Continuous-Flow Data-Rate-Aware CNN Inference on FPGA

***Abstract*—Among hardware accelerators for deep-learning inference, data flow implementations offer low latency and high throughput capabilities. In these architectures, each neuron is mapped to a dedicated hardware unit, making them well-suited for field-programmable gate array (FPGA) implementation. Previous unrolled implementations mostly focus on fully connected networks because of their simplicity, although it is well known that convolutional neural networks (CNNs) require fewer computations for the same accuracy. When observing the data flow in CNNs, pooling layers and convolutional layers with a stride larger than one, the number of data at their output is reduced with respect to their input. This data reduction strongly affects the data rate in a fully parallel implementation, making hardware units heavily underutilized unless it is handled properly. This work addresses this issue by analyzing the data flow of CNNs and presents a novel approach to designing data-rate-aware, continuous-flow CNN architectures. The proposed approach ensures a high hardware utilization close to 100% by interleaving low data rate signals and sharing hardware units, as well as using the right parallelization to achieve the throughput of a fully parallel implementation. The results show that a significant amount of the arithmetic logic can be saved, which allows implementing complex CNNs like MobileNet on a single FPGA with high throughput.**

***Index Terms*—CNN, continuous-flow, parallel architecture, unrolled architecture**

I. Introduction

CONVOLUTIONAL neural networks (CNNs) have proven to be powerful in various applications such as image classification [1]–[3], object detection [4], [5], or natural language processing (NLP) [6]. Various tasks, such as autonomous driving [7], speech recognition [8], server/cloud computing, augmented or virtual reality [9], and high energy physics [10], require low-latency and high-throughput processing. For these tasks, the use of neural networks (NNs) is challenging due to their high computational requirements, especially for applications that require real-time inference. To meet these demands, inference accelerators are employed to enhance performance and efficiency.

Inference accelerators can be categorized into three types [11, p. 734]: general accelerators, data flow or stream architectures, and unrolled architectures. General accelerators are usually implemented on custom integrated circuits (ICs) or application-specific ICs (ASICs) as those are fixed and can not be reconfigured like FPGAs. Hence, generic architectures like systolic arrays are used [12] that use a matrix of processing engines to target large and medium-sized NNs [13], [14]. Stream architectures [15]–[18] also target large and medium-sized NNs but provide a distinct hardware unit for each layer. They are network-specific and are more tailored and thus have typically lower resource and memory footprint than generic architectures [11]. Finally, unrolled architectures generally target small-sized NNs and are highly network-specialized and optimized [18]–[26]. In contrast to generic accelerators and stream architectures, unrolled architectures operate fully in parallel rather than sequentially, and most implementations of unrolled [19]–[22], [26] or partially unrolled architectures [23] are restricted to fully connected NNs.

To reduce the large resource requirements in fully unrolled NNs, many techniques have been proposed. They include quantization [27]–[29], pruning [30]–[32], specific LUT-mappings [19]–[24] for field-programmable gate array (FPGA) implementations, and reducing the neuron inputs. However, fully connected NNs still lack the scalability to implement larger models without exploding hardware costs.

An alternative to implement these larger models is to use CNNs. They have fewer computational requirements, as only a small data window is considered in the convolutional layer. However, convolutional and pooling layers lead to changes in the data rate. For example, in the common 2×2 max-pooling layer, the data rate at the output of the layer is one-fourth of the data rate at its input, since only one output is provided for each square of 2×2 values.

One solution to reach a full utilization of the hardware units in CNNs is to buffer data and share resources [33], [34]. In [33], word- and bit-serial adders were used for layers with a reduced data rate. This adaptation of the adders to the data rate reduces the hardware costs and ensures that the hardware is fully utilized. However, it does not scale further to larger networks.

Another approach proposed in [35] considers a pixel-after-pixel architecture that is tailored to the data rate of the neural network. In this approach, input data is fetched and then sorted into a buffer structure to allow the calculation of the current output of the layer. However, fetching and sorting the input data results in more overhead and requires more resources.

In this work, we present a new paradigm for the design of CNNs that consists of designing a circuit that processes data in a continuous flow. This means that the data rates and the number of components at the CNN stages are adjusted to the amount of data that is processed. Likewise, idle times are removed by multiplexing parallel streams to obtain a smaller number of streams with valid data during most of the time. This results in a smaller number of resources in the CNN and a higher utilization of them, which is close to 100%. In

This work was supported in part by Comunidad de Madrid through the call "Ayudas de Estímulo a la Investigación de Jóvenes Doctores de la Universidad Politécnica de Madrid" under Project APOYO-JOVENES-21-TL23SB-116-I4FOMC; and in part by MCIN/AEI/10.13039/501100011033 and "ESF Investing in your future" under Grant RYC2018-025384-I.

This work has been submitted to the IEEE for possible publication. Copyright may be transferred without notice, after which this version may no longer be accessible.

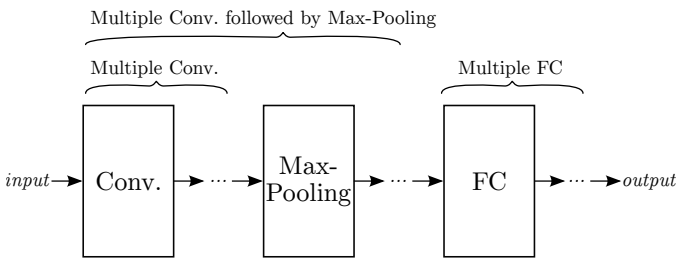


Fig. 1. The typical structure of a CNN.

the context of the state-of-the-art, this approach bridges the gap between stream and unrolled architectures by permitting designs with different degrees of parallelization. The main contributions of this paper are:

- A new paradigm for the design of continuous flow CNNs with different degrees of parallelization, covering the gap between stream and unrolled architectures.
- A thorough analysis of CNN layers from the point of view of achieving continuous flow and reducing idle times, together with the explanation on how to connect multiple layers in a CNN for continuous flow.
- An analysis of the complexity of the CNN layers depending on the parallelization.
- The proposed approach has the possibility of designing CNNs with different data rates and has high scalability.
- An automatic tool to implement continuous-flow CNNs.
- A comparison to the state-of-the-art showing significant improvements in terms of resources, mainly lookup tables (LUTs).

The paper is organized as follows. In Section II, we review the main previous concepts about CNNs. In Section III, we analyze the layers in CNNs from the point of view of continuous flow. In Section IV, we explain how to build an entire continuous-flow CNN by combining different layers. In Section V, we analyze the complexity of the CNN in terms of hardware resources. In Section VI, we compare our designs to the fully parallel implementation of common CNN models. In Section VII, we present experimental results on FPGAs and compare them to the state-of-the-art. Finally, in Section VIII, we summarize the main conclusions of the paper.

II. Background

A. Convolutional Neural Networks

The typical structure of a CNN is shown in Fig. 1. It consists of multiple blocks of convolutional layers with a max-pooling operation at the end of each block, with fully connected layers as the last layers of the model [36]. The input and output of a CNN layer consist of multiple 2D feature maps forming a 3D tensor [36]. This allows the CNN to process features in relation to their position in the tensor. The three dimensions of the tensor are thereby split into the height, width, and channel dimension. Thereby, each channel has a unique feature map. The core operation in CNNs is a sliding window operation applied on each input channel (feature map).

B. Convolutional Layers

In convolutional layers, each input value x_n arrives at a discrete time step n. The feature map has a size of $f \times f$ pixels. Pixels are processed row by row, starting from the top-left corner. According to this, the time of arrival of any input x_n is related to the rows and columns of the feature map, as

$$n = r \cdot f + c\,, \tag{1}$$

where r is the row index and c is the column index in the feature map. Using a grid of weights (kernel) with a shape of $k \times k$ the weighted sum of a sliding window from the input channel data x is calculated as

$$y_n = \sum_{j=0}^{k^2-1} w_j \cdot x_{n+f\left\lfloor \frac{j}{k} \right\rfloor + (j \bmod k)}\,. \tag{2}$$

The sum of all weighted sums of the same sliding window over all input channels, together with a bias, compose one output of an output channel (filter). Each convolutional layer can have multiple filters. Thus, each filter processes each input channel with different weights.

Intermediate accumulations of the summation are denoted as

$$z_{n,i} = \sum_{j=0}^{i} w_j \cdot x_{n+f\left\lfloor \frac{j}{k} \right\rfloor + (j \bmod k)}\,, \tag{3}$$

where $z_{n,i}$ is the result of accumulating the first products until and including index i that lead to the output y_n. Therefore, the output corresponds to

$$y_n = z_{n,k^2-1}\,. \tag{4}$$

When y_n describes the result of the operation applied on the sliding window with x_n in the upper-left corner, the last $k-1$ rows and columns of the feature map would result in a sliding window that is outside the feature map. Thus, valid outputs are defined as

$$y_n \text{ is valid} \,\forall\, n = r \cdot f + c \,|\, r, c \in \{0, ..., f-k\}\,. \tag{5}$$

C. Pooling Layers

Pooling layers also apply a sliding window operation. Each input channel is processed using a pooling operation and defines the output of an output channel. So, pooling layers always have as many input channels as they have output channels. For example, for a max-pooling layer, the sliding-window operation would be

$$y_n = \max_{j=0}^{k^2-1} \left\{ x_{n+f\left\lfloor \frac{j}{k} \right\rfloor + (j \bmod k)} \right\}\,, \tag{6}$$

where max can be replaced by other pooling operations. The most common pooling operations are max and average pooling, where the maximum value or the average is calculated for each sliding window.

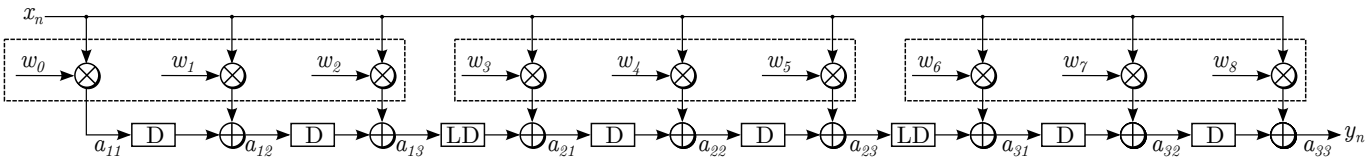


Fig. 2. KPU to calculate a 3×3 convolutional kernel.

TABLE I
TIMING OF THE KPU IN FIG. 2 FOR A 5×5 FEATURE MAP WITH A 3×3 KERNEL.

t	x_n	a_{11}	a_{13}	a_{21}	a_{23}	a_{31}	y_n
0	x_0	$z_{0,0}$	-	-	-	-	-
1	x_1	$z_{1,0}$	-	-	-	-	-
2	x_2	$z_{2,0}$	$z_{0,2}$	-	-	-	-
3	x_3	-	$z_{1,2}$	-	-	-	-
4	x_4	-	$z_{2,2}$	-	-	-	-
5	x_5	$z_{5,0}$	-	$z_{0,3}$	-	-	-
6	x_6	$z_{6,0}$	-	$z_{1,3}$	-	-	-
7	x_7	$z_{7,0}$	$z_{5,2}$	$z_{2,3}$	$z_{0,5}$	-	-
8	x_8	-	$z_{6,2}$	-	$z_{1,5}$	-	-
9	x_9	-	$z_{7,2}$	-	$z_{2,5}$	-	-
10	x_{10}	$z_{10,0}$	-	$z_{5,3}$	-	$z_{0,6}$	-
11	x_{11}	$z_{11,0}$	-	$z_{6,3}$	-	$z_{1,6}$	-
12	x_{12}	$z_{12,0}$	$z_{10,2}$	$z_{7,3}$	$z_{5,5}$	$z_{2,6}$	y_0
13	x_{13}	-	$z_{11,2}$	-	$z_{6,5}$	-	y_1
14	x_{14}	-	$z_{12,2}$	-	$z_{7,5}$	-	y_2
15	x_{15}	-	-	$z_{10,3}$	-	$z_{5,6}$	-
16	x_{16}	-	-	$z_{11,3}$	-	$z_{6,6}$	-
17	x_{17}	-	-	$z_{12,3}$	$z_{10,5}$	$z_{7,6}$	y_5
18	x_{18}	-	-	-	$z_{11,5}$	-	y_6
19	x_{19}	-	-	-	$z_{12,5}$	-	y_7
20	x_{20}	-	-	-	-	$z_{10,6}$	-
21	x_{21}	-	-	-	-	$z_{11,6}$	-
22	x_{22}	-	-	-	-	$z_{12,6}$	y_{10}
23	x_{23}	-	-	-	-	-	y_{11}
24	x_{24}	-	-	-	-	-	y_{12}

D. Fully Connected Layers

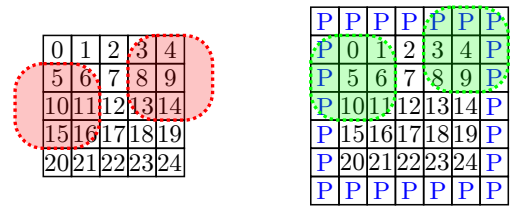
Fully connected layers do not apply a sliding window operation. Each neuron in a fully connected layer has an individual weight for each input feature. The output y_n of the n-th neuron for the input channel data x is the weighted sum of all inputs and can be described as

$$y_n = \sum_{j=0}^{f^2-1} x_j \cdot w_j \,. \tag{7}$$

For simplification, the fully connected layer operation can be described with a sliding window operation. This criterion can be satisfied by setting $k = f$. Each neuron can then be represented as a filter and the fully connected layer can be treated like a convolutional layer.

III. CONTINUOUS FLOW IN CNN LAYERS

In this section, we analyze convolutional, max-pooling, and fully connected layers from the point of view of achieving continuous flow. Once these layers are modelled, in Section IV we show how to combine the different data rates that the layers of the CNN produce to achieve continuous flow, which allows for building an entire continuous-flow CNN.



(a) Without padding. (b) With padding.

Fig. 3. Diagrams to analyze continuous flow with and without padding.

In order to model continuous flow, we consider that each input pixel at layer l consists of $d_{\ell-1}$ features and produces one output pixel consisting of d_ℓ features. The input data rate, $r_{\ell-1}$, is defined as the number of input features per clock cycle. Thereby, it is a factor that controls how fast the output data can be generated. Finally, the stride of the operation also affects the number of output features per clock cycle by only keeping $1/s^2$ of the output pixels. The output data rate, r_ℓ, can thereby be calculated as

$$r_\ell = \frac{d_\ell \cdot r_{\ell-1}}{d_{\ell-1} \cdot s^2} \,. \tag{8}$$

This equation holds for any type of layer.

A. Continuous Flow in Convolutional Layers

The component that implements the convolution in a CNN is called kernel processing unit (KPU). An example of a simple KPU that calculates a 3×3 convolution is shown in Fig. 2. It uses the transposed form [37], which allows for an efficient implementation of the multiplier and post-adder of a digital signal processor (DSP) block in common FPGAs. This KPU circuit implements a classic row buffering scheme [38] and calculates one 3×3 convolution per clock cycle by using 9 general multipliers ($\otimes$), 8 adders ($\oplus$), 6 registers (D) and 2 line buffers (LD).

Fig. 3a shows a 5×5 feature map. The timing in Table I shows how the KPU in Fig. 2 processes this feature map considering a stride of $s = 1$. Invalid values are indicated with (-). Each set of multipliers, marked with a dashed box in Fig. 2, computes one row of the sliding window. The line buffer in the preceding set delays the data by $L = f - k + 1$. The timing shows that the KPU achieves a continuous flow at the input, but does not achieve a continuous flow at the output because there are invalid outputs in between valid ones. For instance, the output at $t = 15$ and $t = 16$ in Table I are invalid and correspond to the sliding windows highlighted in Fig. 3a, which are outside the feature map.

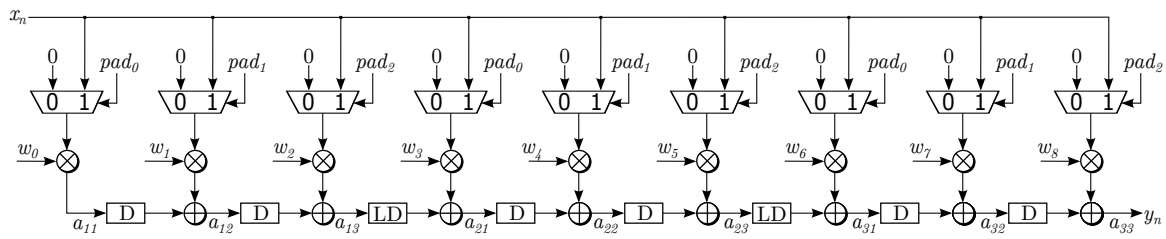


Fig. 4. A KPU with added zero-padding functionality.

B. Continuous Flow with Padding in Convolutional Layers

Padding is a well-known technique in CNNs to preserve spatial dimensions and edge information. When padding is used in the convolutional layer, it is also possible to achieve a continuous flow at the input and the output of the KPU. In this work, we focus on zero-padding because it is the most common and the most simple padding type. It can be observed that the conventional strategy of feeding zeros into the KPU at the start and end of each row breaks the continuous flow at the input by interrupting the valid inputs with explicit zero padding.

By contrast, we suggest to use implicit zero padding by setting specific multipliers of the KPU to zero, depending on the column index of the input pixel. Thus, the data order does not change and the input and output are continuous. To illustrate this idea, Fig. 3b shows the data order for a 5×5 feature map with zero-padding $p = 1$, where p represents the number of padded zeros at the sides of the feature map. Fig. 3b also highlights the same two sliding windows that were highlighted in the Fig. 3a. However, in Fig. 3b, it can be seen that both sliding windows are now contained inside the feature map, leading to continuous flow at the output.

In general, adding padding to convolutional layers leads to continuous flow at the output as long as $p = \frac{k-1}{2}$. The definition of valid outputs for convolutional layers in (5) is then adapted to

$$y_n \text{ is valid} \, \forall \, n = r \cdot f + c \,|\, r, c \in \{0, ..., f - k + 2p\} \,. \quad (9)$$

To implement implicit zero-padding in the KPU, multiplexers are added in front of the general multipliers, as shown in Fig. 4. Each of the three multiplier columns has its own select signal that depends on the current column index of the input pixel. Each multiplexer can be simplified as a bitwise AND gate. This allows for disabling the calculation of the selected columns for each input pixel by setting the weights for the column to zero. All padding select signals, pad_i, are derived from the column index c of the current input pixel. There are k select signals in total for each KPU. The equation for all pad_i signals as a function of the column is

$$pad_i(c) = \begin{cases} 0 & \text{if } c \geq f - p + i, \\ 0 & \text{if } c < p - k + i + 1, \\ 1 & \text{otherwise.} \end{cases} \quad (10)$$

In general, if the current input pixel is located at the far left or the far right of the feature map, select signals corresponding to the opposite side are set to zero. For example, with $k = 3, p = 1, f = 5$ there are three select signals, pad_0, pad_1, and pad_2. When $c = 0$, the current input pixel is at the far left, which makes pad_2 become zero because $c < p - k + i + 1$ is true, thereby padding the far-right column with zero. The opposite applies when $c = 4$, where pad_0 becomes zero because $c \geq f - p + i$ is true. With the pad_i signals, the KPU can now mask the overlapping left and right columns of invalid sliding windows, as shown in Fig. 3b.

TABLE II
TIMING OF THE KPU IN FIG. 4 APPLIED ON A 5×5 FEATURE MAP WITH PADDING $p = 1$. THE PADDING COLUMN DESCRIBES ALL THREE PADDING SELECT SIGNALS AS A TUPLE (pad_0, pad_1, pad_2).

t	x_n	Pad	a_{11}	a_{13}	a_{21}	a_{23}	a_{31}	y_n
0	0	-	$z_{0,0}$	-	-	-	-	-
1	0	-	$z_{1,0}$	-	-	-	-	-
2	0	-	$z_{2,0}$	$z_{0,2}$	-	-	-	-
3	0	-	$z_{3,0}$	$z_{1,2}$	-	-	-	-
4	0	-	$z_{4,0}$	$z_{2,2}$	-	-	-	-
5	0	-	$z_{5,0}$	$z_{3,2}$	$z_{0,3}$	-	-	-
6	x_0	(1,1,0)	$z_{6,0}$	$z_{4,2}$	$z_{1,3}$	-	-	-
7	x_1	(1,1,1)	$z_{7,0}$	$z_{5,2}$	$z_{2,3}$	$z_{0,5}$	-	-
8	x_2	(1,1,1)	$z_{8,0}$	$z_{6,2}$	$z_{3,3}$	$z_{1,5}$	-	-
9	x_3	(1,1,1)	$z_{9,0}$	$z_{7,2}$	$z_{4,3}$	$z_{2,5}$	-	-
10	x_4	(0,1,1)	$z_{10,0}$	$z_{8,2}$	$z_{5,3}$	$z_{3,5}$	$z_{0,6}$	-
11	x_5	(1,1,0)	$z_{11,0}$	$z_{9,2}$	$z_{6,3}$	$z_{4,5}$	$z_{1,6}$	-
12	x_6	(1,1,1)	$z_{12,0}$	$z_{10,2}$	$z_{7,3}$	$z_{5,5}$	$z_{2,6}$	y_0
13	x_7	(1,1,1)	$z_{13,0}$	$z_{11,2}$	$z_{8,3}$	$z_{6,5}$	$z_{3,6}$	y_1
14	x_8	(1,1,1)	$z_{14,0}$	$z_{12,2}$	$z_{9,3}$	$z_{7,5}$	$z_{4,6}$	y_2
15	x_9	(0,1,1)	$z_{15,0}$	$z_{13,2}$	$z_{10,3}$	$z_{8,5}$	$z_{5,6}$	y_3
16	x_{10}	(1,1,0)	$z_{16,0}$	$z_{14,2}$	$z_{11,3}$	$z_{9,5}$	$z_{6,6}$	y_4
17	x_{11}	(1,1,1)	$z_{17,0}$	$z_{15,2}$	$z_{12,3}$	$z_{10,5}$	$z_{7,6}$	y_5
18	x_{12}	(1,1,1)	$z_{18,0}$	$z_{16,2}$	$z_{13,3}$	$z_{11,5}$	$z_{8,6}$	y_6
19	x_{13}	(1,1,1)	$z_{19,0}$	$z_{17,2}$	$z_{14,3}$	$z_{12,5}$	$z_{9,6}$	y_7
20	x_{14}	(0,1,1)	$z_{20,0}$	$z_{18,2}$	$z_{15,3}$	$z_{13,5}$	$z_{10,6}$	y_8
21	x_{15}	(1,1,0)	$z_{21,0}$	$z_{19,2}$	$z_{16,3}$	$z_{14,5}$	$z_{11,6}$	y_9
22	x_{16}	(1,1,1)	$z_{22,0}$	$z_{20,2}$	$z_{17,3}$	$z_{15,5}$	$z_{12,6}$	y_{10}
23	x_{17}	(1,1,1)	$z_{23,0}$	$z_{21,2}$	$z_{18,3}$	$z_{16,5}$	$z_{13,6}$	y_{11}
24	x_{18}	(1,1,1)	$z_{24,0}$	$z_{22,2}$	$z_{19,3}$	$z_{17,5}$	$z_{14,6}$	y_{12}
25	x_{19}	(0,1,1)	-	$z_{23,2}$	$z_{20,3}$	$z_{18,5}$	$z_{15,6}$	y_{13}
26	x_{20}	(1,1,0)	-	$z_{24,2}$	$z_{21,3}$	$z_{19,5}$	$z_{16,6}$	y_{14}
27	x_{21}	(1,1,1)	-	-	$z_{22,3}$	$z_{20,5}$	$z_{17,6}$	y_{15}
28	x_{22}	(1,1,1)	-	-	$z_{23,3}$	$z_{21,5}$	$z_{18,6}$	y_{16}
29	x_{23}	(1,1,1)	-	-	$z_{24,3}$	$z_{22,5}$	$z_{19,6}$	y_{17}
30	x_{24}	(0,1,1)	-	-	-	$z_{23,5}$	$z_{20,6}$	y_{18}
31	0	-	$z_{0,0}$	-	-	$z_{24,5}$	$z_{21,6}$	y_{19}
32	0	-	$z_{1,0}$	-	-	-	$z_{22,6}$	y_{20}
33	0	-	$z_{2,0}$	$z_{0,2}$	-	-	$z_{23,6}$	y_{21}
34	0	-	$z_{3,0}$	$z_{1,2}$	-	-	$z_{24,6}$	y_{22}
35	0	-	$z_{4,0}$	$z_{2,2}$	-	-	-	y_{23}
36	0	-	$z_{5,0}$	$z_{3,2}$	$z_{0,3}$	-	-	y_{24}

Table II shows the timing for the KPU with a stride $s = 1$ and a padding $p = 1$ in Fig. 4 used to process a 5×5 feature map. It can be seen that this KPU achieves continuous flow both at the input and at the output. For the very first row, the

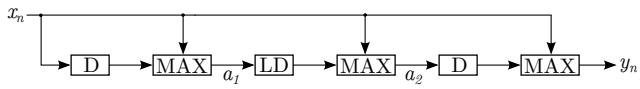


Fig. 5. An example for a 2×2 max-pooling operation.

KPU has to be fed with zeros to implement the padding of the top row. Likewise, after all inputs are received, zeros are fed into the KPU to implement the padding at the bottom row and generate the last valid outputs. This padding of the bottom row is then reused as a padding of the top for the next feature map.

To summarize, adding multiplexers to the KPU allows achieving a continuous flow for $s = 1$ by introducing zero padding where $p = \frac{k-1}{2}$.

C. Continuous Flow With Stride In Convolutional Layers

With a stride $s = 1$, the output of the KPU is considered valid when the corresponding sliding window is fully contained in the feature map. This is already included in (9). With a stride $s > 1$, outputs are skipped where the row and column index are not multiples of s. Therefore, the validity of y_n has to be adapted to

$$y_n \text{ is valid } \forall\, n = r \cdot f + c \,|\, r, c \in \{0, s, 2s, ..., f-k+2p\}. \quad (11)$$

When processing pixel after pixel, a stride $s > 1$ leads inherently to a non-continuous output. We will explain how to handle this case in Section IV-C.

D. Continuous Flow in Pooling Layers

A typical max-pooling operation is implemented using the pooling processing unit (PPU) for the example of a 2×2 kernel and a stride of $s = 2$ is shown in Fig. 5. It uses 2 registers (D), 3 max units that calculate the maximum among their two inputs (MAX), and one line buffer (LD). The circuit works similarly to the KPU, where the line buffer ensures that the correct four pixels are compared to each other accordingly to the sliding window. With the nature of pooling layers to reduce the feature map size, the stride of pooling layers is generally larger than one, which leads to a non-continuous output with a data rate of only $1/s^2$. This issue has to be addressed in the CNN implementation with the goal of achieving continuous flow and is discussed later in Section IV-D.

E. Continuous Flow in Fully Connected Layers

Fully connected layers are typically the last layers in a CNN. The three-dimensional tensor containing all feature maps is flattened into a single dimension. Each neuron in the fully connected layer is connected to all input features. With a continuous flow of input features, the first valid output of a fully connected layer can only be obtained after all inputs are received.

A fully connected layer is implemented using the fully connected unit (FCU) shown in Fig. 6. This FCU implements h neurons of the fully connected layer. The circuit holds j input pixels for h clock cycles and switches between the weights every clock cycle. The multiplexers that are used to switch between weights are implemented as read-only memory (ROM). Because the FCU processes multiple inputs in parallel, the weight tensor w must also include an index corresponding to each input. In each of the h clock cycles, the partial sum of the j weighted inputs is calculated for the corresponding neuron and added to the previous partial sum of the corresponding neuron in the buffer. After h cycles, the next j inputs are loaded. When the last j inputs are processed, the FCU produces h valid outputs for the corresponding neurons.

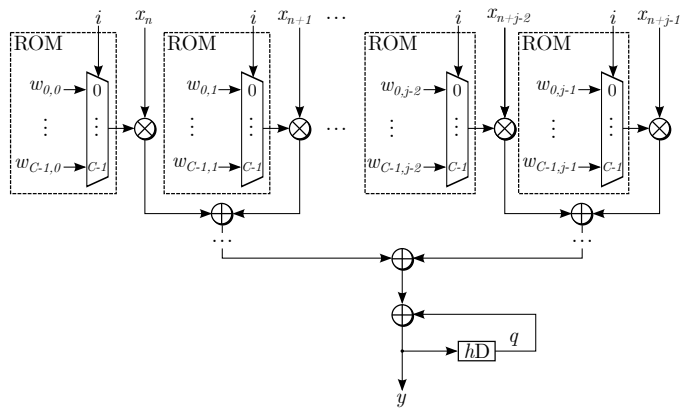


Fig. 6. An example of an FCU that calculates h different neurons with j inputs.

TABLE III
TIMING OF THE FCU IN FIG. 6 WITH $h = 5$ AND $j = 4$.

$\mathbf{t}$	$\boldsymbol{n}$	$\boldsymbol{w_{i,0}}$	$\boldsymbol{w_{i,1}}$	$\boldsymbol{w_{i,2}}$	$\boldsymbol{w_{i,3}}$	$\boldsymbol{q}$	$\boldsymbol{y}$
0	0	$w_{0,0}$	$w_{0,1}$	$w_{0,2}$	$w_{0,3}$	0	$z_{0,3}$
1	0	$w_{1,0}$	$w_{1,1}$	$w_{1,2}$	$w_{1,3}$	0	$z_{1,3}$
2	0	$w_{2,0}$	$w_{2,1}$	$w_{2,2}$	$w_{2,3}$	0	$z_{2,3}$
3	0	$w_{3,0}$	$w_{3,1}$	$w_{3,2}$	$w_{3,3}$	0	$z_{3,3}$
4	0	$w_{4,0}$	$w_{4,1}$	$w_{4,2}$	$w_{4,3}$	0	$z_{4,3}$
5	4	$w_{5,0}$	$w_{5,1}$	$w_{5,2}$	$w_{5,3}$	$z_{0,3}$	y_0
6	4	$w_{6,0}$	$w_{6,1}$	$w_{6,2}$	$w_{6,3}$	$z_{1,3}$	y_1
7	4	$w_{7,0}$	$w_{7,1}$	$w_{7,2}$	$w_{7,3}$	$z_{2,3}$	y_2
8	4	$w_{8,0}$	$w_{8,1}$	$w_{8,2}$	$w_{8,3}$	$z_{3,3}$	y_3
9	4	$w_{9,0}$	$w_{9,1}$	$w_{9,2}$	$w_{9,3}$	$z_{4,3}$	y_4

With each FCU implementing h neurons and by processing j inputs in one clock cycle, the FCUs have to switch between different weight configurations for the multipliers. The number of configurations is calculated by:

$$C = h \cdot \frac{d_{\ell-1}}{j}\,. \quad (12)$$

The timing of the FCU in Fig. 6 when calculating $h = 5$ neurons with $j = 4$ inputs for 8 input features in total is shown in Table III. With 8 input features in total, the FCU calculates all outputs after switching the inputs of the FCU twice. At each of the first 5 clock cycles, a partial sum of products for one of the 5 neurons is calculated. The signal q for the buffer hD holds the accumulated result of the current neuron. The index i is incremented each clock cycle and is reset after all 8 inputs are processed for all 5 neurons.

To implement a fully connected layer, the goal is then to find the number of inputs j_ℓ and neurons h_ℓ for the FCUs that

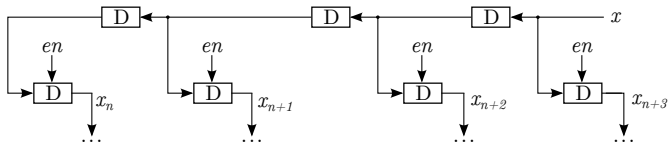


Fig. 7. The data aggregation circuit that is connected upstream of the FCU to aggregate 4 inputs.

TABLE IV
TIMING OF AN FCU WITH $h = 4$ AND $j = 4$ WITH $d_{\ell-1} = 8$.

t	x	$w_{i,0}$	$w_{i,1}$	$w_{i,2}$	$w_{i,3}$	q	y_n
0	(-,-,-,-)	-	-	-	-	0	-
1	(-,-,-,0)	-	-	-	-	0	-
2	(-,-,0,1)	-	-	-	-	0	-
3	(-,0,1,2)	-	-	-	-	0	-
4	(0,1,2,3)	$w_{0,0}$	$w_{0,1}$	$w_{0,2}$	$w_{0,3}$	0	$z_{0,3}$
5	(0,1,2,3)	$w_{1,0}$	$w_{1,1}$	$w_{1,2}$	$w_{1,3}$	0	$z_{1,3}$
6	(0,1,2,3)	$w_{2,0}$	$w_{2,1}$	$w_{2,2}$	$w_{2,3}$	0	$z_{2,3}$
7	(0,1,2,3)	$w_{3,0}$	$w_{3,1}$	$w_{3,2}$	$w_{3,3}$	0	$z_{3,3}$
8	(4,5,6,7)	$w_{4,0}$	$w_{4,1}$	$w_{4,2}$	$w_{4,3}$	$z_{0,3}$	y_0
9	(4,5,6,7)	$w_{5,0}$	$w_{5,1}$	$w_{5,2}$	$w_{5,3}$	$z_{1,3}$	y_1
10	(4,5,6,7)	$w_{6,0}$	$w_{6,1}$	$w_{6,2}$	$w_{6,3}$	$z_{2,3}$	y_2
11	(4,5,6,7)	$w_{7,0}$	$w_{7,1}$	$w_{7,2}$	$w_{7,3}$	$z_{3,3}$	y_3

satisfy the input data rate criteria. To do so, the input data rate $r_{\ell-1}$ is interpreted as

$$r_{\ell-1} = \frac{j_{\text{max}}}{h_{\text{max}}}, \tag{13}$$

where j_{max} inputs are processed over h_{max} clock cycles. With each FCU processing $j_\ell = j_{\text{max}}$ inputs, the number of neurons for each FCU, h_ℓ, can be derived from the greatest divisor of d_ℓ that is smaller or equal to h_{max}, i.e.,

$$h_\ell = \max(\{h \in \mathbb{N} \,|\, h \text{ divides } d_\ell, h \leq h_{\text{max}}\}) \,. \tag{14}$$

This allows for high utilization of all FCUs. When implementing FCUs, the adder for accumulation could have more pipeline stages than h. However, the buffer depth of the FCU has to be h, as shown in Fig. 6, to implement h different summations.

If the number of pipeline stages is larger than h, the design fails. We solve this issue by ensuring that h is big enough by introducing data aggregation. By aggregating inputs a times, the definition for j_{max} and h_{max} change to

$$r_{\ell-1} = \frac{a \cdot j_{\text{max}}}{a \cdot h_{\text{max}}} \,. \tag{15}$$

This also reduces the number of FCUs needed because each FCU calculates multiple neurons. For example, with $r_{\ell-1} = 1$, and $d_{\ell-1} = 8$, each FCU could only process one input and calculate one neuron. With $a = 4$ each FCU can process 4 inputs and calculate 4 neurons. The aggregation circuit that is connected upstream of the FCU and aggregates 4 inputs is shown in Fig. 7. The timing after aggregation is shown in Table IV. In this case, the FCU produces a valid output after 9 clock cycles. Without aggregation, the FCU would produce a valid output after 8 clock cycles. So the additional delay for aggregation is small.

IV. BUILDING AN ENTIRE CONTINUOUS-FLOW CNN

This section shows how to build a continuous-flow CNN. This involves two key design aspects. First, the data rates and hardware resources of all the layers are adjusted so that they can keep the data flow through the entire CNN. Second, idle times caused by invalid outputs of the layers are reduced so that processing units are not underutilized.

A. Running Example

To illustrate the concepts that we present in this section, a running example is used. The parameters of the running example are shown in Table V. It consists of a CNN with five layers, where C1 and C2 are convolutional layers, P1 and P2 are max-pooling layers, and F1 is a fully connected layer. The table shows the shape of the input and weight tensors, and the stride setting s for each layer. The last dimension of the input tensor represents the number of input channels. The last dimension of the weight tensor represents the number of output channels. The number of weight configurations C shows that the first two layers are fully parallel in the example with $C = 1$. The average number of valid output pixels produced per clock cycle, r_ℓ, is also included in the table for each layer. The table also shows the number of KPUs, PPUs, FCUs, and MAX units per layer. Also, the number of adders, multipliers, $2:1$ multiplexers, and registers needed to implement each layer are shown in the table. To unify the costs of different $N:1$ multiplexers, we assume that an $N:1$ multiplexer can be implemented using $N-1$ $2:1$ multiplexers. It also has to be noted that the costs do not include the ReLu implementation and the control circuits. We estimate that those costs are insignificant as control circuits are always on a per-layer basis, and the ReLu activation function is simple and a static scalar function.

The continuous-flow implementation of the running example is shown in Fig. 8. Components are grouped with curly braces. The interleaving components for a layer are grouped separately and are labeled with an "-IL" suffix. Each layer in a CNN has d_ℓ output channels by design. For example, a convolutional layer with 16 filters has 16 output channels. Each output channel has a non-continuous output when $s > 1$. All output channels together form the output data rate r_ℓ of the layer, producing r_ℓ valid outputs per clock cycle on average. By interleaving multiple input channels $d_{\ell-1}$ together, a continuous flow of data can be ensured. This requires that components can handle multiple configurations to process interleaved input data. In the following, we will analyze the flow of data in CNNs and then discuss how to interleave data, adapt our components to process interleaved data, and how to build common CNN layers based on the components to ensure a continuous flow of data.

B. Data Rates at the CNN Layers

Starting from the first layer, the data rate of each layer, r_ℓ, is calculated according to (8). The input data rate for the first layer, r_0, is equal to d_0, which is the depth of the input image.

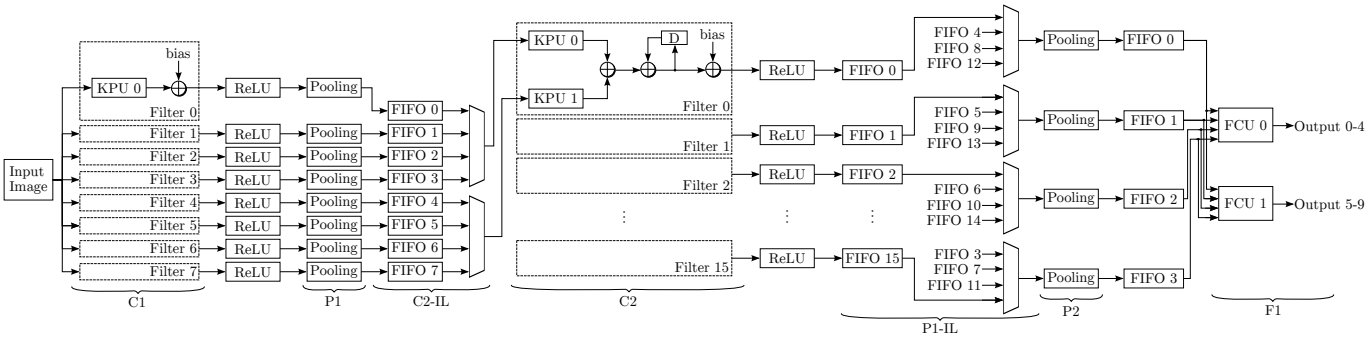


Fig. 8. Continuous-flow CNN architecture that results from the implementation of the running example.

TABLE V
THE STRUCTURE AND ANALYSIS OF THE RUNNING EXAMPLE FOR EACH LAYER.

Layer	Input (h,w,$d_{\ell\text{-}1}$)	f	k	s	p	d_ℓ	C	r_ℓ	Weights (h,w,$d_{\ell\text{-}1}$,d_ℓ)	Add.	Mul.	Reg.	2:1 MUX	MAX	KPU	FCU	PPU
C1	(24,24,1)	24	5	1	2	8	1	8	(5,5,1,8)	200	200	800	0	0	8	0	0
P1	(24,24,8)	24	2	2	0	8	1	2	-	0	0	200	0	24	0	0	8
C2	(12,12,8)	12	5	1	2	16	4	4	(5,5,8,16)	816	800	6.7k	2.4k	0	32	0	0
P2	(12,12,16)	12	3	3	0	16	4	4/9	-	0	0	416	108	32	0	0	4
F1	(256)	4	4	4	-	10	320	0.02	(256,10)	8	8	10	2.6k	0	0	2	0
Sum.		**-**	**-**	**-**	**-**	**-**	**-**	**-**	**5960**	**1024**	**1008**	**8.1k**	**5.1k**	**56**	**40**	**2**	**12**

C. Interleaving Convolutional Layers

When the data rate at the output of a layer is reduced due to a stride larger than one, we use pipeline interleaving [39] to restore continuous flow. Pipeline interleaving is based on the fact that the dataflow has idle times with non valid data. Thus, the valid data from multiple branches is interleaved into a single branch to form a flow of data with less or without idle times.

An example for an interleaved convolutional layer is shown in the running example in Fig. 8 with C2-IL showing the interleaving circuit and C2 showing the interleaved convolutional layer. The circuit implements a convolutional layer with 16 filters ($d_\ell = 16$), 8 input channels ($d_{\ell-1} = 8$) and an input data rate of $r_{\ell-1} = 2$. This leads to 8 input signals that are interleaved and form 2 output signals that provide a continuous flow of data for the connected KPUs. Note that each KPU calculates 4 different kernels and each filter consists of 2 KPUs summing up 4 kernel outputs each.

The number of KPUs per convolutional layer thereby depends on the input data rate and the number of filters according to

$$\#\text{KPUs} = \lceil r_{\ell-1} \rceil \cdot d_\ell \,. \tag{16}$$

To calculate all $d_{\ell-1}$ kernels for an output channel (filter), each KPU has to calculate $\frac{d_{\ell-1}}{\lceil r_{\ell-1} \rceil}$ different kernels. This leads to $d_{\ell-1} \cdot d_\ell$ signals to be interleaved into $\lceil r_{\ell-1} \rceil \cdot d_\ell$ signals that have their continuous-flow restored and can be fed into one KPU each. An example of a KPU that can calculate 4 different kernels is shown in Fig. 9. Each cycle, the KPU switches to the next kernel and buffers all intermediate results via pipeline interleaving. The Multiplexers used for the weights can be implemented as a ROM on the FPGA.

In case the data rate r_ℓ is reduced below 1, interleaving can not restore a continuous flow of data, stalling the connected KPUs. For example, with $r_{\ell-1} = 0.5$, the current input pixel would be present for two clock cycles and the KPU can calculate the corresponding kernel of two different filters, one after the other, by reconfiguring the KPU. The number of configurations per KPU is then calculated as

$$C = \min\left(\left\lceil \frac{d_{\ell-1}}{r_{\ell-1}} \right\rceil, d_{\ell-1} d_\ell \right). \tag{17}$$

An example is shown in Fig. 10 where a convolutional layer is implemented with $r_{\ell-1} = 0.5$, $d_{\ell-1} = 8$, and $d_\ell = 16$.

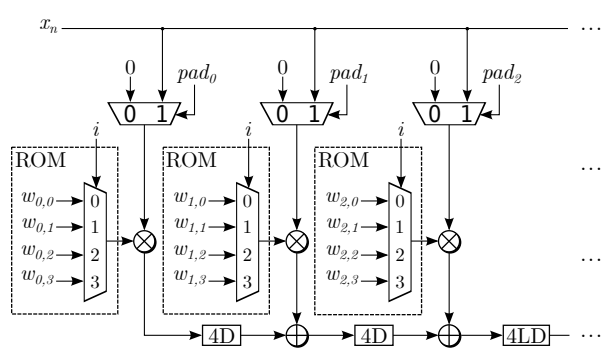


Fig. 9. An example of an adapted KPU that can process interleaved data. This KPU processes 4 different 3×3 kernels out of 4 interleaved channels.

Each KPU now calculates 8 kernels for two filters, leading to 16 configurations for the KPU. With each pixel present for two clock cycles, the KPU processes the current input pixel and calculates the current kernel of the two filters sequentially. The summation has to be interleaved to calculate two separate

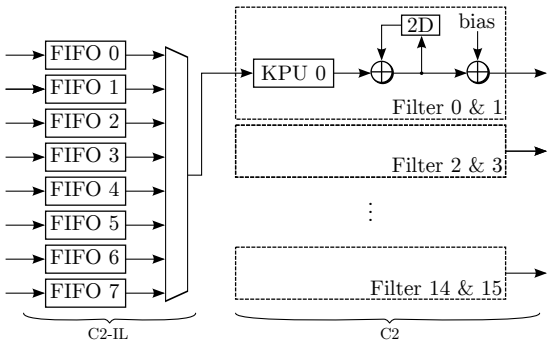


Fig. 10. The implementation of a convolutional layer where the input data rate $r_{\ell-1} = 0.5$. This allows one KPU to calculate two filters.

sums for the two filters. The number of interleaved channels I is then calculated as

$$I = \left\lceil \frac{C}{d_{\ell-1}} \right\rceil . \tag{18}$$

This adaption changes the number of KPUs to

$$\#\text{KPUs} = \lceil r_{\ell-1} \rceil \cdot \frac{d_\ell}{I} , \tag{19}$$

and allows for a more efficient implementation that reduces idle times even more by allowing the architecture to use even fewer components per layer when the data rate is low. This is especially important for the depthwise-separable convolution, which consists of fewer kernels compared to the normal convolution. This type of convolutional layer consists of a depthwise convolution followed by a pointwise convolution.

The depthwise convolution has g groups, where g is a multiple of $d_{\ell-1}$. For simplification, we assume the most prominent setting for g with $g = d_{\ell-1}$. So each output channel (group) processes a single input channel with a single kernel. Compared to the standard convolution, each channel in the depthwise convolution only depends on a single input channel. This removes the d_ℓ factor when calculating the number of KPUs needed for a depthwise convolution, leading to

$$\#\text{KPUs} = \lceil r_{\ell-1} \rceil . \tag{20}$$

Without d_ℓ as a factor, the number of configurations per KPU is then calculated as

$$C = \min \left(\left\lceil \frac{d_{\ell-1}}{r_{\ell-1}} \right\rceil , d_{\ell-1} \right) . \tag{21}$$

The pointwise convolution is a standard convolution with a 1×1 kernel. With an 1×1 KPU consisting of just a single multiplier, the output of a channel would consist of $d_{\ell-1}$ multiplications summed up over $\frac{d_{\ell-1}}{r_{\ell-1}}$ clock cycles. The pointwise convolution can thereby be implemented as a fully connected layer, where each FCU is configured following (13). An example of a depthwise-separable convolution is shown in Fig. 11. The input data rate is $r_{\ell-1} = 2$ with $d_{\ell-1} = 8$ input channels. The output data rate of the depthwise convolution stays at $r_\ell = 2$ and is fed into the pointwise convolution components. Each FCU processes $j = j_{\max} = 2$ input pixels and calculates $h = h_{\max} = 1$ output channels because $r_{\ell-1} = \frac{j_{\max}}{h_{\max}} = \frac{2}{1}$. Thereby, only 8 FCUs are needed.

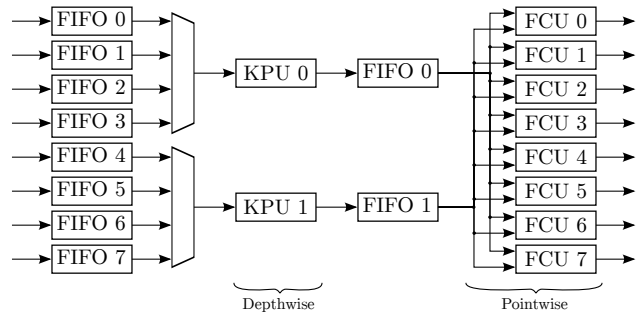


Fig. 11. An example for an interleaved depthwise-separable convolution with $d_{\ell-1} = 8$ input channels and a data rate of $r_{\ell-1} = 2$.

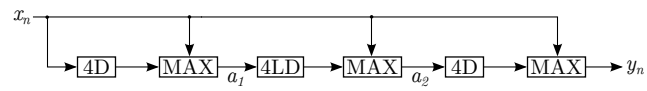


Fig. 12. An example for an interleaved 2×2 max-pooling operation with 4 channels.

D. Interleaving Pooling Layers

Interleaving pooling layers works similarly to convolutional layers, except that each output channel only processes one input channel. This leads to $d_{\ell-1}$ input signals to interleave. Note also that pooling layers do not change the number of channels, i.e., $d_{\ell-1} = d_\ell$. With each PPU processing one pixel per clock cycle, the equation for the number of PPUs breaks down to

$$\#\text{PPUs} = \lceil r_{\ell-1} \rceil . \tag{22}$$

An example for an interleaved PPU is shown in Fig. 12. The circuit implements a 2×2 max-pooling operation for 4 interleaved channels using pipeline interleaving.

E. Interleaving Fully Connected Layers

The FCU circuit in Fig. 6 produces a continuous flow at the output despite a non-continuous flow of data at the input because the FCU generates valid outputs for each neuron sequentially with the last batch of input values. Therefore, a fully connected layer does not need multiplexers or interleaving of the input. Each FCU simply processes multiple input features at once and calculates multiple output channels sequentially.

V. Complexity analysis

In the following, we will discuss the complexity of the KPU, PPU, and FCU components, how the complexity of layers changes regarding the input data rate, and how our approach compares to a fully parallel implementation of common CNN models.

A. Complexity of Data Interleaving

The cost of data interleaving is split into the cost of registers and multiplexers. Regarding multiplexers, $\frac{d_\ell}{I}$ signals are interleaved into r_ℓ signals to ensure a continuous flow of data. This leads to

$$\#\text{MUX} = \lceil r_{\ell-1} \rceil \left(\frac{d_{\ell-1}}{\lceil r_{\ell-1} \rceil I} - 1 \right) = \frac{d_{\ell-1}}{I} - \lceil r_{\ell-1} \rceil \tag{23}$$

$2:1$ multiplexers where each $N:1$ multiplexer is implemented using $N-1$ $2:1$ multiplexers. The previous layer will produce d_ℓ valid outputs every C clock cycles. Those valid outputs are spread over the last I of C clock cycles. Therefore,

$$\#\text{Registers} = \frac{d_\ell}{I} I = d_\ell \tag{24}$$

registers are needed for the FIFOs.

B. Complexity of the KPU

The parameters of a KPU component consist of the kernel size, k, the feature map size, f, and the number of configurations C. The number of adders and multipliers depends on the kernel size k because there are k^2 multiplications in the transposed form that have to be accumulated. This leads to

$$\#\text{Adders} = k^2 - 1\,, \tag{25}$$

$$\#\text{Multipliers} = k^2\,. \tag{26}$$

To store all partial sums, each KPU has $k(k-1)$ registers and $k-1$ line buffers. Each line buffer consists of $f-k+1$ registers. The KPU has to store all partial sums for each configuration. This leads to a number of registers for each KPU equal to

$$\#\text{Registers} = (k(k-1)+(k-1)(f-k+1))\,C\,. \tag{27}$$

The number of multiplexers also depends on the number of configurations and the kernel size. For all k^2 multipliers, $C-1$ $2:1$ multiplexers are needed to switch between the weights of the configurations, leading to a number of $2:1$ multiplexers equal to

$$\#\text{MUX} = k^2(C-1)\,. \tag{28}$$

C. Complexity of the Convolutional Layer Channel Accumulation

Each output channel (filter) in a convolutional layer processes all input channels. By removing the multipliers, the FCU can be repurposed for the accumulation. For each output channel, $j = \lceil \frac{\#\text{KPUs}}{d_\ell} \rceil$ inputs are accumulated per clock cycle and produce $h = \lceil \frac{C}{d_{\ell-1}} \rceil = I$ interleaved outputs over C clock cycles. Each output signal of a convolutional layer that interleaves I output channels needs one accumulator component. This leads to $\frac{d_\ell}{I}$ accumulators per convolutional layer. The cost for the accumulation is then defined by:

$$\#\text{Registers} = \frac{d_\ell}{I} I = d_\ell\,, \tag{29}$$

$$\#\text{Adders} = \frac{d_\ell}{I} \left\lceil \frac{\#\text{KPUs}}{d_\ell} \right\rceil\,. \tag{30}$$

There is a special case when $d_{\ell-1} = 1$ and $r_{\ell-1} = 1$ where no accumulation is needed. This is shown in the running example at (C1). In this case, with $d_{\ell-1} = 1$, the sum consists of a single value. Thereby, no accumulation is needed.

D. Complexity of the Bias

Some layers add an individual bias to each output channel. This leads to

$$\#\text{Adders} = \frac{d_\ell}{I} \tag{31}$$

adders for each interleaved output. Each adder has an I:1 multiplexer. This adds a multiplexing cost of

$$\#\text{MUX} = \frac{d_\ell}{I}(I-1) = d_\ell - \frac{d_\ell}{I} \tag{32}$$

$2:1$ multiplexers in total.

E. Complexity of the PPU

The PPU shares similar complexity characteristics to a KPU with

$$\#\text{MAX} = k^2 - 1\,, \tag{33}$$

and the same complexity for registers shown in (27).

F. Complexity of the FCU

The FCU component has j inputs and calculates h different sums of products. Each clock cycle, the weights are switched and a different weighted sum of all j inputs is calculated until all $d_{\ell-1}$ inputs are processed. This results in

$$\#\text{Multipliers} = j\,, \tag{34}$$

$$\#\text{MUX} = j(C-1)\,. \tag{35}$$

The weighted sum is accumulated using one adder and one register, leading to

$$\#\text{Adders} = j\,, \tag{36}$$

$$\#\text{Registers} = h\,. \tag{37}$$

G. Impact of the Data Rate on the Complexity

By interleaving data and introducing reconfigurable components to handle interleaved data, the number of components is reduced, and the utilization of the components is high. Next, we discuss these effects based on the convolutional layer, which is the most prominent layer in a CNN.

Table VI shows the resources for the implementation of a convolutional layer with different data rates. The convolutional layer has the settings $f = 28$, $k = 7$, $p = 3$, $d_{\ell-1} = 8$, $d_\ell = 16$, and the input data is not interleaved. The costs for FIFOs and data interleaving are left out because they depend on the previous layer. The first row of the table shows the resources for a fully parallel implementation where the data rate is equal to $d_{\ell-1}$. For each row below, the data rate is halved. It can be seen that the number of KPUs is proportional to the data rate, and thereby is halved for each row. By halving the number of KPUs, the number of adders and multipliers also halves. But, each KPU has to implement more and more configurations because data is interleaved more intensely to achieve a continuous flow with the decreasing data rate. This results in higher multiplexer costs. The number of registers stays the same, but they are organized differently based on the data interleaving. The last row shows an example where the data rate is so low that a continuous flow of data cannot

TABLE VI
A RESOURCE COMPARISON OF THE IMPLEMENTATION OF A CONVOLUTIONAL LAYER WITH DIFFERENT INPUT DATA RATES.

$r_{\ell-1}$	Add.	Mul.	Reg.	2:1 MUX	KPUs
8	6,272	6,272	22,288	0	128
4	3,136	3,136	22,288	3,136	64
2	1,568	1,568	22,288	4,704	32
1	784	784	22,288	5,488	16
0.5	392	392	22,288	5,880	8
0.25	196	196	22,288	6,076	4
0.125	98	98	22,288	6,174	2
0.0625	49	49	22,288	6,223	1
0.03125*	49	49	22,288	6,223	1

*The input data rate leads to a stall.

TABLE VII
A RESOURCE COMPARISON OF THE IMPLEMENTATION OF A DEPTHWISE-SEPARABLE CONVOLUTIONAL LAYER WITH DIFFERENT INPUT DATA RATES.

$r_{\ell-1}$	Add.	Mul.	Reg.	2:1 MUX	KPUs	FCUs
8	512	520	1,416	0	8	16
4	256	260	1,416	260	4	16
2	128	130	1,416	390	2	16
1	64	65	1,416	455	1	16
0.5*	56	57	1,416	463	1	8
0.25*	52	53	1,416	467	1	4

*The input data rate leads to a stall.

be restored by interleaving. Thereby, the KPUs will stall. For the depthwise-separable convolution, the same observations can be made regarding the number of adders, multipliers, and registers. The resource comparison for the depthwise-separable convolution is shown in Table VII. For this table, the same parameters were used as in Table VI.

It can be seen that the depthwise-separable convolution needs far fewer resources than the normal convolution. Our approach still achieves a significant reduction in adders and multipliers for high data rates where $r_{\ell-1} \geq 1$. Not only is the number of KPUs reduced, but also the number of inputs for each FCU. Both reductions have a significant impact on the number of adders and multipliers. In the low data rate regime ($r_{\ell-1} < 1$), the number of KPUs has already dropped to 1, and only the number of FCUs can be reduced. This results in a marginal reduction of resources. Also, the KPU is stalling because the continuous flow cannot be restored anymore.

VI. Implementation of Common CNN models

To evaluate our approach, we have implemented

a MobileNetV1 [3] model and analyzed the ResNet18 from the ResNet model family [2]. All models were trained on the ImageNet [40] dataset. The MobileNetV1 architecture heavily relies on the depthwise-separable convolution and consists of one convolutional layer, 13 depthwise-separable convolutional layers, followed by an average pooling layer and a fully connected layer. The number of filters in the convolutional layers is controlled by the factor α. The MobileNetV1 model family consists of 4 variations with $\alpha \in \{0.25, 0.5, 0.75, 1.0\}$. The ResNet18 architecture consists of residual layer blocks that are put in sequence, where the suffix 18 in the name

TABLE VIII
A COMPARISON BETWEEN A FULLY PARALLEL IMPLEMENTATION AND OUR CONTINUOUS-FLOW APPROACH FOR DIFFERENT MODELS.

Model	Param.	Imp.	Add.	Mul.	Reg.	2:1 MUX	KPUs	FCUs
Running example	6.0k	Ref.	6.0k	6.0k	8.1k	0	136	10
		Ours	1.0k	1.0k	8.1k	5.1k	40	2
MobileNet α=0.25	470k	Ref.	475k	476k	76k	0	1.5k	2.5k
		Ours	1.1k	1.1k	76k	477k	44	632
MobileNet α=0.5	1.3M	Ref.	1.3M	1.3M	151k	0	3.0k	4.0k
		Ours	3.4k	3.5k	151k	1.3M	80	2.2k
MobileNet α=0.75	2.6M	Ref.	2.6M	2.6M	226k	0	4.6k	5.5k
		Ours	7.2k	7.2k	249k	2.6M	122	1.9k
MobileNet α=1.0	4.2M	Ref.	4.3M	4.3M	300k	0	6.1k	7.0k
		Ours	12.2k	12.2k	300k	4.3M	158	5.5k
ResNet18	11.7M	Ref.	11.7M	11.7M	30M	0	1.2M	1.9k
		Ours	33.7k	33.7k	30M	11.7M	2.8k	514

stands for the number of layers in the model. In Table VIII, we compare the number of parameters and the resource costs for our proposed architecture (Ours) and for a fully parallel implementation (Ref.). Both implementations always have the same input data rate. The costs for the implementation of the activation functions are not included. Activation functions are stateless, scalar functions that do not have to be adapted to a continuous flow architecture, and thereby are not considered in the comparison. The costs for the layer control circuits, for example, the costs for counters to keep track of the current row and column of the input pixel, are also not included as they only apply on a per-layer basis.

The Mobilenet and ResNet models use a single average pooling layer before the fully connected layer. This can be implemented using a depthwise convolutional layer with constant weights. For example, the average of four pixels can be calculated with a 2×2 KPU where all weights are set to $1/4$. Another detail is the implementation of residual layers. Considering the residual layers in the ResNet model family, the merging of two layers is implemented by addition. The output pixel of the residual layer is calculated when both the pixel of the shortcut and the pixel of the sequential part of the block are calculated. We thereby assume that the layer after the merged activations has an input data rate equal to the lowest data rate of the two merged layers.

By analyzing the table, it can be observed that the number of adders and multipliers in the proposed approach is reduced substantially compared to the fully parallel implementation. The running example needs around $1/6$ of adders and multipliers, whereas the MobileNet and ResNet models reduce the number of adders and multipliers by several orders of magnitude. By contrast, reconfigurable components, multiplexing between weights and input channels is needed in the proposed approach. However, almost all multiplexers can be implemented using BRAM, because multiplexing between weights represents a read-only memory and the multiplexing cost of interleaving input channels is low as the data rate decreases. Another aspect to consider is that the number of registers does not change when our continuous-flow approach

is applied, except for the MobileNet α=0.75. This specific model has filter sizes that are not base two. This leads to a rounding in (18), rounding up the number of KPUs needed. This breaks the continuous flow and adds register costs. However, the increase in registers is small compared to the massive savings in adders and multipliers.

VII. Synthesis Results

To further evaluate our approach, Table IX compares our MobileNetV1 to previous implementations on FPGA considering the results provided by Vivado. To implement the CNN, we performed a quantization-aware training to adapt the model to the 8-bit fixed-point format used in the implementation. For the quantization-aware training, we have used the MQUAT framework[1]. Using MQUAT, the model has been trained to use 8-bit wide fixed-point formats for weights and activations. Only the output activations of the last layer have been set to 12 bits. The process to implement CNNs has been fully automated by developing a code generator. To implement all arithmetic units, FloPoCo [11] has been used except for the general multipliers, where we used the same concept shown in [35], where a single DSP implements two multiplications. The code generator also provides a bit- and clock-accurate simulator for precise accuracy determination using TensorFlow to allow fast inference under hardware conditions. The code generator performs a worst-case analysis to calculate the width of the adders in the KPUs and FCUs of the design. With the given equations, the code generator can implement layer after layer by calculating the data rate for the next layer based on (8), which then allows the software to generate the right number of FCUs/KPUs needed. Thereby, the code generator derives the number of units, configurations, etc., automatically.

TABLE IX
Comparison of MobileNetV1 implementations

	FINN **[41]***	**[35]**	**[42]**	**Ours**
Freq. (MHz)	333	211	250	**350**
LUT Util	501,363	412,354	402,200	**204,931**
FF Util	**476,316**	991,909	-	563,255
DSP Util	**106**	5,852	6,414	5,691
BRAM Util	898	1,838.5	**214**	1,702.5
URAM Util	0	0	394	0
FPGA	Alveo U280	XCVU37P	XCVU9P	XCVU37P
Power (W)	41.69	39.465	-	24.646
FPS (Inf./s)	925	4,205.5	2,637	**6,944.44**
P. Eff. (mW/Inf.)	45.07	9.38	-	**3.55**
Latency (ms)	-	0.60	379.21	**0.37**
Precision	4-bit	8-bit	8-bit	8-bit
Top-1 acc.	70.4 %	70.1%	-	**70.5%**

*FINN implementation results are from **[43]**

All power results in Table IX are obtained using the Vivado power analysis, with accurate switching activity obtained from a post-synthesis simulation. In general, our approach reaches the highest number of FPS with the lowest latency and the best energy efficiency with just 3.55 mW per inference, whereas the FINN [41] implementation shows a much more efficient use of DSP resources relative to the FPS. Compared to [35] on the same FPGA, the number of DSPs and BRAMs used in the proposed CNN is lower by a small margin. The number of LUTs and registers is almost cut in half. Furthermore, our implementation allows a far higher frequency, which reduces the latency by almost half and almost doubles the throughput. Finally, with the quantization-aware training, we reach a better top-1 accuracy for our implementation.

[1]MQUAT: www.uni-kassel.de/go/mquat

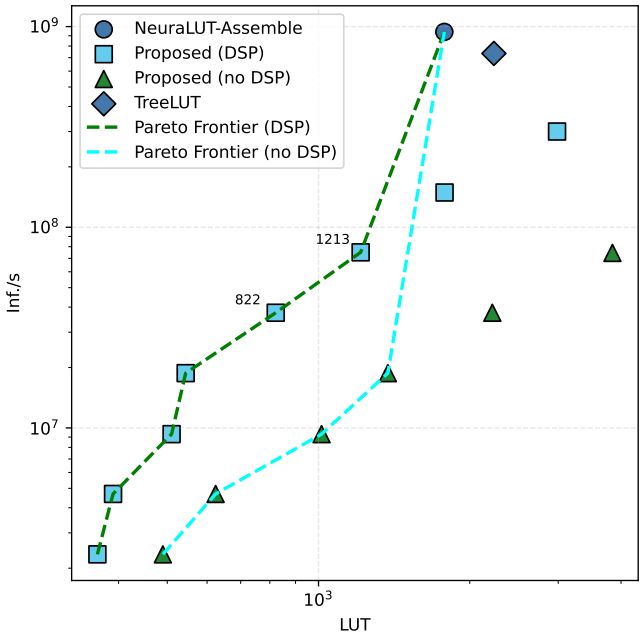


Fig. 13. Throughput compared to LUT utilization for our approach compared to the state-of-the-art parallel implementations.

We also implemented and trained a DNN with two dense layers, which have 16 neurons each, with a final third dense layer with 5 neurons. This DNN was then trained with MQUAT on the jet substructure tagging dataset JSC [49], reaching 75.2% top-1 validation accuracy. We then compared different implementations of the DNN with state-of-the-art fully parallel implementations [21], [22], [44]–[48].

Most state-of-the-art implementations do not use DSPs in their designs and are, by design, restricted to only use LUT and FF resources. To make our results more comparable to the state-of-the-art, we also created implementations that use LUT resources for the multiplications instead. The LUT multipliers were generated using the `IntMultiplier` operator from FloPoCo [50], which implements the state-of-the-art approach shown in [51], [52]. For the synthesis results, we used the `xcvu9p-flgb2104-2-i` FPGA, a Xilinx Virtex Ultrascale+, as it is the same one used in [53], [21], and [22]. Fig. 13 shows a Pareto plot that compares the throughput with LUT utilization for all our implementations compared to the state-of-the-art that reached at least a validation accuracy of 75% and use less than 5k LUTs. We also limit the maximum frequency to the maximum frequency of the FPGA board, which is 800 MHz, as higher frequencies are not possible to reach in practice. The plot shows how flexible our approach

TABLE X
COMPARISON WITH FULLY PARALLEL ARCHITECTURES ON THE JSC DATASET.

	Accuracy	r_0	**F_{max}** MHz	**LUT**	**FF**	**BRAM**	**DSP**	**Speed** MInf/s	**latency** ns
PolyLUT (JSC-XL) [22]	75%	16	235	236,541	2,775	0	0	235.0	21.0
NeuraLUT (JSC-5L) [44]	75%	16	368	92,357	4,885	0	0	368.0	14.0
NeuraLUT-Assemble [45]	76.0%	16	941	1,780	540	0	0	941	2.1
TreeLUT [46]	75.6%	16	735	2,234	347	0	0	735	2.7
DWN [47]	76.3%	16	695	6,302	4,128	0	0	695	14.4
hls4ml [48]	76.2%	16	200	63,251	4,394	0	38	200	45.0
Proposed (DSP)	75.2%	16	690	5,308	19,162	0.0	184	690.1	72.5
Proposed (DSP)	75.2%	8	600	2,984	12,215	3.5	149	300.0	93.3
Proposed (DSP)	75.2%	4	596	1,782	7,048	3.5	76	149.0	104.0
Proposed (DSP)	75.2%	2	600	1,213	4,450	3.5	38	75.0	120.0
Proposed (DSP)	75.2%	1	600	822	2,535	3.5	35	37.5	160.0
Proposed (DSP)	75.2%	1/2	600	544	1,610	4.0	19	18.8	240.0
Proposed (DSP)	75.2%	1/4	596	510	1,230	2.0	11	9.3	339.0
Proposed (DSP)	75.2%	1/8	600	390	765	2.5	5	4.7	655.0
Proposed (DSP)	75.2%	1/16	600	363	639	2.5	3	2.3	1,060.0
Proposed (no DSP)	75.2%	16	564	12,857	23,728	0.0	0	564.0	86.9
Proposed (no DSP)	75.2%	8	564	10,513	16,503	3.5	0	282.0	97.5
Proposed (no DSP)	75.2%	4	556	6,529	10,287	3.5	0	139.0	109.7
Proposed (no DSP)	75.2%	2	596	3,846	6,350	3.5	0	74.5	119.1
Proposed (no DSP)	75.2%	1	600	2,218	4,007	3.5	0	37.5	158.3
Proposed (no DSP)	75.2%	1/2	600	1,374	2,474	4.0	0	18.8	238.3
Proposed (no DSP)	75.2%	1/4	596	1,014	1,775	2.0	0	9.3	337.3
Proposed (no DSP)	75.2%	1/8	600	624	1,027	2.5	0	4.7	653.3
Proposed (no DSP)	75.2%	1/16	600	490	781	2.5	0	2.3	1,058.3

is to reach a certain throughput or resource utilization target, and how the Pareto frontier is extended for lower throughput targets.

The plot also shows that fully parallel implementations become more efficient at some point. This is expected, as our approach takes advantage of the reconfiguration of arithmetic units that become less reconfigurable with higher data rates. In Table X, the resources, throughput, validation accuracy, and latency for all state-of-the-art implementations with at least 75.0% validation accuracy are shown together with all of our implementations. It can be seen that without any DSPs and just 1374 LUTs, a DNN can be implemented that solves the JSC dataset with 75.2% accuracy.

VIII. CONCLUSIONS

In this work, we have proposed a novel approach to design continuous-flow CNN architectures. It is based on analyzing the data flow of the CNN, interleaving data based on the data rate for each layer, constructing reconfigurable components to handle the interleaved data, and thereby ensuring a high utilization close to 100%. Experimental results show that this approach saves a significant number of resources for common CNN models compared to the state-of-the-art and the 1:1 mapping of neurons to hardware units as used in most previous work using unrolled architectures. Furthermore, the experimental synthesis results demonstrate that our approach also translates to significantly lower resources compared to the state-of-the-art.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Comm. of the ACM*, Jun. 2017, pp. 84–90.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 770–778.

[3] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017.

[4] Z. Zou, K. Chen, Z. Shi, Y. Guo, and J. Ye, "Object detection in 20 years: A survey," *Proceedings of the IEEE*, vol. 111, no. 3, pp. 257–276, Mar. 2023.

[5] M. Hussain, "YOLO-v1 to YOLO-v8, the rise of YOLO and its complementary nature toward digital manufacturing and industrial defect detection," *Machines*, vol. 11, no. 7, Jul. 2023.

[6] A. Conneau, H. Schwenk, L. Barrault, and Y. Lecun, "Very deep convolutional networks for text classification," in *Conf. European Chapter Comm. Comput. Ling.*, Apr. 2017, pp. 1107–1116.

[7] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi, "Edge computing for autonomous driving: Opportunities and challenges," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1697–1716, Aug. 2019.

[8] S. Alharbi, M. Alrazgan, A. Alrashed, T. Alnomasi, R. Almojel, R. Alharbi, S. Alharbi, S. Alturki, F. Alshehri, and M. Almojil, "Automatic speech recognition: Systematic literature review," *IEEE Access*, vol. 9, pp. 131 858–131 876, Sep. 2021.

[9] C. Anthes, R. J. García-Hernández, M. Wiedemann, and D. Kranzlmüller, "State of the art of virtual reality technology," in *IEEE Aerosp. Conf.*, 2016, pp. 1–19.

[10] S. Engel, P. Boutachkov, and R. Singh, "Application of Machine Learning towards Particle Counting and Identification," in *Proc. IBIC'22*, ser. International Beam Instrumentation Conference, no. 11. JACoW Publishing, Geneva, Switzerland, 12 2022, pp. 508–511. [Online]. Available: https://jacow.org/ibic2022/papers/wep42.pdf

[11] F. de Dinechin and M. Kumm, *Application-Specific Arithmetic*. Springer, 2024.

[12] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295 – 2329, 00 2017. [Online]. Available: http://ieeexplore.ieee.org/document/8114708/

[13] N. Jouppi, C. Young, N. Patil, and D. Patterson, "Motivation for and evaluation of the first tensor processing unit," *IEEE Micro*, vol. 38, no. 3, pp. 10–19, May 2018.

[14] X. Li, G. Zhang, H. H. Huang, Z. Wang, and W. Zheng, "Performance analysis of gpu-based convolutional neural networks," in *Int. Conf. Parallel Process.*, Aug. 2016, pp. 67–76.

[15] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A framework for fast, scalable binarized neural network inference," in *Proc. ACM/SIGDA Int. Symp. FPGAs*, ser. FPGA '17, Feb. 2017, p. 65–74.

[16] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and Z. Wu, "Fast inference of deep neural networks in FPGAs for particle physics," *J. Instrum.*, vol. 13, no. 07, p. P07027, Jul. 2018.

[17] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs," in *IEEE Int. Symp. Field-Programmable Custom Comput. Machines*, May 2016, pp. 40–47.

[18] Z. Li, Z. Zhang, J. Hu, Q. Meng, X. Shi, J. Luo, H. Wang, Q. Huang, and S. Chang, "A high-performance pixel-level fully pipelined hardware accelerator for neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, pp. 1–14, Jul. 2024.

[19] Y. Umuroglu, Y. Akhauri, N. J. Fraser, and M. Blott, "LogicNets: Co-designed neural networks and circuits for extreme-throughput applications," in *Proc. Int. Conf. Field-Programmable Logic Appl.*, Aug. 2020, pp. 291–297.

[20] M. Andronic and G. A. Constantinides, "PolyLUT: Learning piecewise polynomials for ultra-low latency FPGA LUT-based inference," in *Proc. Int. Field Programmable Tech.*, Dec. 2023, pp. 60–68.

[21] ——, "NeuraLUT: Hiding neural network density in boolean synthesizable functions," in *Proc. Int. Conf. Field-Programmable Logic Appl.*, Sep. 2024.

[22] B. Lou, R. Rademacher, D. Boland, and P. H. W. Leong, "PolyLUT-Add: FPGA-based LUT inference with wide inputs," 2024, Under review.

[23] E. Wang, J. J. Davis, P. Y. K. Cheung, and G. A. Constantinides, "LUTNet: Rethinking inference in FPGA soft logic," in *IEEE Int. Symp. Field-Programmable Custom Comput. Machines*, Apr. 2019, pp. 26–34.

[24] M. Ji, Z. Al-Ars, P. Hofstee, Y. Chang, and B. Zhang, "FPQNet: Fully pipelined and quantized CNN for ultra-low latency image classification on FPGAs using OpenCAPI," *Electronics*, vol. 12, no. 19, Oct. 2023.

[25] X. Dai, Y. Chen, and M. S. Abdelfattah, "Kratos: An FPGA benchmark for unrolled DNNs with fine-grained sparsity and mixed precision," 2024, Under review.

[26] T. Habermann, J. Kühle, M. Kumm, and A. Volkova, "Hardware-aware quantization for multiplierless neural network controllers," in *Proc. IEEE Asia-Pacific Conf. Circuits Syst.*, Nov. 2022, pp. 541–545.

[27] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Haq: Hardware-aware automated quantization with mixed precision," in *IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2019, pp. 8604–8612.

[28] S. S. Miriyala, P. K. Suhas, U. Tiwari, and V. N. Rajendiran, "Mixed precision neural quantization with multi-objective bayesian optimization for on-device deployment," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, Apr. 2024, pp. 6260–6264.

[29] Z. Dong, Y. Gao, Q. Huang, J. Wawrzynek, H. K. So, and K. Keutzer, "Hao: Hardware-aware neural architecture optimization for efficient inference," in *IEEE Int. Symp. Field-Programmable Custom Comput. Machines*, May 2021, pp. 50–59.

[30] S. Gao, Z. Zhang, Y. Zhang, F. Huang, and H. Huang, "Structural alignment for network pruning through partial regularization," in *IEEE/CVF Int. Conf. Comput. Vis.*, Oct. 2023, pp. 17 356–17 366.

[31] S. Gao, F. Huang, W. Cai, and H. Huang, "Network pruning via performance maximization," in *IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2021, pp. 9266–9276.

[32] M. B. Hossain, N. Gong, and M. Shaban, "Computational complexity reduction techniques for deep neural networks: A survey," in *IEEE Int. Conf. Artificial Intell. Blockchain Internet Things*, Sep. 2023, pp. 1–6.

[33] S. Tridgell, M. Kumm, M. Hardieck, D. Boland, D. Moss, P. Zipf, and P. H. W. Leong, "Unrolling ternary neural networks," *ACM Trans. Reconf. Tech. Syst.*, vol. 12, no. 4, Dec. 2019.

[34] Z. Wang, "A digits-recognition convolutional neural network on FPGA," Master's thesis, Dept. of Electrical Engineering, Linköping University, Oct. 2019.

[35] Z. Li, Z. Zhang, J. Hu, Q. Meng, X. Shi, J. Luo, H. Wang, Q. Huang, and S. Chang, "A high-performance pixel-level fully pipelined hardware accelerator for neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–14, 2024.

[36] N. Aloysius and M. Geetha, "A review on deep convolutional neural networks," in *Int. Conf. Comm. Signal Process.*, Apr. 2017, pp. 0588–0592.

[37] M. Kumm, D. Fanghänel, K. Möller, P. Zipf, and U. Meyer-Baese, "FIR filter optimization for video processing on FPGAs," *EURASIP J. Adv. Signal Process.*, vol. 2013, no. 1, p. 111, Dec. 2013.

[38] D. G. Bailey, *Design for Embedded Image Processing on FPGAs*. Wiley-IEEE Press, 2011.

[39] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*. John Wiley & Sons, 1999.

[40] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[41] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser, and K. Vissers, "Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, no. 3, Dec. 2018. [Online]. Available: https://doi.org/10.1145/3242897

[42] W. Huang, C. Luo, B. Zhao, H. Jiao, and Y. Huang, "Hcg: Streaming dcnn accelerator with a hybrid computational granularity scheme on fpga," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 36, no. 10, pp. 18 681–18 695, 2025.

[43] Y. Xie, Z. Li, D. Diaconu, S. Handagala, M. Leeser, and X. Lin, "Lutmul: Exceed conventional fpga roofline limit by lut-based efficient multiplication for neural network inference," in *Proceedings of the 30th Asia and South Pacific Design Automation Conference*, ser. ASPDAC '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 713–719.

[44] M. Andronic and G. A. Constantinides, "Neuralut: Hiding neural network density in boolean synthesizable functions," in *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*, 2024, pp. 140–148.

[45] ——, "NeuraLUT-Assemble: Hardware-Aware Assembling of Sub-Neural Networks for Efficient LUT Inference," in *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2025, pp. 208–216, doi: 10.1109/FCCM62733.2025.00077.

[46] A. Khataei and K. Bazargan, "Treelut: An efficient alternative to deep neural networks for inference acceleration using gradient boosted decision trees," in *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 14–24. [Online]. Available: https://doi.org/10.1145/3706628.3708877

[47] A. T. L. Bacellar, Z. Susskind, M. Breternitz Jr., E. John, L. K. John, P. M. V. Lima, and F. M. G. França, "Differentiable weightless neural networks," in *Proceedings of the 41st International Conference on Machine Learning*, ser. ICML'24. JMLR.org, 2024.

[48] F. Fahim, B. Hawks, C. Herwig, J. Hirschauer, S. Jindariani, N. Tran, L. P. Carloni, G. D. Guglielmo, P. Harris, J. Krupa, D. Rankin, M. B. Valentin, J. Hester, Y. Luo, J. Mamish, S. Orgrenci-Memik, T. Aarrestad, H. Javed, V. Loncar, M. Pierini, A. A. Pol, S. Summers, J. Duarte, S. Hauck, S.-C. Hsu, J. Ngadiuba, M. Liu, D. Hoang, E. Kreinar, and Z. Wu, "hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices," 2021. [Online]. Available: https://arxiv.org/abs/2103.05579

[49] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and Z. Wu, "Fast inference of deep neural networks in fpgas for particle physics," *Journal of Instrumentation*, vol. 13, no. 07, p. P07027, jul 2018. [Online]. Available: https://doi.org/10.1088/1748-0221/13/07/P07027

[50] F. de Dinechin, "Reflections on 10 years of FloPoCo," in *26th IEEE Symposium of Computer Arithmetic (ARITH-26)*, Jun. 2019.

[51] A. Böttcher and M. Kumm, "Multiplier design addressing area-delay trade-offs by using dsp and logic resources on fpgas," in *2024 IEEE 35th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2024, pp. 217–225.

[52] ——, "Small logic-based multipliers with incomplete sub-multipliers for fpgas," in *2024 IEEE 31st Symposium on Computer Arithmetic (ARITH)*, 2024, pp. 124–131.

[53] O. Weng, M. Andronic, D. Zuberi, J. Chen, C. Geniesse, G. A. Constantinides, N. Tran, N. J. Fraser, J. M. Duarte, and R. Kastner, "Greater than the sum of its luts: Scaling up lut-based neural networks with amigolut," in *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 25–35. [Online]. Available: https://doi.org/10.1145/3706628.3708874