respectively, and with one factor for each of the pipelined loops in Algorithm 1. With α values transferred upfront and the OVSF method generating

all weights on-chip, the off-chip memory transfers involve only the input/output activations

$$t^i_{\text{mem in}}(\sigma, W_i) = \frac{T_R \cdot P \cdot WL}{bw_{\text{in}}}, \quad t^i_{\text{mem out}}(\sigma, W_i) = \frac{T_R \cdot T_C \cdot WL}{bw_{\text{out}}} \tag{6}$$

where WL is the adopted wordlength, and $bw_{\{\text{in,out}\}}$ are the memory bandwidths for transferring inputs/outputs.

With T_C and T_P dimensions unrolled, computing an output tile requires the pipelined processing of $\frac{P_i}{T_P}$ tiles for each of the T_R rows. Hence, the processing engine's runtime for each output tile is estimated as $t^i_{\text{eng}}(\sigma, W_i) = T_R \left\lceil \frac{P_i}{T_P} \right\rceil$. With the input selective PEs, the runtime is refined as

$$t^i_{\text{eng}^*}(\sigma, W_i) = \left(T_C - C_i + \left\lceil \frac{T_R \cdot C_i - (T_C - C_i) \cdot (C_i + 1)}{T_C} \right\rceil \right) \cdot \left\lceil \frac{P_i}{T_P} \right\rceil \tag{7}$$

where dimension T_R is partially unrolled by processing rows of T_R through the underutilised PEs.

Overall, the accelerator forms a pipeline of three coarse stages: the concurrent input transfer and weights generation, the CNN engine processing and the output transfer. In this context, the initiation interval of the architecture is given by the maximum initiation interval of the three-stage pipeline, calculated as

$$II^i(\sigma, W_i) = \max\left(\max\left(t^i_{\text{mem in}}, t^i_{\text{CNN-WGen}} \right), t^i_{\text{eng}^*}, t^i_{\text{mem out}} \right) \tag{8}$$

As such, the total runtime for layer i is given by $t^i_{\text{total}}(\sigma, W_i) = II^i(\sigma, W_i) \left\lceil \frac{R_i}{T_R} \right\rceil \left\lceil \frac{C_i}{T_C} \right\rceil$. Thus, for a CNN with N_L layers, the workload tuple is $W{=}\langle W_i \mid \forall i \in \{1, ..., N_L\} \rangle$ and the throughput in inferences per sec (inf/s) is estimated as $T(\sigma, W) = 1/\sum_{i=1}^{N_L} t^i_{\text{total}}(\sigma, W_i)$.

5.2 Resource Consumption Model

The primary factor that constrains the mapping of a CNN engine on a given platform is resource availability. Each candidate configuration has a corresponding resource consumption. We define the *feasible space* of our model as the set of configurations that satisfy all the platform-specific resource constraints. In our context, the main design constraints are the DSPs and on-chip RAM blocks of the target FPGA. Assuming that all MAC operators are mapped to DSPs, the values of $\langle M, T_P, T_C \rangle$ are constrained as $D_{\text{MAC}} \times (M + T_P T_C) \leq D_{\text{fpga}}$, with D_{fpga} the available DSPs and D_{MAC} the DSPs/MAC. We consider 16-bit fixed-point precision, where $D_{\text{MAC}}{=}1$ on the evaluated Xilinx FPGAs.

In terms of on-chip RAM, the accelerator has the I/O and Alpha buffers with wordlength WL and the binary OVSF FIFO, with a total capacity requirement as given by Eq. (9).

$$\left(2(T_R T_P + T_R T_C) + D^{\text{Alpha}} N_P^{\text{Alpha}} \right) WL + K^2_{\text{max}} K^2_{\text{max}} \leq C_{\text{fpga}} \tag{9}$$

where the factor of 2 accounts for double-buffering and C_{fpga} is the on-chip RAM capacity of the target device.

To further estimate the consumption of look-up tables (LUTs), we used a set of place-and-route measurements and fitted linear regression models as a function of unzipFPGA's tunable parameters. Overall, we formally capture the resource consumption of a design point σ by means of vector $\boldsymbol{rsc}(\sigma)$ that holds the utilised amount of DSPs, BRAMs and LUTs. Similarly, we denote the FPGA resource vector by $\boldsymbol{rsc}_{\text{Avail.}}$.

5.3 Configuration Optimisation Framework

To yield the highest performing design for the given CNN-FPGA pair, we cast the DSE task as a constrained optimisation problem that aims to determine the values of the configurable parameters

$\langle M, T_R, T_P, T_C \rangle$ that achieve the highest performance for the target CNN and available hardware resources. Formally, we express this setup as

$$\min_{\sigma=\langle M,T_R,T_P,T_C\rangle} T(\sigma, W) \quad \text{s.t.} \quad \boldsymbol{rsc}(\sigma) \leq \boldsymbol{rsc}_{\text{Avail.}} \tag{10}$$

where T, $\boldsymbol{rsc}$ and $\boldsymbol{rsc}_{\text{Avail.}}$ are the throughput in inferences per second (inf/s), the resource consumption of the current design point σ and the resource vector of the target platform, respectively. Given a CNN-FPGA pair, we perform exhaustive search for different resource allocations between `CNN-WGen` and the processing engine. Designs that violate the resource constraints are pruned as infeasible to accelerate the exploration.

6 DERIVING LIGHTWEIGHT OVSF MODELS

Having presented unzipFPGA's hardware architecture, its strategy for mapping OVSF models on the accelerator and its design space exploration process, we now describe important challenges for constructing efficient OVSF models. The main challenges comprise: *i)* extracting correctly-sized filters from OVSF codes, *ii)* selecting a subset of OVSF vectors to meet a given OVSF ratio for each layer, and *iii)* setting the per-layer OVSF ratios themselves. Section 6.1 discusses our approach to *i)* and *ii)*, while Section 6.2 introduces our novel hardware-aware scheme for tuning OVSF ratios.

6.1 Practical Considerations to Train OVSF Models

Unlike standard CNNs, architectures using OVSF codes do not learn convolutional filters directly. Instead, they learn weighting coefficients for each OVSF code representing a filter. However, despite their simplicity as a straight drop and replacement option for standard convolutional layers, the nature of OVSF codes and the filter generation process, present two fundamental challenges: *1)* OVSF codes are of power-of-two length: This constrains the generation of filters with all N_{out}, N_{in}, and K being power-of-two integers. While this might be reasonable for the input and output channel dimensions, it prevents the construction of 3×3 filters, which are ubiquitous in modern CNN architectures; and *2)* choosing a subset of basis: Model compression is only achieved when OVSF ratio $\rho < 1$, which raises the question of *which bases to choose* from the total L available for OVSF codes of length L. Intuitively, their should be an optimal subset of basis for a given $\rho < 1$ that allows the learning of more expressive filters.

For *1)*, we consider between *i)* utilising the first $\lfloor \rho \cdot K^2 \rceil$ codes and *ii)* iteratively discarding OVSF codes based on their associated scalar α until the target compression ratio ρ is reached. Compared to *ii)*, with *i)* we have a simpler optimization objective at the expense of potentially limiting the expressivity of OVSF filters. For *2)*, we consider *i)* extracting a 3×3 crop from a 4×4 filter and *ii)* learning a mapping to a 3×3 filter by means of an average pooling layer. Similarly to the first pair of solutions, *i)* represents a simpler training stage at the expense of a reduced effective field over the OVSF basis when constructing 3×3 filter. In Sec. 7.1.2, we compare both pairs of approaches for the above challenges.

In certain scenarios, a pre-trained model with standard convolutions might be available or can be trained very cheaply. In such cases, the formulation in Eq. (2) could be reinterpreted as a minimisation problem and regress the set of $\boldsymbol{\alpha}_i^*$ that minimise the difference w.r.t the standard filter $\hat{f}_i$ as $\boldsymbol{\alpha}^* = \text{argmin}_\alpha \left\| f - \hat{f} \right\|_2^2$, which can be implemented as a 2-layer MLP regression stage. We leverage this strategy when training OVSF models on ImageNet. More details are provided in Section 7.1.3.

6.2 Hardware-Aware Tuning of OVSF Ratios

A critical component of unzipFPGA is the *OVSF Ratios Selection* module of the *OVSF Model Converter* (Fig. 2). In the original work [87], the OVSF ratios (*i.e.* ρ for each layer) were manually selected in a

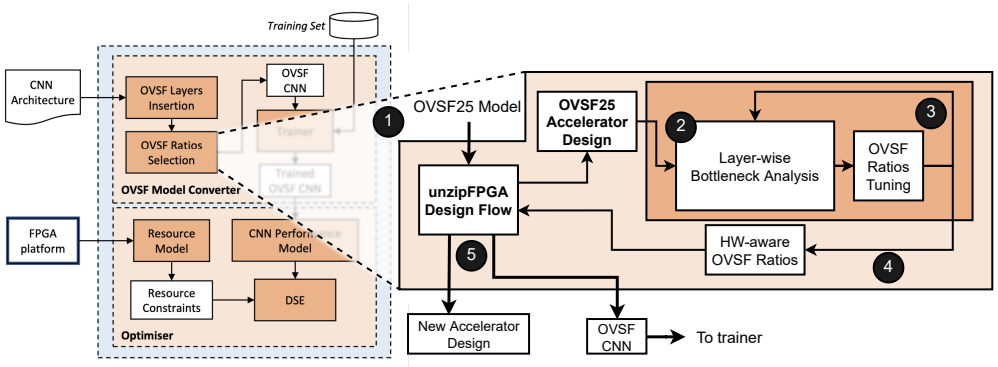


Fig. 7. Overview of the proposed hardware-aware tuning of the OVSF ratios.

coarse, per-block manner, with the objective to reach a given compression ratio while minimising accuracy degradation. For example, as detailed in Section 7.1.3, to achieve a compression of 50% in model size, denoted by OVSF50, the hand-tuned ratios were set by the tuple $[1.0, 0.5, 0.5, 0.5]$, indicating the OVSF ratio for each of the four blocks comprising a ResNet. Lower ratios were assigned to deeper layers as these contain a larger portion of the model parameters and are known to be more resilient to compression. The first CONV layer in the network remains untouched (*i.e.* not OVSF) as it has been shown to be less resilient to approximations including quantisation [7]. To reach higher compression ratios with minimal accuracy drop, more involved tuning is required. This can be observed through the OVSF25 variant which achieves 75% compression with diverse ratios of $[1.0, 0.4, 0.25, 0.125]$. Nonetheless, this process neither takes into account the impact of ratio values on hardware performance nor is automated, requiring elaborate tuning.

To alleviate this, we introduce a hardware-aware autotuning scheme for selecting the OVSF ratios of a given model on a target device. The key insight behind our method is that, for layers that are either compute- or memory-bound, we can allow the weights generation stage to consume more cycles by using more OVSF vectors (*i.e.* using a higher OVSF ratio) *without* affecting the processing speed. As such, `CNN-WGen` will output a better approximation of the layer's weights, increasing the model's expressivity and potentially improving accuracy. With reference to our performance model (Section 5.1), this case occurs for layer i when either the processing engine's runtime ($t^i_{\text{eng}*}$) or the off-chip memory transfers ($t^i_{\text{mem in}}$ or $t^i_{\text{mem out}}$) dominate the initiation interval in Eq. (8). This allows us to allocate more cycles for $t^i_{\texttt{CNN-WGen}}$ by using a higher OVSF ratio and obtaining a better approximation of the weights.

With this insight, Fig. 7 presents our hardware-aware autotuning scheme is as follows. As a first step, we run unzipFPGA's design flow (Fig. 2) using the OVSF25 ratios (*e.g.* $[1.0, 0.4, 0.25, 0.125]$ for ResNet) and derive the corresponding accelerator configuration (❶). Next, we perform a bottleneck analysis of each layer's mapping on the accelerator that indicates which stage dominates the initiation interval (❷), *i.e.* whether it is memory-bound (either input or output activation transfer), compute-bound or weights-generation-bound. For the layers where `CNN-WGen` is *not* the bounding factor, we iteratively increase the OVSF ratios up to the point where the bottleneck does not shift to the weights generation stage (❸). This leads to a more balanced pipelining of each layer, hence increasing accuracy by better utilising the instantiated accelerator. At the end of this process, the converged set of OVSF ratios are passed as the output of the *OVSF Ratios Selection* module (Fig. 2) and the rest of unzipFPGA's flow is run (❹). As such, the model is retrained and the design space exploration is rerun with the new OVSF ratios (❺), and the final model-accelerator pair are deployed on the target FPGA. Despite the additional retraining step, we note that the

Table 1. Different OVSF ratio selection methods with respect to accuracy and bottleneck stage for ResNet18.

Memory Bandwidth	OVSF Ratio Selection Method	Accuracy (%)		Layer ID L0	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L11	L12	L13	L14	L15	L16	L17	L18	L19
1.1 GB/s	OVSF25	67.3	Bound	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM
			OVSF Ratio	1.0	1.0	1.0	1.0	1.0	0.4	0.4	1.0	0.4	0.4	0.25	0.25	1.0	0.25	0.25	0.125	0.125	1.0	0.125	0.125
	uniform-1.0	N/A	Bound	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM
			OVSF Ratio	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	hw-aware-autotuning	68.5	Bound	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM	IFM
			OVSF Ratio	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.25
2.2 GB/s	OVSF25	67.3	Bound	IFM	IFM	IFM	IFM	IFM	C	C	IFM	C	C	C	C	IFM	C	C	C	C	IFM	C	C
			OVSF Ratio	1.0	1.0	1.0	1.0	1.0	0.4	0.4	1.0	0.4	0.4	0.25	0.25	1.0	0.25	0.25	0.125	0.125	1.0	0.125	0.125
	uniform-1.0	N/A	Bound	IFM	IFM	IFM	IFM	IFM	C	C	IFM	C	C	C	C	IFM	C	C	C	W	IFM	W	W
			OVSF Ratio	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	hw-aware-autotuning	68.4	Bound	IFM	IFM	IFM	IFM	IFM	C	C	IFM	C	C	C	C	IFM	C	C	C	C	IFM	C	C
			OVSF Ratio	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.25
4.4 GB/s	OVSF25	67.3	Bound	IFM	C	C	C	C	C	C	OFM	C	C	C	C	IFM	C	C	C	C	IFM	C	C
			OVSF Ratio	1.0	1.0	1.0	1.0	1.0	0.4	0.4	1.0	0.4	0.4	0.25	0.25	1.0	0.25	0.25	0.125	0.125	1.0	0.125	0.125
	uniform-1.0	N/A	Bound	IFM	C	C	C	C	C	C	OFM	C	C	W	W	IFM	W	W	W	W	IFM	W	W
			OVSF Ratio	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	hw-aware-autotuning	67.6	Bound	IFM	C	C	C	C	C	C	OFM	C	C	C	C	IFM	C	C	C	C	IFM	C	C
			OVSF Ratio	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.333	0.333	0.5	0.333	0.333	0.333	0.25	0.25	0.25	0.25	0.25

* IFM: Memory-bound w.r.t. input feature maps | OFM: Memory-bound w.r.t. output feature maps | C: Compute-bound | W: Weights Generation-bound.

training protocol, *i.e.* all hyperparameters, remains the same throughout, without the need for further tuning.

In step ❸, the candidate values of ratio ρ^l for the l-th layer lie in $\left\{L^l/n \mid \forall\, n \in [1, L^l]\right\}$, where $L^l = N_{\text{in}}^l K^l K^l$ is the layer's code length (Sec. 2.2). As such, there are $|L^l|$ candidate OVSF ratio values per OVSF layer. Overall, for a CNN with N_L OVSF layers, there is a total of $\prod_{l=1}^{N_L} |L^l|$ possible OVSF ratio combinations. With an increase in either the model's depth or an OVSF layer's width, an enumerative exhaustive search quickly becomes computationally intractable. To alleviate this, we perform a parallel search for all OVSF layers, starting from the OVSF25 configuration. At the first iteration, we first calculate the *throughput ratio* between the weights-generation stage and the bottleneck stage of each layer and set the respective OVSF ratio to the closest feasible value. Then, we search the neighbouring candidate ratios until we find the maximum value that does not turn the weights-generation stage into the bottleneck. Throughout our experiments, this process lead to an average of 5 iterations to converge to the final hardware-aware set of ratios across the examined models.

Table 1 illustrates the impact of our scheme, denoted by `hw-aware-autotuning`, using ResNet18 on Z7045 for varying bandwidth availability. In the most bandwidth-constrained case (1.1 GB/s), OVSF25 is memory-bound, with all layers being limited by the transfer of the input feature maps. Our method exploits severe memory-boundedness and selectively increases the OVSF ratios, leading to an accuracy improvement of 1.2pp over OVSF25 with no sacrifice of the processing speed. For medium bandwidth levels (2.2 GB/s), a number of OVSF25 layers become compute-bound. If we naively set all OVSF ratios to 1.0 (shown as `uniform-1.0`), several layers become bound by the weights generation stage. Instead, with our bottleneck-guided method, the weights are more accurately generated while no change occurs to the boundedness of each layer. This results in achieving the same throughput as OVSF25, but with a 1.1pp increase in accuracy. Finally, in the high-bandwidth case (4.4 GB/s), our method introduces a 0.3pp accuracy gain, without affecting the hardware performance.

Overall, our OVSF ratio selection method incorporates three features: *1)* it is fine-grained by allowing for different per-layer OVSF ratios within each block. As such, we obtain finer-grained control over the accuracy-compression trade-off; *2)* it bounds the accuracy drop from below by means of an informed initialisation of the ratio values. By starting from the OVSF25 ratios (*i.e.* our most lightweight setting), we guarantee the accuracy's lowest bound and, by allowing only increases in OVSF ratios, we ensure that these would only potentially contribute accuracy gains; and *3)* it is hardware-aware as it is guided by the bottleneck analysis of each layer's processing.

Table 2. FPGA platforms used for evaluation.

Platform	Processor	LUTs	Flip-Flops	DSPs	BRAM
Zynq 7045	Arm Cortex A9	218,600	437,200	900	2.40 MB
UltraScale+ ZU7EV	Arm Cortex A53	230,000	461,000	1,728	4.75 MB

7 EVALUATION

7.1 Experimental Setup

In our experiments, we target two widely used FPGA platforms with varied computational capabilities and memory resources (Table 2): the Xilinx ZC706 board mounting the mid-tier Zynq Z7045 and the Xilinx ZCU104 board with the more resource-rich Zynq UltraScale+ ZU7EV. The two platforms are based on the Xilinx Zynq-7000 SoC and UltraScale+ MPSoC architectures, respectively, integrating a dual-core Arm Cortex A9 CPU and a quad-core Arm Cortex A53 CPU, respectively, alongside an FPGA fabric on the same chip. Our hardware designs were synthesised and placed-and-routed with Xilinx Vivado HLS and Vivado Design Suite (v2019.2) and run on both boards, with operating clock frequencies of 150 MHz for ZC706 and 200 MHz for ZCU104, respectively. The achieved clock frequency is currently constrained by the technology of the target device and the use of HLS, which relies on the vendor's toolchain and does not allow for low-level optimisations to shorten the critical path.

The corresponding Arm CPU was used to set up the transactions with the off-chip memory, launch the execution of inference and measure the end-to-end performance of each design. unzipFPGA provides support for both custom fixed-point and floating-point precisions. For the evaluation, 16-bit fixed-point precision was used, following the practice of the FPGA works we compare with. The available off-chip memory bandwidth was controlled by using a different number of memory ports and amount of word packing, spanning from 1.1 GB/s (1×) to 13.4 GB/s (12×).

7.1.1 ***Benchmarks.*** We evaluate on CNNs of varying depth, workload and memory footprint. Each CNN has been selected to impose a different design challenge. In particular, we target the widely used family of residual networks [38] and map variants of different depths to evaluate the scalability of our design. Concretely, we use ResNet18, ResNet34 and ResNet50 on the ImageNet dataset. In addition to image classification, ResNet models are also found as backbone of other tasks including object detection [50], super-resolution [57] and semantic segmentation [18]. We also target SqueezeNet1.1 [42], to assess unzipFPGA's efficacy on a highly optimised network for resource-constrained devices.

7.1.2 ***Basis Selection and 3×3 extraction.*** The proposed on-the-fly formulation using OVSF codes allows for different strategies for: *i)* selecting which basis to use when $\rho<1$; and, *ii)* extracting 3×3 filters from *true* OVSF filters that are restricted to be of shape $K\times K$ with K being a power-of-two. In 6.1 we presented two solutions for each of aforementioned considerations. Table 3 shows our analysis of the different approaches on CIFAR-10 with ResNet18/34. For the basis selection strategy *i)*, iteratively dropping bases consistently yields higher-accuracy models. For *ii)*, as the models become more compact (*e.g.* for OVSF50/25), cropping achieves higher accuracy compared to the average pooling approach. Thus, we leverage these findings to inform the parametrisation for ImageNet for the rest of the evaluation.

7.1.3 ***Training Scheme.*** We have developed unzipFPGA's offline flow on top of *PyTorch* (1.5). To derive the OVSF models, we modified the official *PyTorch*-based ResNet by replacing all 3×3 convolutional layers within residual blocks with their OVSF counterparts. In all our experiments, we employed pre-trained ImageNet models from *torchvision* (0.6.0). After a regression stage that transforms standard models into OVSF ones, the models were fine-tuned for 30 epochs using an

Table 3. Impact on accuracy for i) each basis selection strategy and ii) method to extract 3×3 filters from 4×4 OVSF filters. Models trained on CIFAR-10, with ResNet18/34 adapted for this dataset adn the much smaller variants (†) proposed in [38]. Performing an iterative drop of bases, as opposed to taking the first $\lfloor \rho \cdot K^2 \rceil$, consistently results in better models. As model size is reduced, taking a 3×3 crop from a 4×4 filter performed better than using an average pooling stage.

Model Arch. (baseline)	Basis Strategy	Filters to 3×3	OVSF100 Param.	OVSF100 Acc.	OVSF50 Param.	OVSF50 Acc.	OVSF25 Param.	OVSF25 Acc.
ResNet18 93.2% 11.2M	Sequential	Crop	19.7	93.9	9.1	93.7	3.6	92.9
		Adaptive		93.7		93.8		93.0
	Iterative	Crop		94.1		93.6		**93.6**
		Adaptive		94.0		93.8		92.3
ResNet18† 91.3% 0.27M	Sequential	Crop	0.48	90.8	0.25	90.8	0.15	88.3
		Adaptive		91.1		91.2		88.5
	Iterative	Crop		91.1		91.3		**91.4**
		Adaptive		91.2		91.4		91.0
ResNet34 93.9% 21.3M	Sequential	Crop	37.7	94.1	17.6	93.9	7.2	93.4
		Adaptive		94.3		94.0		93.4
	Iterative	Crop		94.1		93.8		**94.3**
		Adaptive		93.8		93.7		93.2
ResNet34† 92.1% 0.46M	Sequential	Crop	0.82	92.3	0.43	91.4	0.26	89.3
		Adaptive		92.2		91.5		89.2
	Iterative	Crop		92.3		91.8		**92.2**
		Adaptive		92.4		91.7		91.7

Adam optimiser [49] and learning rate decay every 10 epochs. For each given model, we trained two OVSF variants following different distributions of ratios ρ for layers in each of the four residual blocks. First, OVSF50 with ratios=[1.0, 0.5, 0.5, 0.5]; and OVSF25 with ratios=[1.0, 0.4, 0.25, 0.125]. We follow the same procedure and ratios for SqueezeNet's *Fire* modules.

7.1.4 ***Baselines.*** We introduce two highly optimised single computation engines executing: *a)* the vanilla CNN and *b)* pruned variants. For *b)*, we use a state-of-the-art method [65] which applies channel pruning based on the first-order Taylor approximation contribution of each filter to the model's loss. This process is carried out iteratively until a target compression ratio is reached. We refer to a pruned model that keeps 82% of the filters as Tay82 and follow the same naming scheme for other ratios. The baseline architecture comprises the conventional CNN engine design shown in Fig. 3, with the weights transferred from the off-chip memory into the $T_P \times T_C$ weights buffer, if they do not fit on-chip. Both *a)* and *b)* are parametrised with tile sizes $\langle T_R, T_P, T_C \rangle$ and roofline modelling [102] is used to obtain the highest throughput configuration for the target CNN-FPGA pair.

7.2 Performance Comparison

This section analyses the performance of the proposed framework with respect to both our optimised baselines and existing FPGA work.

7.2.1 ***Comparison with Optimised Baselines.*** Tables 4 and 5 show the achieved validation set accuracy and actual performance of each design as measured on ZC706 under varying bandwidth budget. Across bandwidths (1×/2×/4× where 4× is the 4.5 GB/s peak measured bandwidth on ZC706), unzipFPGA's OVSF50 and OVSF25 designs outperform the faithful baseline by 2.1×/1.3×/1.1× and 2.1×/1.6×/1.2× respectively for ResNet34, and by 1.6×/1.6×/1.24× and 1.4×/1.5×/1.3× respectively for ResNet18. As bandwidth availability increases, the baseline becomes less memory-bound and the performance gap closes. Table 6 shows the comparison of unzipFPGA with the faithful baseline for SqueezeNet on ZU7EV with peak measured bandwidth of 13.4 GB/s (12×). Both OVSF50 and OVSF25 designs yield increasing throughput gains as the bandwidth becomes more restricted, with OVSF25 sustaining over 57% speedup for up to 4× bandwidth. Under 1× bandwidth, OVSF25 offers minimal

Table 4. Accuracy and number of parameters for ResNet34 models on ImageNet following different compression schemes. Performance measured on ZC706 at different memory bandwidths.

Model Arch.	Compression Method	Params (millions)	Accuracy (%)	Performance (inf/sec) (1×, 2×, 4×)
ResNet34	-	21.8	73.3	(8.6, 16.8, 28.7)
ResNet34	Tay82	17.4	72.7	(10.7, 21.0, 35.6)
ResNet34	Tay72	15.1	71.9	(13.3, 25.8, 44.0)
ResNet34	Tay56	9.4	67.8	(18.3, 36.3, 63.8)
ResNet34	Tay45	6.3	63.1	(21.8, 43.4, 79.8)
ResNet34	OVSF50	17.2	72.8	(18.1, 21.8, 31.1)
ResNet34	OVSF25	7.2	71.5	(18.4, 27.3, 33.5)
ResNet34	Tay82+OVSF50	13.2	71.1	(18.6, 30.0, 37.3)
ResNet34	Tay82+OVSF25	6.7	70.6	(18.8, 31.0, 38.9)
ResNet34	Tay72+OVSF50	11.9	70.3	(18.8, 32.0, 40.2)
ResNet34	Tay72+OVSF25	4.9	68.9	(18.9, 33.3, 42.0)

Table 5. Accuracy and number of parameters for ResNet18 models on ImageNet following different compression schemes. Performance measured on ZC706 at different memory bandwidths.

Model Arch.	Compression Method	Params (millions)	Accuracy (%)	Performance (inf/sec) (1×, 2×, 4×)
ResNet18	-	11.7	69.8	(12.0, 23.5, 40.1)
ResNet18	Tay88	9.1	68.8	(14.3, 28.0, 46.4)
ResNet18	Tay82	7.9	67.3	(14.3, 27.8, 45.4)
ResNet18	Tay72	6.0	64.8	(18.2, 35.3, 57.6)
ResNet18	Tay56	3.7	58.3	(23.8, 47.3, 82.2)
ResNet18	OVSF50	9.1	69.2	(19.4, 33.8, 49.9)
ResNet18	OVSF25	4.1	67.3	(19.4, 34.8, 51.0)
ResNet18	Tay82+OVSF50	6.3	66.2	(24.5, 43.2, 57.9)
ResNet18	Tay82+OVSF25	2.8	64.4	(24.5, 43.6, 59.7)

additional gains. This is because, below a compression ratio, even though the memory needs are further reduced, activations begin to dominate I/O, and hence further weights reduction does not provide significant benefits. Activations compression techniques [20, 68] can be orthogonally combined to obtain further gains.

Based on our evaluation using SqueezeNet, we observe that computation can take place fast due to its lighter workload. As such, the attainable performance depends on how rapidly we can feed the CNN Engine with new inputs. Specifically, for the 4× bandwidth configuration, all layers of SqueezeNet are memory-bound. On the other hand, at 12× bandwidth, 88% of the layers become compute-bound. As such, when there is restricted or medium availability of memory bandwidth, unzipFPGA significantly improves performance through our weights generation approach, with 78%, 74% and 55% higher throughput for the 1×, 2× and 4× bandwidth configurations, respectively (Table 6). This improvement gradually decreases as the available bandwidth increases, with 15% gain at 12× bandwidth.

Comparison with Pruned Baselines. Compared to the pruned baselines, unzipFPGA's OVSF models are more resilient at high compression ratios while resulting in similar accuracy at lower compression ratios. Informed by the analysis in Table 3, OVSF models are trainined to extract a 3×3 from a 4×4 and, to iteratively discard OVSF basis until the target compression ratio ρ for each layer is reached, as first discussed in Sec. 7.1.2. In terms of throughput, unzipFPGA delivers

Table 6. Comparing unzipFPGA with faithful baseline on SqueezeNet on ImageNet. Performance measured on the UltraScale+ ZCU104 platform at different memory bandwidths.

Model Arch.	Compression Method	Params (millions)	Accuracy (%)	Performance (inf/sec) (1×, 2×, 4×, 12×)
SqueezeNet	-	1.24	58.2	(72.9, 145.2, 290.4, 687.4)
SqueezeNet	OVSF50	1.07	57.6	(129.8, 252.9, 452.1, 792.1)
SqueezeNet	OVSF25	0.86	57.1	(129.8, 252.9, 456.8, 800.6)

Table 7. Comparison with prior FPGA work on ResNet18 (4.03 GOps), ResNet34 (7.40 GOps) & SqueezeNet (0.78 GOps).

Comparison with:	**Compiler-based Design**		**Compression-based Design**		**Light-CNN-tailored Design**	**Multi-Accelerator Designs**		
	ResNet18 [17]	unzipFPGA: ResNet18*	Sparse ResNet34 [59] using Deep Compression	unzipFPGA: ResNet34*	SqueezeNet [100]	SqueezeNet [75]		unzipFPGA: SqueezeNet*
FPGA	Z7045	Z7045	Z7045	Z7045	K325T	V485T	V690T	ZU7EV
Clock (MHz)	250	150	166	150	200	170	170	200
Precision	16b fixed	16b fixed	16b fixed	16b fixed	8b fixed	16b fixed	16b fixed	16b fixed
DSPs†	900	900	900	900	840	2800	3600	1728
Logic Capacity	218.6 kLUTs	218.6 kLUTs	218.6 kLUTs	218.6 kLUTs	203.8 kLUTs	303.6 kLUTs	433.2 kLUTs	230.0 kLUTs
Block RAM	2.40 MB	2.40 MB	2.40 MB	2.40 MB	1.95 MB	4.52 MB	6.46 MB	4.75 MB
DSP Util.†	28.4%	100%	56.8%	100%	83.8%	80%	80%	100%
inf/s	21.38	49.90	27.84	31.1	420.90	913.40	1173.00	792.20
inf/s/DSP†	0.0237	0.0576	0.0309	0.0369	0.2505	0.3260	0.3258	0.4584
inf/s/Logic	0.0978	0.2282	0.1273	0.1422	2.0652	3.0085	2.7077	3.444

* using OVSF50, ** batch size = 1, † 18×18, 19×18 and 25×18 DSP configurations, inf/s/DSP is adjusted based on precision for fair comparison (multiplied by 0.5 for 8b).

Table 8. Comparison with prior FPGA work on ResNet50 (8.41 GOps).

Comparison with:	**Compiler-based Designs**		**CNN-to-FPGA Toolflows**			**CNN-tailored Designs**	**Overlay Designs**	**Cloud-based Designs**	**Interconnect-aware Designs**	**Full-stack-optimised Designs**	
	Snowflake [31]	unzipFPGA: ResNet50*	xDNN [95]	DNNVM [96]	ALAMO [62]		ResNetAccel [63]	FTDL [76]	Cloud-DNN [19]	Scaling the Cascades [73]	Full-Stack [58]	unzipFPGA: ResNet50*
FPGA	Z7045	Z7045	VU9P	ZU9	Arria 10 GX1150	Stratix 10 GX2800	Arria 10 GX1150	VU125	VU9P	VU37P	Arria 10 GX1150	ZU7EV
Clock (MHz)	250	150	500	500	240	150	300	650	125	650	200	200
Precision	16b fixed	16b fixed	8b fixed	8b fixed	16b fixed	16b fixed	16b fixed	16b fixed	16b fixed	8b fixed	8b fixed	16b fixed
DSPs†	900	900	6840	2520	3036	11,520	3036	1200	3036	9024	3036	1728
Logic Capacity	218.6 kLUTs	218.6 kLUTs	1182.0 kLUTs	274.0 kLUTs	427.2 kALMs	933.0 kALMs	427.2 kALMs	716.0 kLUTs	1182 kLUTs	1304 kLUTs	427.2 kALMs	230.0 kLUTs
Block RAM	2.40 MB	2.40 MB	9.48 MB	4.01 MB	6.60 MB	28.62 MB	6.60 MB	11.075 MB	43.23 MB	42.61 MB	6.60 MB	4.75 MB
DSP Util.†	28.4%	100%	100%	83.8%	80%	80%	56.8%	100%	80.2%	95%	97%	100%
inf/s	17.7	28.18	153.57	80.95	71.38	77.55	33.93	151.22	71.94	766	197.23	71.71
inf/s/DSP†	0.0196	0.0313	0.0112	0.016	0.0235	0.0067	0.0111	0.1260	0.0105	0.0424	0.0324	0.0415
inf/s/Logic	0.0809	0.1289	0.0649	0.1477	0.1671	0.0831	0.0794	0.2112	0.0608	0.5874	0.4616	0.3117

* using OVSF50, ** batch size = 1, † 18×18, 19×18 and 25×18 DSP configurations, inf/s/DSP is adjusted based on precision for fair comparison (multiplied by 0.5 for 8b).

faster processing at more constrained bandwidths. Concretely, ResNet34-OVSF50 is 80% faster than Tay82 at 1× bandwidth, with less than 1 percentage point (pp) accuracy drop. Despite being almost identical in terms of model size and accuracy, Tay82's approach, which prioritises the pruning of layers with the least accuracy impact, leads to the pruning of mostly compute-bound layers when targeting ResNet34. On the other hand, ResNet34-OVSF50 compresses more effectively memory-bound layers, leading to significantly higher throughput at low bandwidths. A similar pattern is observed for ResNet18. At higher compression ratios, ResNet34-OVSF25 yields 3.7 pp higher accuracy than Tay56, despite using 25% fewer parameters.

To explore the benefits of combining unzipFPGA's OVSF execution scheme with channel pruning, we derive, train and map on unzipFPGA Tay-OVSF models. This results in competitive lightweight models that are not attainable through pruning alone. For instance, ResNet18 with Tay82+OVSF25 is 25% smaller than ResNet18-Tay56 and achieves 6.1 pp higher accuracy, while achieving 34.6% and 23.5% higher throughput over ResNet18-Tay72 with less than 0.5 pp accuracy drop.

7.2.2 ***Comparison with Existing FPGA Designs.*** To assess the performance of the proposed framework with respect to existing FPGA work, we perform a number of comparison with a broad range of state-of-the-art works that optimise CNN inference from different aspects. These span accelerators that aggressively apply compiler techniques [17, 31, 95, 96], the highest performing FPGA-based accelerators for sparse [59] and lightweight CNNs [100], a multi-accelerator design that addresses PE underutilisation for SqueezeNet [75], a state-of-the-art CNN-to-FPGA toolflow [62], an optimised overlay architecture [76], a highly customised accelerator for residual networks [63], a

cloud-optimised framework [19], a CNN accelerator designed in an interconnect-aware manner [73] and an accelerator that applies full-stack optimisations [58].

Table 7 lists the performance results for ResNet18/34 and SqueezeNet. On Z7045, unzipFPGA achieves 2.33× and 1.12× higher throughput than [17] and [59], respectively. For SqueezeNet, our design delivers 1.83× and 1.67× higher performance density in inf/s/DSP and inf/s/Logic than Light-OPU [100]. Compared to the multi-accelerator design [75] that also addresses the PE underutilisation, unzipFPGA yields 1.40× higher inf/s/DSP and 1.14×-1.27× higher inf/s/Logic despite having the same (V48T-based design [75]) or 36% lower (V690T-based design [75]) on-chip memory budget.

The original ResNet50 reaches 76.15% accuracy with a model size of 25.56M parameters. With unzipFPGA's ResNet50-OVSF50 variant improves accuracy to 76.23% while having 10% fewer parameters (22.84M). Table 8 presents the measured performance results for ResNet50. On Z7045, unzipFPGA outperforms Snowflake by 1.59× in inf/s. Compared with designs on larger devices, our design achieves higher performance density (inf/s/DSP) by 3.69×, 2.58×, 1.76×-6.16×, 3.17× and 3.94× over xDNN, DNNVM, ALAMO, ResNetAccel and Cloud-DNN. The overlay-based FTDL reaches higher inf/s/DSP and 1.47× lower inf/s/Logic, but targets a platform with 2.33× larger on-chip memory and 2× higher bandwidth, both of which substantially reduce the off-chip memory accesses and the associated latency. Similarly, compared to the interconnect-aware design of [73], unzipFPGA reaches 97.8% of its inf/s/DSP, despite using a platform with 8.9× smaller on-chip memory. Finally, unzipFPGA outperforms the full-stack-optimised accelerator of [58] by 1.28× in inf/s/DSP.

Discussion. Based on the presented evaluation, unzipFPGA consistently outperforms a wide range of FPGA-based accelerator designs, in spite of their diverse designs. As such, our framework delivers an average throughput gain of 2.23× (2.05× geo. mean) over designs that aggressively apply compiler optimisations on fixed accelerators [17, 31, 95, 96] and, at the same time, achieves an average inf/s/DSP gain of 2.5× (2.41× geo. mean) over highly customised CNN-tailored designs [63, 100] and 3.94× over the cloud-optimised mapping of Cloud-DNN. A notable comparison is with the sparse CNN accelerator for ResNet34 presented in [59], with unzipFPGA achieving 12% throughput gain. It should be noted that the sparse CNN accelerator applies Deep Compression [37] to sparsify the target CNN, employs a specialised dataflow and modifies the underlying PEs in order to extract high performance. In contrast, unzipFPGA improves the performance of CNN engines while affecting neither the selected dataflow nor the internal design of the PEs, and still delivers 12% higher throughput than the sparse CNN accelerator.

7.2.3 ***Resource Usage.*** We select unzipFPGA and baseline designs with up to 1-pp accuracy drop and compare their post place-and-route resource usage on Z7045, reported in [DSPs, BRAM, LUTs] tuples for 4× bandwidth. For ResNet34, the faithful baseline consumes [99%,83%,77%], Tay82 [99%,79%,77%], OVSF50 [100%,81%,78%] and Tay82+OVSF50 [100%,87%,81%]. For ResNet18, the faithful

Table 9. Resource usage breakdown of unzipFPGA's designs.

Design Config.	Platform	Resource Type	`CNN-WGen`	CNN Engine
ResNet18-OVSF50	ZC706	DSPs	7.5%	92.5%
		LUTs	1%	74%
ResNet34-OVSF50	ZC706	DSPs	11.3%	88.7%
		LUTs	3%	75%
ResNet50-OVSF50	ZC706	DSPs	11.1%	88.9%
		LUTs	3%	75%

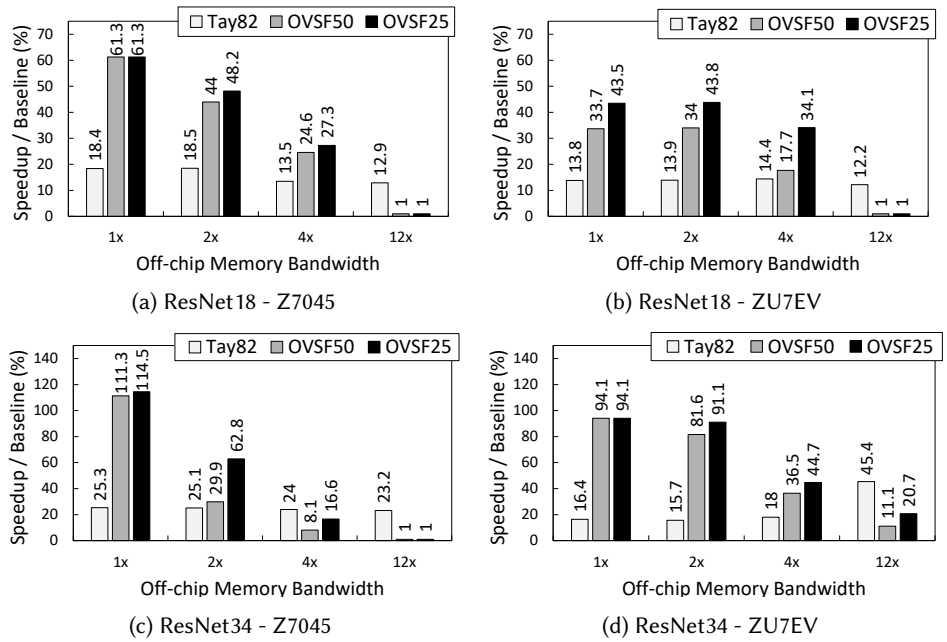


Fig. 8. Speedup over optimised baselines when varying the available off-chip memory bandwidth.

baseline [78%,99%,70%], Tay88 [78%,99%,69%], OVSF50 [100%,87%,75%] and Tay82+OVSF50 [100%,83%,80%]. For ResNet50, OVSF50 on ZU7EV consumes [100%,87%,78%]. Finally, the input selective PE mechanism adds a minimal LUTs overhead of less than 7%. We further report the breakdown of resource consumption between CNN-WGen and the CNN Engine in Table 9. We observe that, using our performance model, the DSE stage is able to balance the allocation of DSPs between the two modules. Moreover, the LUT overhead of the weights generator is minimal compared to the CNN Engine, providing a beneficial trade-off.

7.3 Sensitivity to Off-Chip Memory Bandwidth

Fig. 9 shows the impact of varying off-chip memory bandwidth over performance on the two target platforms. The figure compares the speedup of unzipFPGA and the Tay82 baseline over the vanilla baseline when varying the external memory bandwidth from 1× to 12×. The bandwidth's impact is most prominent on the larger ZU7EV, where the performance gains are sustained higher across 1×-4×. In the case of the mid-tier Z7045, we observe a sharper drop in the speedup as the bandwidth increases. This is due to the more limited computational resources of Z7045, which makes most CNN layers compute-bound. In contrast, the abundance of computational resources on ZU7EV makes the CNN layers more memory-bound. For instance, at 4× bandwidth (4.5 GB/s), the vanilla ResNet18 baseline yields DSP utilisation of 71% on the compute-bound Z7045 and 53% on the more memory-bound ZU7EV. In this case, unzipFPGA significantly improves both cases by mapping ResNet18-OVSF25 with 89% and 71% DSP utilisation. As a result, unzipFPGA sustains its gains across a wider range of bandwidths and outperforms Tay82, until the bandwidth-abundant case (12×) where computational resources become the critical factor. In this case, Tay82's lower number of operations due to pruning leads to higher performance.

Across the designs, the input selective PEs contribute an additional speedup of up to 20%, with varying gains depending on the CNN-FPGA pair and the available bandwidth. For ResNet34-OVSF25 on ZU7EV, disabling this mechanism leads to 0/13.9/3.3/5.9% lower throughput for the

Table 10. Ablation study of input selective PEs.

Model Arch.		FPGA Platform	Input Selective PEs without	with	Performance Gain
ResNet18	OVSF50	Z7045	49.9 inf/s	49.9 inf/s	1.00×
	OVSF25	Z7045	50.4 inf/s	51.0 inf/s	1.01×
	OVSF50	ZU7EV	124.1 inf/s	124.1 inf/s	1.00×
	OVSF25	ZU7EV	135.2 inf/s	135.2 inf/s	1.00×
ResNet34	OVSF50	Z7045	25.4 inf/s	31.1 inf/s	1.22×
	OVSF25	Z7045	33.5 inf/s	33.3 inf/s	0.6%
	OVSF50	ZU7EV	81.1 inf/s	81.1 inf/s	1.00×
	OVSF25	ZU7EV	72.4 inf/s	88.0 inf/s	1.21×
ResNet50	OVSF50	Z7045	23.7 inf/s	27.0 inf/s	1.14×
	OVSF25	Z7045	23.7 inf/s	28.1 inf/s	1.18×
	OVSF50	ZU7EV	63.1 inf/s	71.7 inf/s	1.13×
	OVSF25	ZU7EV	68.5 inf/s	77.8 inf/s	1.13×
SqueezeNet	OVSF50	ZU7EV	724.2 inf/s	792.2 inf/s	1.09×
	OVSF25	ZU7EV	731.4 inf/s	800.6 inf/s	1.09×
Average					1.12×
Geo. Mean					1.11×

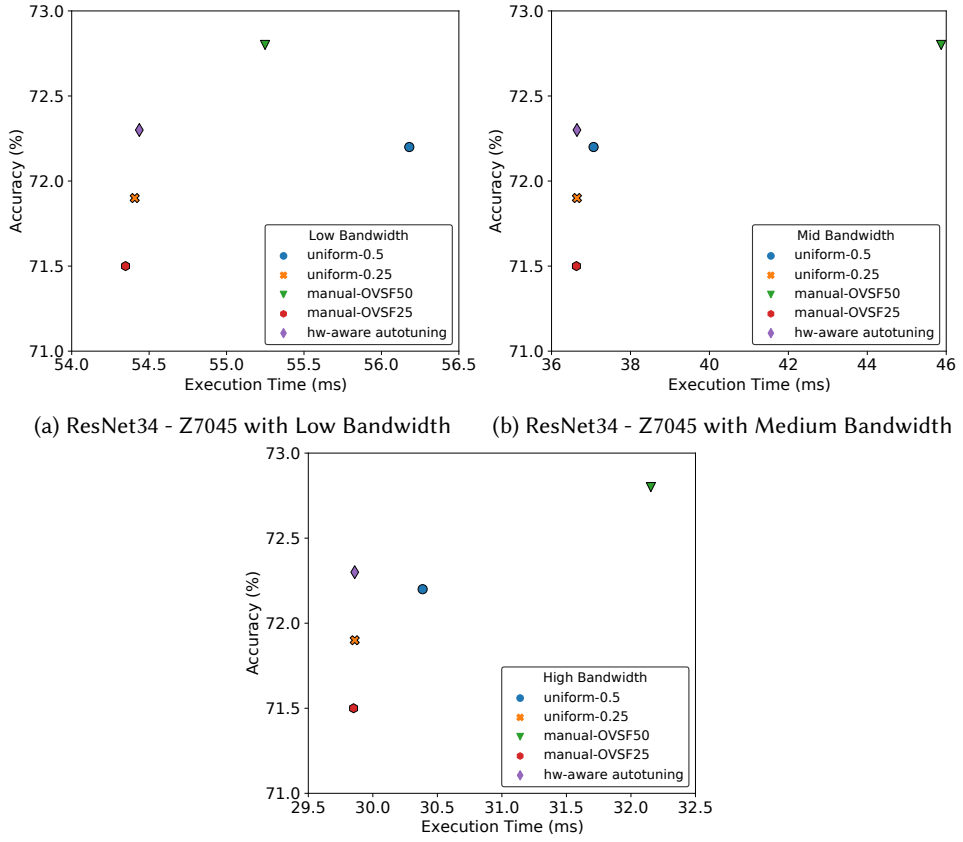
four bandwidths, with a similar pattern observed for the rest of the CNNs. Our input selective PEs effectively improve the performance of suboptimally mapped layers in compute-bound settings, whereas no gain is obtained for the most bandwidth-constrained case (1×) where the designs are severely memory-bound, limiting further improvements through higher PE utilisation.

7.4 Impact of Input Selective PEs

Here, we evaluate the impact of input selective PEs on the achieved performance. This is investigated by implementing unzipFPGA's selected hardware design for each of the benchmark CNNs with and without the input selective PEs and comparing the achieved performance, measured on the two target FPGA platforms. When the input selective PEs are omitted, we call the designs *ablated*. Table 10 presents the achieved performance gains between the two designs.

The PE-enhancing mechanism contributes varying throughput gains, yielding up to 22% faster inference and an average improvement of 12% (11% geo. mean). For ResNet18, the ablated designs already sustain high DSP utilisation with ResNet18-OVSF50 and -OVSF25, reaching 90% and 86.5% of the theoretical peak performance of Z7045 and ZU7EV, respectively. On the other hand, the ablated ResNet34-OVSF50 design on Z7045 achieves only 69.6% of the theoretical peak throughput. Similarly, the ablated ResNet34-OVSF25 design on ZU7EV achieves 77.5% of the theoretical performance. In both cases, the input selective PEs are able to substantially increase the DSP utilisation, with the enhanced CNN engines achieving 85.1% and 94.2% of the peak performance, respectively.

A similar effect is observed for ResNet50 and SqueezeNet. The ablated designs yield 73.8% of the theoretical peak performance for both OVSF50 and OVSF25 on Z7045, and 76.7% and 83.3% for OVSF50 and OVSF25, respectively, on ZU7EV. In this case, our input selective PEs are able to improve the DSP utilisation, achieving 84.1% and 87.4% of the peak throughput on Z7045 for OVSF50 and OVSF25, respectively, and 87.2% and 94.7% for OVSF50 and OVSF25, respectively, on ZU7EV. Finally, for SqueezeNet, the input selective PEs improve the measured throughput from 73.2% to 80.1% of the peak performance for OVSF50 and from 73.9% to 80.9% for OVSF25 on ZU7EV. As such, enhancing a CNN engine's PEs with our proposed input selectivity technique alleviates the resource underutilisation due to the diverse layer shapes within a CNN. In the cases where our technique is estimated to provide minimal gains (*i.e.* <5%) and its usage is not justified, unzipFPGA opts for omitting it to save LUT resources.



(a) ResNet34 - Z7045 with Low Bandwidth (b) ResNet34 - Z7045 with Medium Bandwidth

(c) ResNet34 - Z7045 with High Bandwidth

Fig. 9. Accuracy-execution time trade-off for different OVSF ratio selections. Hardware-aware autotuning closes the performance gap between OVSF50 and OVSF25 while being within 1pp of the original model's accuracy (73.3%).

7.5 Hardware-Aware vs. Manual Tuning of OVSF Ratios

Next, we evaluate the effectiveness of our hardware-aware tuning of OVSF ratios in yielding designs with improved accuracy-performance trade-off. To this end, we compare against two ratio selection methods: *i)* `uniform-`ρ which uses the same ratio ρ across all layers, with the exception of the first CONV layer. This baseline represents a brute-force approach of setting the OVSF ratios; and *ii)* `manual-OVSF50` and `manual-OVSF25` which use the manually selected ratios detailed in Section 7.1.3 to achieve 50% and 75% reduction in model size from the original model. This baseline constitutes an optimised hand-engineered method. We perform the comparison by implementing ResNet18 and ResNet34 using both the hardware-aware and the baseline flows for different bandwidth availability and comparing the achieved performance, measured on Z7045.

Figure 9a shows the achieved accuracy and execution time measured on the target FPGA and depicts how our method, denoted by `hw-aware autotuning`, yields Pareto-optimal designs that were previously unattainable. For ResNet34, our method sustains the same performance as the fast OVSF25 design across all memory bandwidths. However, it additionally improves OVSF25's accuracy by 0.8pp, thus outputting design that are within 1pp of the original model's accuracy (72.3% for all three bandwidths vs 73.3% for the vanilla ResNet34). At the same time, it is consistently faster than the coarse `uniform-0.5`. We obtain similar results for ResNet18, with the same processing

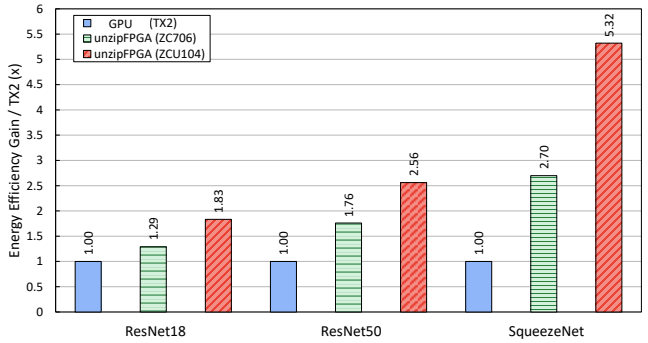


Fig. 10. Energy efficiency comparison between unzipFPGA and TX2 designs.

speed as OVSF25 and accuracy gains in the range of 0.3pp-1.2pp (0.86pp average gain across bandwidths) over OVSF25. Across all cases, the `uniform-`ρ baselines were either unnecessarily slow (`uniform-0.5`) or low in accuracy (`uniform-0.25`), further advocating for a principled method of selecting the OVSF ratios.

By exploiting the bounding factor of each layer, our hardware-aware scheme selectively allows for a longer weights generation stage without affecting the processing speed. As such, we can obtain a better approximation of the weights and sustain high throughput. As shown through our experiments, the hardware-aware methodology yielded competitive designs, performing either better or in par even against highly optimised hand-tuned configurations (OVSF50 and OVSF25).

7.6 Comparison with Embedded GPU

With the majority of CNNs deployed for inference on embedded and mobile devices, our evaluation focuses on the embedded space. In power-constrained applications, the main metrics of interest comprise: *1)* the absolute power consumption and *2)* the energy efficiency in performance-per-watt. In this respect, we investigate the energy efficiency of unzipFPGA in relation to the widely used high-performance NVIDIA Jetson TX2 platform. In all cases, for unzipFPGA we use the OVSF50 variant with less than 1-pp accuracy drop.

For the performance evaluation on TX2, we use NVIDIA TensorRT as supplied by the JetPack 4.1.1 package. TensorRT is run with the NVIDIA cuDNN library and 16-bit half-precision floating-point arithmetic (FP16), which enables the highly optimised execution of layers. Across all platforms, each CNN is run 100 times to obtain the average throughput. Furthermore, power measurements for the GPU and FPGAs are obtained via a power monitor on the corresponding board. In all cases, we subtract the average idle power from the measurement to obtain the power due to benchmark execution. The idle power of the FPGA platforms is measured at the board level with no design programmed in the FPGA fabric, so that the clock tree power and the power leakage of the chip are also included in the run-time power due to benchmark execution. Across all experiments, we used a batch size of 1, as is typical in mobile and embedded settings.

Tegra X2 mounts a 256-core GPU with native support for FP16 arithmetic which supports a range of operating modes with different clock frequencies and power consumption. To perform a fair comparison with respect to energy efficiency in terms of performance-per-watt, we configure the GPU with the maximum energy efficiency mode (Max-Q) which sets the frequency of the GPU at 850 MHz and configures all components of TX2 to achieve the best power-throughput trade-off. Fig. 10 presents the conducted comparison. unzipFPGA achieves an energy efficiency improvement over TX2 of up to 5.32× in inf/s/W with an average of 2.57× (2.31× geo. mean) across the benchmarks.

As a result, unzipFPGA consistently demonstrates significant gains in performance-per-watt across the benchmarks over highly optimised embedded GPU implementations.

8 CONCLUSION

In this work we have presented unzipFPGA, a framework for FPGA-based CNN accelerators that mitigates the limitations that prevent single computation engines from attaining high resource utilisation and throughput. By generating the layer weights on demand and selectively balancing the PE load, unzipFPGA outperforms both status-quo and pruned CNN engines for the same bandwidth, while largely improving performance density compared to diverse state-of-the-art CNN accelerators. Furthermore, we demonstrated the superiority of models optimised with unzipFPGA in terms of energy efficiency compared to these being deployed on embedded GPU platforms. The benefits of the proposed on-the-fly formulation brought the largest gains at reduced memory bandwidths, which we envision to be a turning point towards enabling multi-tenant FPGA-based CNN models running concurrently and sharing the same off-chip memory.

REFERENCES

[1] Mohamed S Abdelfattah, Łukasz Dudziak, Thomas Chau, Royson Lee, Hyeji Kim, and Nicholas D Lane. 2020. Best of Both Worlds: AutoML Codesign of a CNN and its Hardware Accelerator. In *Design Automation Conference (DAC)*.

[2] M. S. Abdelfattah, D. Han, A. Bitar, R. DiCecco, S. O'Connell, N. Shanker, J. Chu, I. Prins, J. Fender, A. C. Ling, and G. R. Chiu. 2018. DLA: Compiler and FPGA Overlay for Neural Network Inference Acceleration. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 411–4117.

[3] Fumiyuki Adachi et al. 1998. Wideband DS-CDMA for next-generation mobile communications systems. *IEEE communications Magazine* 36, 9 (1998), 56–69.

[4] F. Adachi, M. Sawahashi, and K. Okawa. 1997. Tree-structured generation of orthogonal spreading codes with different lengths for forward link of DS-CDMA mobile radio. *Electronics Letters* 33 (January 1997), 27–28(1). Issue 1.

[5] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. 2017. Bit-Pragmatic Deep Neural Network Computing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 382–394.

[6] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*. 1–13.

[7] Milad Alizadeh, Javier Fernández-Marqués, Nicholas D. Lane, and Yarin Gal. 2019. A Systematic Study of Binary Neural Networks' Optimisation. In *International Conference on Learning Representations (ICLR)*.

[8] Mario Almeida et al. 2019. EmBench: Quantifying Performance Variations of Deep Neural Networks Across Modern Commodity Devices. In *EMDL*.

[9] M. Alwani, H. Chen, M. Ferdman, and P. Milder. 2016. Fused-layer CNN accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12.

[10] Boris D. Andreev et al. 2003. Orthogonal Code Generator for 3G Wireless Transceivers. In *Proceedings of the 13th ACM Great Lakes Symposium on VLSI (GLSVLSI)*. 229–232. https://doi.org/10.1145/764808.764868

[11] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. 2017. An OpenCL™ Deep Learning Accelerator on Arria 10. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. 55–64.

[12] Arash Azizimazreah and Lizhong Chen. 2021. Polymorphic Accelerators for Deep Neural Networks. *IEEE Trans. Comput.* (2021).

[13] E. Baek, D. Kwon, and J. Kim. 2020. A Multi-Neural Network Acceleration Architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 940–953.

[14] Sourav Bhattacharya and Nicholas D. Lane. 2016. Sparsification and Separation of Deep Learning Layers for Constrained Resource Inference on Wearables. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems (SenSys)*. ACM, 176–189.

[15] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. 2020. What is the State of Neural Network Pruning?. In *Conference on Machine Learning and Systems (MLSys)*.

[16] Michaela Blott, Thomas B. Preußer, Nicholas J. Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. 2018. FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks. *ACM Trans. Reconfigurable Technol. Syst. (TRETS)* 11, 3, Article 16 (2018), 23 pages.

[17] Andre Xian Ming Chang, Aliasger Zaidy, Vinayak Gokhale, and Eugenio Culurciello. 2017. Compiling Deep Learning Models for Custom Hardware Accelerators. *arXiv preprint arXiv:1708.00117* (2017).

[18] L. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille. 2018. DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 40, 4 (2018), 834–848.

[19] Yao Chen, Jiong He, Xiaofan Zhang, Cong Hao, and Deming Chen. 2019. Cloud-DNN: An Open Framework for Mapping DNN Models to Cloud FPGAs. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[20] Y. Chen, T. Krishna, J. S. Emer, and V. Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits (JSSC)* 52, 1 (2017), 127–138.

[21] G. Csordas, M. Asiatici, and P. Ienne. 2019. In Search of Lost Bandwidth: Extensive Reordering of DRAM Accesses on FPGA. In *2019 International Conference on Field-Programmable Technology (ICFPT)*. 188–196.

[22] Chunhua Deng, Siyu Liao, Yi Xie, Keshab K. Parhi, Xuehai Qian, and Bo Yuan. 2018. PermDNN: Efficient Compressed DNN Architecture with Permuted Diagonal Matrices. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 189–202.

[23] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan, X. Ma, Y. Zhang, J. Tang, Q. Qiu, X. Lin, and B. Yuan. 2017. CirCNN: Accelerating and Compressing Deep Neural Networks Using Block-Circulant Weight Matrices. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 395–408.

[24] Zhen Dong, Zhewei Yao, Amir Gholami, Michael Mahoney, and Kurt Keutzer. 2019. HAWQ: Hessian AWare Quantization of Neural Networks with Mixed-Precision. In *IEEE International Conference on Computer Vision (ICCV)*.

[25] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. 2015. ShiDianNao: Shifting Vision Processing Closer to the Sensor. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 92–104.

[26] Javier Fernández-Marqués, Vincent W.-S. Tseng, Sourav Bhattachara, and Nicholas D. Lane. 2018. On-the-Fly Deterministic Binary Filters for Memory Efficient Keyword Spotting Applications on Embedded Devices. In *Proceedings of the 2nd International Workshop on Embedded and Mobile Deep Learning (EMDL)*. ACM, 13–18.

[27] Javier Fernández-Marqués, W-S Tseng Vincent, Sourav Bhattachara, and Nicholas D Lane. 2018. BinaryCmd: Keyword Spotting with deterministic binary basis. In *Conference on Machine Learning and Systems (MLSys)*.

[28] Javier Fernandez-Marques, Paul N. Whatmough, Andrew Mundy, and Matthew Mattina. 2020. Searching for Winograd-aware Quantized Networks. In *Conference on Machine Learning and Systems (MLSys)*.

[29] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 1–14.

[30] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[31] V. Gokhale, A. Zaidy, A. X. M. Chang, and E. Culurciello. 2017. Snowflake: An Efficient Hardware Accelerator for Convolutional Neural Networks. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–4.

[32] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and TN Vijaykumar. 2019. SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 151–165.

[33] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong. 2017. FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 152–159.

[34] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang. 2018. Angel-Eye: A Complete Design Flow for Mapping CNN Onto Embedded FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 37, 1 (2018), 35–47.

[35] David Ha, Andrew Dai, and Quoc V. Le. 2017. HyperNetworks. In *International Conference on Learning Representations (ICLR)*.

[36] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 243–254.

[37] Song Han, Huizi Mao, and William J Dally. 2015. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *International Conference on Learning Representations (ICLR)*.

[38] K He, X Zhang, S Ren, and J Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778.

[39] Yang He, Guoliang Kang, Xuanyi Dong, Yanwei Fu, and Yi Yang. 2018. Soft Filter Pruning for Accelerating Deep Convolutional Neural Networks. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

[40] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 319–333.

[41] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. 2016. A High Performance FPGA-based Accelerator for Large-Scale Convolutional Neural Networks. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 1–9.

[42] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[43] Andrey Ignatov et al. 2019. AI Benchmark: All About Deep Learning on Smartphones in 2019. In *ICCVW*.

[44] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[45] Jun-Woo Jang et al. 2021. Sparsity-Aware and Re-configurable NPU Architecture for Samsung Flagship Mobile SoC. In *ISCA*.

[46] Norman P Jouppi et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Annual International Symposium on Computer Architecture (ISCA)*.

[47] S. C. Kao and T. Krishna. 2020. GAMMA: Automating the HW Mapping of DNN Models on Accelerators via Genetic Algorithm. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9.

[48] S. Kim, M. Kim, C. Shin, J. Lee, and Y. Kim. 2009. Efficient implementation of OVSF code generator for UMTS systems. In *2009 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*. 483–486.

[49] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations (ICLR)*.

[50] A. Kouris, C. Kyrkou, and C. Bouganis. 2019. Informed Region Selection for Efficient UAV-based Object Detectors: Altitude-aware Vehicle Detection with CyCAR Dataset. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 51–58.

[51] A. Kouris, S. I. Venieris, and C. Bouganis. 2018. CascadeCNN: Pushing the Performance Limits of Quantisation in Convolutional Neural Networks. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 155–1557.

[52] A. Kouris, S. I. Venieris, and C. Bouganis. 2020. A Throughput-Latency Co-Optimised Cascade of Convolutional Neural Network Classifiers. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1656–1661.

[53] Raghuraman Krishnamoorthi. 2018. Quantizing Deep Convolutional Networks for Efficient Inference: A Whitepaper. arXiv:1806.08342 [cs.LG]

[54] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. 2020. MAESTRO: A Data-Centric Approach to Understand Reuse, Performance, and Hardware Cost of DNN Mappings. *IEEE Micro* 40, 3 (2020), 20–29.

[55] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 461–475.

[56] Alberto Delmás Lascorz, Sayeh Sharify, Isak Edo, Dylan Malone Stuart, Omar Mohamed Awad, Patrick Judd, Mostafa Mahmoud, Milos Nikolic, Kevin Siu, Zissis Poulos, and Andreas Moshovos. 2019. ShapeShifter: Enabling Fine-Grain Data Width Adaptation in Deep Learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 28–41.

[57] Royson Lee, Stylianos I. Venieris, Lukasz Dudziak, Sourav Bhattacharya, and Nicholas D. Lane. 2019. MobiSR: Efficient On-Device Super-Resolution through Heterogeneous Mobile Processors. In *The 25th Annual International Conference on Mobile Computing and Networking (MobiCom)*.

[58] Shuanglong Liu, Hongxiang Fan, Martin Ferianc, Xinyu Niu, Huifeng Shi, and Wayne Luk. 2021. Toward Full-Stack Acceleration of Deep Convolutional Neural Networks on FPGAs. *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)* (2021).

[59] L. Lu, J. Xie, R. Huang, J. Zhang, W. Lin, and Y. Liang. 2019. An Efficient Hardware Accelerator for Sparse Convolutional Neural Networks on FPGAs. In *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 17–25.

[60] 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.

[61] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. 2017. ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression. In *IEEE International Conference on Computer Vision (ICCV)*.

[62] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo. 2020. Automatic Compilation of Diverse CNNs Onto High-Performance FPGA Accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 39, 2 (2020), 424–437.

[63] Y. Ma, M. Kim, Y. Cao, S. Vrudhula, and J. Seo. 2017. End-to-End Scalable FPGA Accelerator for Deep Residual Networks. In *IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–4.

[64] K. Manev, A. Vaishnav, and D. Koch. 2019. Unexpected Diversity: Quantitative Memory Analysis for Zynq UltraScale+ Systems. In *2019 International Conference on Field-Programmable Technology (ICFPT)*. 179–187.

[65] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. 2019. Importance Estimation for Neural Network Pruning. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[66] Alexander Montgomerie-Corcoran and Christos Savvas-Bouganis. 2021. DEF: Differential Encoding of Featuremaps for Low Power Convolutional Neural Network Accelerators. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference (ASP-DAC*.

[67] Yue Niu, Rajgopal Kannan, Ajitesh Srivastava, and Viktor Prasanna. 2020. Reuse Kernels or Activations? A Flexible Dataflow for Low-Latency Spectral CNN Acceleration. In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)* (Seaside, CA, USA). 266–276.

[68] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally. 2017. SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 27–40.

[69] Adrien Prost-Boucle, Alban Bourge, and Frédéric Pétrot. 2018. High-Efficiency Convolutional Ternary Neural Networks with Custom Adder Trees and Weight Compression. *ACM Trans. Reconfigurable Technol. Syst. (TRETS)* 11, 3, Article 15 (2018), 24 pages.

[70] G. Purohit, V. K. Chaubey, K. S. Raju, and P. V. Reddy. 2013. FPGA based implementation and testing of OVSF code. In *2013 International Conference on Advanced Electronic Systems (ICAES)*. 88–92.

[71] Qiang Qiu, Xiuyuan Cheng, robert Calderbank, and Guillermo Sapiro. 2018. DCFNet: Deep Neural Network with Decomposed Convolutional Filters. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*. 4198–4207.

[72] T. Rintakoski, M. Kuulusa, and J. Nurmi. 2004. Hardware unit for OVSF/Walsh/Hadamard code generation [3G mobile communication applications]. In *2004 International Symposium on System-on-Chip (ISSOC)*. 143–145.

[73] Ananda Samajdar, Tushar Garg, Tushar Krishna, and Nachiket 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)*.

[74] Y. Shen, M. Ferdman, and P. Milder. 2017. Escher: A CNN Accelerator with Flexible Buffering to Minimize Off-Chip Transfer. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 93–100.

[75] 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)*.

[76] R. Shi, Y. Ding, X. Wei, H. Li, H. Liu, H. K. . H. So, and C. Ding. 2020. FTDL: A Tailored FPGA-Overlay for Deep Learning with High Scalability. In *57th ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[77] Shun Yan et al. 2021. An FPGA-based MobileNet Accelerator Considering Network Structure Characteristics. In *2021 31st International Conference on Field Programmable Logic and Applications (FPL)*.

[78] K. Siu, D. M. Stuart, M. Mahmoud, and A. Moshovos. 2018. Memory Requirements for Convolutional Neural Network Hardware Accelerators. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 111–121.

[79] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang. 2020. Tensaurus: A Versatile Accelerator for Mixed Sparse-Dense Tensor Computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 689–702.

[80] Vincent W.-S. Tseng, Sourav Bhattacharya, Javier Fernández Marqués, Milad Alizadeh, Catherine Tong, and Nicholas D. Lane. 2018. Deterministic Binary Filters for Convolutional Neural Networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI)*. 2739–2747.

[81] F. Tu, S. Yin, P. Ouyang, S. Tang, L. Liu, and S. Wei. 2017. Deep Convolutional Neural Network Architecture With Reconfigurable Computation Patterns. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems (TVLSI)* 25, 8 (2017), 2220–2233.

[82] Y. Umuroglu, Y. Akhauri, N. J. Fraser, and M. Blott. 2020. LogicNets: Co-Designed Neural Networks and Circuits for Extreme-Throughput Applications. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. 291–297. https://doi.org/10.1109/FPL50879.2020.00055

[83] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. 65–74.

[84] S. I. Venieris and C. Bouganis. 2017. Latency-Driven Design for FPGA-based Convolutional Neural Networks. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8.

[85] S. I. Venieris and C. Bouganis. 2019. fpgaConvNet: Mapping Regular and Irregular Convolutional Neural Networks on FPGAs. *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)* 30, 2 (2019), 326–342.

[86] S. I. Venieris and C. S. Bouganis. 2018. f-CNN^{x}: A Toolflow for Mapping Multiple Convolutional Neural Networks on FPGAs. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8.

[87] Stylianos I. Venieris, Javier Fernandez-Marques, and Nicholas D. Lane. 2021. unzipFPGA: Enhancing FPGA-based CNN Engines with On-the-Fly Weights Generation. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 165–175.

[88] Stylianos I. Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. 2018. Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions. *ACM Comput. Surv. (CSUR)* 51, 3, Article 56 (2018), 39 pages.

[89] Stylianos I. Venieris, Ioannis Panopoulos, and Iakovos S. Venieris. 2021. OODIn: An Optimised On-Device Inference Framework for Heterogeneous Mobile Devices. In *IEEE SMARTCOMP*.

[90] E. Wang, J. J. Davis, P. Y. K. Cheung, and G. Constantinides. 2020. LUTNet: Learning FPGA Configurations for Highly Efficient Neural Network Inference. *IEEE Transactions on Computers (TOC)* (2020).

[91] Tianzhe Wang, Kuan Wang, Han Cai, Ji Lin, Zhijian Liu, and Song Han. 2020. APQ: Joint Search for Network Architecture, Pruning and Quantization Policy. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[92] Yunhe Wang, Chang Xu, Chao Xu, and Dacheng Tao. 2017. Beyond Filters: Compact Feature Map for Portable Deep Model. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*. 3703–3711.

[93] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning Structured Sparsity in Deep Neural Networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NeurIPS)*.

[94] Carole-Jean Wu et al. 2019. Machine Learning at Facebook: Understanding Inference at the Edge. In *HPCA*.

[95] Xilinx. 2020. Adaptive Machine Learning Acceleration. https://www.xilinx.com/products/acceleration-solutions/xilinx-machine-learning-suite.html. [Retrieved: July 25, 2023].

[96] Y. Xing, S. Liang, L. Sui, X. Jia, J. Qiu, X. Liu, Y. Wang, Y. Shan, and Y. Wang. 2019. DNNVM: End-to-End Compiler Leveraging Heterogeneous Optimizations on FPGA-based CNN Accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2019).

[97] Lei Yang, Zheyu Yan, Meng Li, Hyoukjun Kwon, Weiwen Jiang, Liangzhen Lai, Yiyu Shi, Tushar Krishna, and Vikas Chandra. 2020. Co-Exploration of Neural Architectures and Heterogeneous ASIC Accelerator Designs Targeting Multiple Tasks. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference (DAC)*.

[98] Yingzhen Yang, Jiahui Yu, Nebojsa Jojic, Jun Huan, and Thomas S. Huang. 2020. FSNet: Compression of Deep Convolutional Neural Networks by Filter Summary. In *International Conference on Learning Representations (ICLR)*.

[99] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. 2017. Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*. 548–560.

[100] Yunxuan Yu, Tiandong Zhao, Kun Wang, and Lei He. 2020. Light-OPU: An FPGA-Based Overlay Processor for Lightweight Convolutional Neural Networks. In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. 122–132.

[101] Y. Yu, T. Zhao, M. Wang, K. Wang, and L. He. 2020. Uni-OPU: An FPGA-Based Uniform Accelerator for Convolutional and Transposed Convolutional Networks. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)* 28, 7 (2020), 1545–1556.

[102] 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)*. 161–170.

[103] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong. 2019. Caffeine: Toward Uniformed Representation and Acceleration for Deep Convolutional Neural Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 38, 11 (2019), 2072–2085.

[104] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen. 2016. Cambricon-X: An Accelerator for Sparse Neural Networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12.

[105] Y. Zhao, X. Gao, X. Guo, J. Liu, E. Wang, R. Mullins, P. Y. K. Cheung, G. Constantinides, and C. Xu. 2019. Automatic Generation of Multi-Precision Multi-Arithmetic CNN Accelerators for FPGAs. In *2019 International Conference on*

Field-Programmable Technology (ICFPT). 45–53.

[106] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen. 2018. Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 15–28.