NN2FPGA: Optimizing CNN Inference on FPGAs With Binary Integer Programming

NN2FPGA: Optimizing CNN inference on FPGAs with Binary Integer Programming

Roberto Bosio*, Filippo Minnella*, Teodoro Urso*, Mario R. Casu*, Luciano Lavagno*, Mihai T. Lazarescu* and Paolo Pasini*
*DET - Department of Electronics and Telecommunications, *Politecnico di Torino*, Turin, IT

***Abstract*—Skip connections have emerged as a key component of modern convolutional neural networks (CNNs) for computer vision tasks, allowing for the creation of more accurate and deeper models by addressing the vanishing gradient problem. However, the existing implementations of field-programmable gate array (FPGA)-based accelerators for ResNets and MobileNetV2 often experience decreased performance and increased computational latency due to the implementation of skip blocks. This paper presents a novel framework for developing deep learning models on FPGAs that focuses on skip connections, with a unique approach to reduce buffering overhead. This results in a more efficient utilization of resources in the implementation of the skip layer. The *nn2fpga* compiler follows a thorough set of high-level synthesis (HLS) design principles and optimization strategies, exploiting in novel ways standard techniques to effectively map skip connection-based networks into static dataflow accelerators. To maximize throughput and efficiently use the available resources, our compiler employs a fast and effective design space exploration method based on a binary integer programming model which accurately assigns FPGA resources to the network layers, to maximize global throughput under resource constraints and then minimize resources for the achieved maximum throughput. Experimental results on the CIFAR-10 and ImageNet datasets demonstrate substantial gains in throughput (3× to 7× on past HLS-based work) for ResNet8, ResNet20, and MobileNetV2 models deployed on various Xilinx FPGA boards. Notably, MobileNetV2 deployed on the ZCU102 achieves a throughput of 2115 FPS, representing even a 10% speedup over a state-of-the-art highly optimized manual RTL implementation, showing that HLS can actually improve over manual design, thanks to the faster exploration of the design space.**

***Index Terms*—Dataflow architecture, FPGA acceleration, graph optimization, high-level synthesis, pipelining, quantization, residual neural networks, codesign.**

I. Introduction

Convolutional neural networks (CNNs) have consistently achieved state-of-the-art results across various tasks, such as computer vision and speech recognition [1]. Their efficacy stems from the heightened accuracy and efficiency offered by convolutional layers compared to earlier methods. These layers demand less memory bandwidth than fully connected (FC) layers, thus enhancing computational efficiency [2]. The hardware choice for implementing convolutional layers profoundly impacts their applicability. CPUs are versatile and easy to program, but their architecture makes them relatively inefficient. On the other hand, GPUs excel in handling massive parallelism. This feature aligns well with the inherently parallel nature of CNNs, although at the cost of higher energy consumption [3].

Application-specific integrated circuits (ASICs) and field-programmable gate arrays (FPGAs) offer different trade-offs between cost and flexibility for algorithm acceleration [4]. While FPGAs are less performant and energy-efficient than ASICs due to their reprogrammability, they significantly reduce design costs and can be more easily customized for specific applications.

Neural networks (NNs) optimized for embedded applications [5], [6] are designed to run efficiently on devices with limited processing power, memory, and energy. They excel particularly on small datasets, such as CIFAR-10 [7] and MNIST [8], commonly employed in real-time scenarios demanding quick response and minimal latency.

Residual neural networks (ResNets) [9] use residual blocks (see Fig. 1) to mitigate the vanishing gradient problem for deep networks through *skip connections*. They allow intermediate feature maps to be reprocessed at different points in the network computation, increasing accuracy. However, state-of-the-art implementations of *skip connections* require significant on-chip buffering resources, reducing the benefits of streaming-based FPGA implementations. Recent works have been focusing on optimizing and shrinking the residual structure of NNs [10] alongside exploiting new quantization techniques to enhance hardware design efficiency [11].

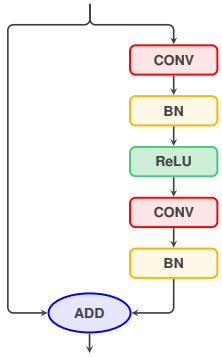


Fig. 1: Basic residual block with two convolutional layers in the main branch and a skip connection.

Deep networks have numerous parameters and need extensive quantization to reduce their size in order to fit into the FPGA on-chip memory [12]. For this reason, widely used tools such as FINN [13] focus on low-bit quantization (such as 1-bit [14] or 2-bit [15]), at the expense of suboptimal resource utilization for higher-bit quantization. However, low-bit quantizations degrade NNs accuracy and may not be suitable for accurate inference on complex problems [16].

To address these challenges, a novel framework called *nn2fpga*[1] is proposed for implementing convolutional neural networks (CNNs) on FPGAs, with a specific emphasis on efficiently integrating residual block architectures into dataflow implementations. The main contributions of this study include:

- Introduction of an optimized architecture for CNNs that efficiently supports residual networks and uses a novel method to reduce buffering resources, allowing on-chip storage of the parameters and activations, and thus significantly improving the overall performance.
- Development of an optimization strategy based on binary integer programming (BIP) to efficiently allocate resources to each layer of the model, while maximizing the overall throughput.
- Integration of all the steps in an automatic high-level synthesis (HLS) code generation flow, from the quantized model to the FPGA bitstream using Vitis HLS [17].

The validation of the accelerator architectures and the *nn2fpga* implementation flow is conducted by deploying ResNet8 and ResNet20 for CIFAR-10, as well as MobilenetV2 for ImageNet, on several Xilinx embedded FPGA boards, demonstrating the advantages of the proposed solution.

The rest of the paper is organized as follows. Section II presents the background and motivation behind this work. Section III describes in depth the *nn2fpga* compiler, along with the accelerator architecture design, with a special focus on skip connection management. Section IV presents the experimental setup and discusses the results. Section V concludes the paper.

II. Related Work

The field of FPGA-based acceleration for deep neural networks has gained significant interest for its potential to deliver high-performance and energy-efficient inference. Several approaches and architectures have been proposed in the literature to address this challenge [18].

In systolic array overlay-based architectures, each processing element (PE) is a single instruction multiple data (SIMD) vector accumulation module, receiving activation inputs and weights from the horizontally and vertically adjacent PEs at each cycle. Pipelined groups of PEs with short local communication and regular architectures can achieve high clock frequencies and efficient global data transfers [19]. The overlay architecture [20] sequentially performs the computation of the convolution layers over a systolic array. However, despite its flexibility, it suffers from high latency due to frequent transfers between external memory and on-chip memory.

An alternative approach involves implementing a custom dataflow architecture where each layer is associated with a dedicated compute unit. This structure can be pipelined, and activations and weights can be stored in on-chip memory, reducing latency and increasing throughput. The main limitation of this architecture is the number of digital signal processor blocks (DSPs) and lookup tables (LUTs) required to implement the convolutional layers, as well as the size of on-chip buffers for weight storage [21]. Since with static dataflow networks streaming tasks have well-defined pipelining production rates, a customized approach can lead to optimized processing, resulting in improved performance and resource-saving.

Widely recognized as one of the leading frameworks for deploying deep neural networks (DNNs) on FPGAs, Xilinx Vitis AI [22] provides a comprehensive set of tools specifically designed for optimizing and deploying DNNs on Xilinx FPGAs. With support for popular frameworks such as TensorFlow [23], PyTorch [24], and Caffe [25], it incorporates various optimization techniques such as pruning, quantization, and kernel fusion to improve performance and reduce memory consumption. The deep learning processor unit (DPU) is the accelerator core used in Vitis AI and consists of several key modules, including a high-performance scheduler, a hybrid computing array, an instruction fetch unit, and a global memory pool [26]. The DPU is responsible for executing the microcode of the specified DNN model, known as the *xmodel*. Vitis AI uses an overlay-based architecture where model weights and biases are stored in double data rate (DDR) memory and cached in the on-chip weight buffer during inference. Input and output data of the PE array are also cached on-chip. This architecture generally scales with the dimension of the DNN, but may have higher resource utilization and less performance compared to custom dataflow accelerators, due to its general-purpose nature and the overhead associated with off-chip memory accesses.

Another widely used tool is FINN [13], an open-source framework developed by Xilinx that allows the generation of highly optimized DNNs for FPGA acceleration, with emphasis on dataflow-style architectures. FINN uses HLS to convert trained DNN models into hardware intellectual property (IP) blocks that can be easily integrated into FPGA-based systems. While FINN offers significant customization capabilities, it is primarily designed for low-bitwidth quantization schemes, such as binarized networks. Achieving high performance with FINN often leads to lower accuracy and/or higher resources, particularly when using the 8-bit quantization that has been shown to be the best compromise between resources and accuracy [27] [28].

In the literature, [29] evaluates the accuracy, power, throughput, and design time of three different CNNs implemented on FPGAs and compares these metrics to their GPU equivalents. A comparison was also made between a custom implementation of two DNNs using System Verilog and an implementation using the Xilinx tools FINN and Vitis AI [30]. In addition, [31] reports a comparison between FINN and Vitis AI using a widely used set of ResNet model configurations.

Recent advances in HLS for artificial intelligence applications have leveraged sophisticated compilation frameworks to optimize the performance and efficiency of hardware designs. To support optimization, [32] provides a flexible and extensible compiler infrastructure based on MLIR (Multi-Level Intermediate Representation) that facilitates fine-grained transformations and analyses through its hierarchical approach. Built on top of MLIR, ScaleHLS further extends these capabilities by offering domain-specific optimizations specifically tailored for HLS. [33] demonstrates how ScaleHLS leverages the multi-level structure of MLIR to achieve significant improvements for AI accelerators in traditional HLS tools in various benchmarks.

[1]The source code of the framework is available at: https://github.com/minnellf/NN2FPGA

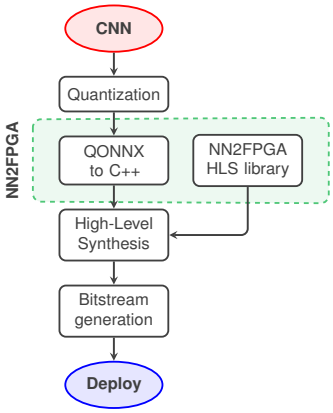


Fig. 2: Implementation flow

III. Methodology

Figure 2 illustrates the overall flow used to convert a generic NN model into an FPGA static dataflow accelerator. This process entails the following steps:

- Utilize Brevitas [34] for NN quantization, generating the network graph in QONNX format as detailed in Section III-A. This format provides a comprehensive description of the network, including layer type, input and output quantization, and layer interconnections.
- Generate optimized C++ code from the QONNX file. The *nn2fpga* compiler described in this paper applies transformation steps to optimize the network graph for inference and manages resource allocation for each network layer. Detailed explanations are provided in Section III-B, Section III-E and Section III-F.
- Obtain the synthesizable register-transfer level (RTL) code through Vitis HLS, using the network-specific code generated at the previous step and a set of network-independent C++ functions, each implementing a network layer using the dataflow architecture. Section III-C and Section III-D describe the *nn2fpga* library details.
- Integrate the RTL code as a block into Vivado and generate the bitstream for programming the FPGA.

A. Integer quantization

Quantization plays a crucial role in enabling the deployment of deep learning models in real-world applications with limited computational resources. It involves reducing the precision of weights and activations from floating-point to fixed-point numbers, typically represented as integers.

There are two primary methods for quantizing a neural network: post-training quantization (PTQ) and quantization-aware training (QAT) [35]. PTQ consists in quantizing the weights and activations of a pre-trained model to lower precision, offering speed and cost-effectiveness but potentially sacrificing accuracy. QAT involves employing floating-point calculations and model quantization through clamping and rounding methods. During back-propagation, inputs and weights are dequantized to enhance convergence and accuracy. However, during loss evaluation, quantization is applied to match the results of the hardware implementation.

The quantization $Q(\cdot)$ of a floating point value b into an integer value on bw bits is defined as:

$$Q(b) = \text{clip}(\text{round}(b \cdot 2^{bw-s}), a_{\min}, a_{\max}) \cdot 2^s \quad s \in \mathbb{N} \tag{1}$$

$$a_{\min} = act_{\min}(s) = \begin{cases} 0 & \text{if unsigned} \\ -2^{bw-1-s} & \text{if signed} \end{cases} \tag{2}$$

$$a_{\max} = act_{\max}(s) = \begin{cases} 2^{bw-s} - 1 & \text{if unsigned} \\ 2^{bw-1-s} & \text{if signed} \end{cases} \tag{3}$$

where 2^s is the scaling factor, $a_{\min}$ is the lower clipping bound and $a_{\max}$ is the higher one. Since *nn2fpga* uses symmetric quantization, all zero points are equal to zero and omitted in the expressions above. Furthermore, scaling factors are restricted to powers of two, to map alignment operations between different quantizations into hardware-friendly bit shifts [36].

After training, another optimization applied to the network is the batch normalization folding, which merges the batch normalization with the convolutional layer [37]. This is possible since at inference time batch normalization is just an affine transformation and thus can be merged directly with the parameters of the convolutional layer.

Brevitas is an open-source PyTorch extension that provides quantization for deep learning models. Thanks to its flexibility, it has become a popular tool for quantizing NNs. In this work, we use *Brevitas* to quantize the NN model with the parameters described above, generating the resulting network graph in the QONNX format [38], [39]. Both *Brevitas* and Vitis HLS are products of AMD Xilinx, leading to numerous synergies and seamless interoperability of data formats between them. Nonetheless, *nn2fpga* remains independent of the quantization tool used during training, provided it adheres to the integer data format with a scaling factor in powers of 2. Table I compares integer quantization using power-of-two scaling factors versus floating point scaling factors. This comparison is based on Torchvision pre-trained models and demonstrates that the choice of either method results in minimal accuracy loss with respect to the floating point baseline.

B. Accelerator architecture

Given a QONNX description, the *nn2fpga* compiler generates a static dataflow accelerator written in synthesizable C++ code. The accelerator is a directed acyclic graph of dataflow tasks communicating through data streams, as shown in Fig. 3. The tasks can be divided into two main categories:

- *Computation tasks*: one for each convolution or pooling node to implement the layer computations (see Section III-C).
- *Window buffer tasks*: multiple tasks for each convolution or pooling node, to buffer and provide the input data to the computation tasks (see Section III-D).

All tasks are part of a reusable, templated C++ library that can adapt to different activation and weight tensor dimensions,

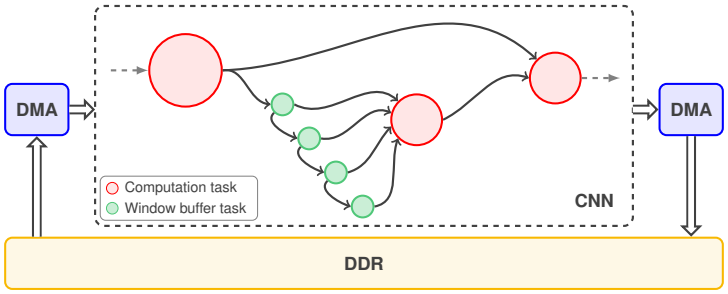


Fig. 3: Accelerator architecture with direct memory access (DMA) blocks for memory transfers (blue boxes) and concurrent tasks communicating through data streams.

TABLE I: Comparison of integer PTQ using power-of-two versus floating point scaling factors on ImageNet [34].

Model	Quant.	Weight bit	Act. bit	Scaling	Accuracy
MobilenetV2	Floating point baseline				71.90%
	INT	8	8	FP32	-0.83%
	INT	8	8	Po2	-0.77%
ResNet18	Floating point baseline				69.76%
	INT	8	8	FP32	-0.06%
	INT	8	8	Po2	-0.14%
ResNet50	Floating point baseline				76.13%
	INT	8	8	FP32	-0.44%
	INT	8	8	Po2	-0.23%
ResNet101	Floating point baseline				77.37%
	INT	8	8	FP32	-0.49%
	INT	8	8	Po2	-0.34%

data types, and computational parallelism. Each task is composed of a main loop that performs multiple operations.

To increase the accelerator throughput, pipelining is enabled at two different levels:

- Inter-task: concurrent layer execution is achieved using the Vitis HLS `dataflow` pragma, which enables processes to start as soon as input data is ready. The throughput of the slowest task determines the overall accelerator throughput and is balanced by the design space exploration (DSE) algorithm described in Section III-F.
- Intra-task: concurrent operation execution is used to improve and balance task throughput. All tasks are internally pipelined achieving an initiation interval (II) equal to 1, i.e., a new iteration of the loop is initiated every clock cycle. Furthermore, tasks with high computational workload, like convolutions, are partially unrolled by factors chosen during the DSE.

Network inputs and outputs are managed as continuous streams of data, with DMA blocks handling the reading of input activations and the writing of the results from and to off-chip memory. Parameters of the network are also streamed from off-chip memory to the computation tasks, but only at start-up, after reset.

Accelerators built with the *nn2fpga* framework have a data-driven execution approach, processing frames sequentially and as a continuous stream. This is achieved in Vitis HLS by using the `ap_ctrl_none` pragma for the computation and window buffer tasks. Inference begins as soon as the DMA attached to the input port of the top-level interface is enabled to download input images.

C. Convolution computation task

Convolution stands out as the primary computation task in CNNs, utilizing a significant portion of available resources. Each convolution receives a window of input activations from a window buffer task, while reading the needed filter windows from the local on-chip memory. The C++ HLS code outlined in Alg. 1 shows the convolution process and examples of how the computation pipeline receives input data and computes the partial results. Fig. 4a provides a visual representation of the convolution architecture.

TABLE II: Symbol definitions.

Symbol	Description
ich	Input tensor channels
ih	Input tensor height
iw	Input tensor width
och	Output tensor channels
oh	Output tensor height
ow	Output tensor width
fh	Filter tensor height
fw	Filter tensor width
s	Convolution stride

The innermost loops are completely unrolled over the filter dimensions and partially over the input channels, output channels and output width dimensions. In a typical convolution, *nn2fpga* exploits three types of parallelization:

- Over input channels (ich^{par}): processing multiple 2D windows of the same pixel simultaneously, along the depth of the tensor.
- Over output channels (och^{par}): processing multiple filters concurrently.
- Over output width (ow^{par}): processing multiple output activations in parallel.

The unroll factors denoted by "*par*" imply fully data-parallel execution and are closely tied to the number of PEs utilized. The relation between unroll factors and resources is analyzed in Section III-F, as they are leveraged by DSE to achieve low latencies and high global throughput.

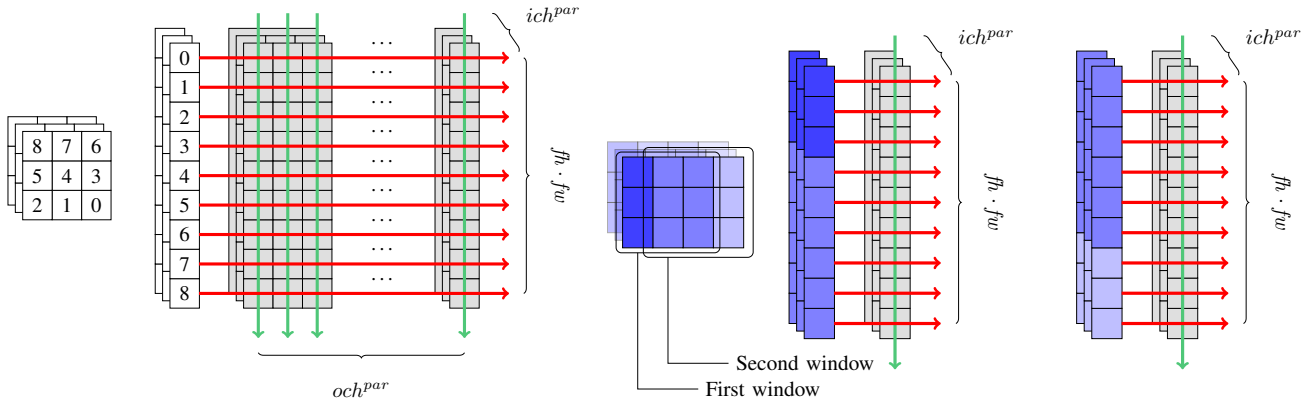


Fig. 4: Convolution architectures. Activations follow horizontal lines, while accumulators follow vertical ones. (a) depicts a 3×3 standard convolution. (b) depicts a 3×3 depthwise convolution with two overlapping windows convolved simultaneously.

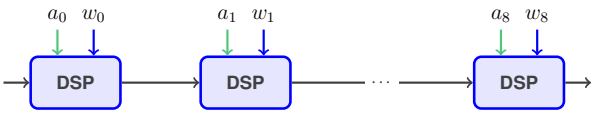


Fig. 5: DSP chain propagating the accumulator.

Depthwise convolutions are a special case of the generic convolution, in which a different convolutional filter is applied to each channel, separately. From the perspective of code 1, depthwise convolution can be reinterpreted as a generic convolution with just one filter and without the sum of partial results across input channels. Following the *nn2fpga* notation, this translates into having *och*, and by consequence och^{par}, equal to 1. Fig. 4b depicts the resulting depthwise convolution architecture.

To efficiently utilize FPGA resources, *nn2fpga* adopts two DSP optimizations. The first optimization exploits DSP packing, which minimizes the hardware resources required for processing quantized data. This technique allows for multiple operations to be executed on a single DSP block. The configurations used in this study are based on the methods detailed in [40] and [41], which cover the computation of packed data with 8-bit and 4-bit precision. With the former, it is possible to perform two 8-bit multiply-adds (MADDs) operations on a single DSP, whereas with the latter, up to 4 MADDs can be packed into one DSP. The packing of multiple operations in a single DSP is directly implemented in the *nn2fpga* library through custom HLS code.

The second optimization is DSP chaining, which consists of connecting multiple DSPs in long chains propagating the same accumulation, as shown in Fig. 5. Although this technique increases pipeline depth, it effectively reduces resource usage by leveraging the internal adders within the DSPs. The Vitis HLS pragma `EXPRESSION_BALANCE OFF` can force the tool to create longer chains of operation.

D. Window buffer tasks

The window buffer is responsible for providing the activation windows to the computation tasks with which it is connected. These processes must store and move enough data to feed the computation task pipeline. Activations are produced following a depth-first order by the convolution that computes the input tensor, while the input window is spread across a single channel and multiple lines. Consequently, it is necessary to store all the lines needed to form an input window and thus each window buffer, also known as line buffer in the literature, should be appropriately sized to accommodate the required activations.

Considering a generic computation task i, the portion of the input tensor B_i that the buffer must retain to create an input window is:

$$B_i = [(fh_i - 1) \cdot iw_i + fw_i - 1] \cdot ich_i. \tag{4}$$

The size of the buffer is constant over time, since whenever an activation is no longer needed by any of the associated windows, it is shifted out [42].

As discussed in the previous section, computation tasks require input windows of activations. However, the data needed to form the input window is not contiguous and cannot be directly read from the buffer, because it is stored sequentially in a first in first out (FIFO) buffer with only one read port available. To provide the necessary bandwidth, the FIFO must be partitioned into $fh_i \cdot fw_i$ segments, connected sequentially.

The size of each FIFO corresponds to the distance, measured in number of activations, between consecutive positions within the same input window, considering that the tensor is processed in depth-first order. The distance between two activations within the same row of the input window is determined by the number of channels ich_i in the tensor. Similarly, the gap between two activations in separate rows of the input window is equal to one row ($ich_i \cdot iw_i$) of activations, minus the filter width fw_i.

Since each FIFO represents a position in the input window, the structure of the window buffer depends on the unroll factor ow^{par}, as it directly impacts the number of activation windows required in parallel. With the unroll factor considered, the number of activations required to form the input window becomes $(fw + ow^{par} - 1) \cdot fh$, thus needing a partitioned window buffer to increase memory access parallelism. Fig. 6 offers a visual representation of how the input tensor is partially

```
template <typename a_in_t, typename a_out_t,
          typename w_t, typename b_t, size_t ich,
          size_t iw, size_t ih, size_t och,
          size_t ow, size_t oh, size_t fh,
          size_t fw, size_t ich_par, size_t och_par,
          size_t ow_par, ...>
void conv(hls::stream<a_in_struct_t> &in_stream,
          hls::stream<a_out_struct_t> &out_stream,
          hls::stream<w_struct_t> &weights_stream,
          hls::stream<b_struct_t> &bias_stream)
{
  t_acc s_acc_buff[och / och_par][och_par * ow_par];
#pragma HLS array_partition variable=s_acc_buff dim=3
#pragma HLS expression_balance off

  for (auto wh_i=0; wh_i < (ow*oh)/ow_par; wh_i++){
    for (auto ich_i=0; ich_i < ich; ich_i+=ich_par){
      for (auto och_i=0; och_i < och; och_i+=och_par){
#pragma HLS pipeline II=1 style=stp

        w_t weights[och_par][ich_par][fh][fw];
        b_t bias[och_par];
        a_in_t input[ich_par][fh][fw + (ow_par - 1)];
        a_out_t output[och_par];

        weights = weights_stream.read();
        if (ich_i == 0) bias = bias_stream.read();
        if (och_i == 0) input = in_stream.read();

        // Completely unrolled section
        for (auto s_och=0; s_och < och_par; s_och++){
          for (auto s_ow=0; s_ow < ow_par; s_ow++){
            for (auto s_ich=0; s_ich < ich_par; s_ich++){
              compute_core<fh, fw>(s_acc_buff,
                input, weights, bias, output);
            }
          }
        }

        if (ich_idx == ich - ich_par)
          out_stream.write(output);
      }
    }
  }
}
```

Algorithm 1: Simplified convolution code from the *nn2fpga* HLS library. The convolutional loop is split into two parts, allowing the unrolled section to be adjusted based on template parameters.

stored in the line buffer to provide the input window to the computation task.

When considering more input windows convolved simultaneously, activations participate in multiple input windows in different relative positions. This results in activations flowing differently through the FIFO slices. More in detail:

- If ow^{par} is equal to 1, there is a one-to-one correspondence between the positions of the input and filter. Consequently, the activation must pass through all FIFO slices, because each of them represents a position within input/filter windows, as shown in Fig. 7.
- If ow^{par} is greater than 1, the input keeps several windows to be convolved with the filter. Thus, part of the activations are evaluated in adjacent positions for each input window, causing each activation to skip a number of FIFO equal to the unroll factor. Fig. 8 exemplifies how the input window is mapped to the input tensor when $ow^{par} = 2$.

During the analysis, the stride has been omitted to simplify the explanation. However, the same reasoning can be applied to the stride, acknowledging that activations are evaluated in positions that are not adjacent in the input window.

E. Graph Optimizations

The *nn2fpga* compiler provides a structured methodology to efficiently implement residual blocks in a dataflow accelerator with concurrent processes.

The input tensor of each residual blocks is processed by multiple nodes, with multiple branches starting from the convolution that generates the input tensor. Typically, in residual networks, branches end up being merged by an add layer. Fig. 9 shows *Resnet20* and *Resnet8* residual block topologies with two branches per input tensor and one or no convolutions on the skip connection. The add layer adds the values from the two branches.

Thanks to the dataflow architecture, processes start as soon as all input streams have data. However, in residual blocks, the time at which each input stream is required to receive data is different. In particular, the skip connection stream that reaches the add node has data available either in parallel with the conv_0 input stream in cases without downsampling, or upon initiation of conv_2 processing in cases with downsampling. Differently, the input stream from the long branch has data as soon as conv_1 provides its first output activation, but also conv_1 starts processing data as soon as its input buffer is full. Consequently, the amount of data that must be buffered in skip connections, B_{sc}, is equal to the amount needed by conv_0 to generate enough data to start conv_1. In the literature, this value is known as *receptive field* [43], and represents the portion of feature maps related to successive layers that contributes to produce an activation.

Fig. 10 shows the *receptive field* of the conv_1 window with respect to the conv_0 that generates it. B_{sc} is the buffering required to store both the receptive field (dark blue) and the activations in between consecutive rows (light blue) of the receptive field itself. From [43], the data to store for each receptive field B_{r} is

$$rh_0 = fh_1 + fh_0 - 1 \tag{5}$$
$$rw_0 = fw_1 + fw_0 - 1 \tag{6}$$
$$B_r = rh_0 \cdot rw_0 \tag{7}$$

Sliding the receptive field window over ich_0 and iw_0, the obtained buffering B_{sc} is equal to:

$$B_{\text{sc}} = [iw_0 \cdot (rh_0 - 1) + rw_0] \cdot ich_0 \tag{8}$$

In dataflow architectures, the "bypass" branch must store the input activation data from the previous stage until the convolution generates the first output activation, enabling the merged output to be produced.

To efficiently support *residual* blocks, the multiple endpoints of the input tensor and the increased buffering caused by the different number of convolutions (and thus different computation delays) per branch must be handled differently.

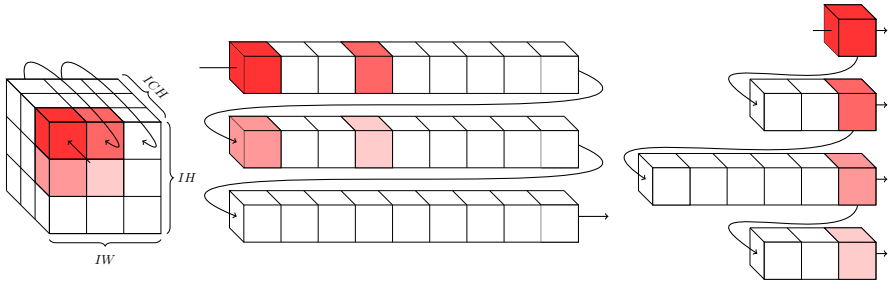


Fig. 6: Visual representation of how a 2×2 window is spread in the flattened tensor and then partitioned to form the line buffer. The activation tensor (leftmost image) is streamed in depth-first order, resulting in the flattened representation stored in the line buffer (center image). The line buffer is partitioned so that a whole input window is simultaneously available at its read points (rightmost image).

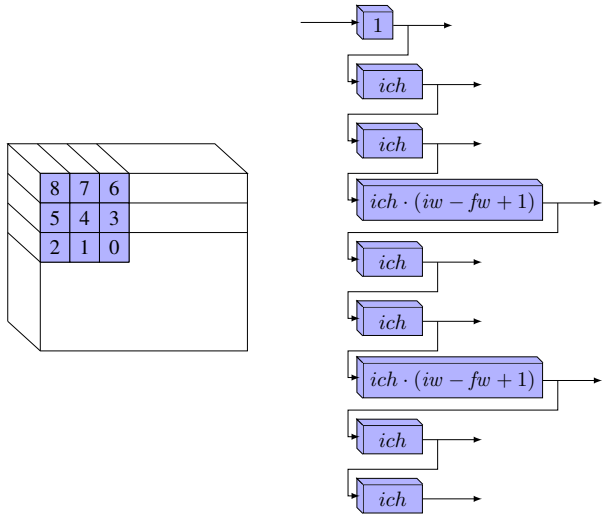


Fig. 7: Buffer partitioning in case of a 3×3 window, with $ow^{par} = 1$.

Such a drawback can be avoided by using a combination of optimization steps for dataflow architectures.

For residual blocks without downsampling, to avoid buffering the same tensor twice, the compiler optimizes the layer introducing a second output stream. This stream forwards the input tensor values to the next layer once the window buffer has completely used them. This optimization, called Temporal Reuse, reduces the required buffering of the skip connection for the residual block to the size of the conv_1 window buffer, calculated as:

$$B_{\text{sc}}^{\text{opt}} = [(fh_1 - 1) \cdot iw_0 + fw_1] \cdot ich_0. \tag{9}$$

thereby reducing the total amount needed by:

$$B_{\text{sc}} - B_{\text{sc}}^{\text{opt}} = [(fh_0 - 1) \cdot iw_0 + fw_0 - 1] \cdot ich_0. \tag{10}$$

For residual blocks with a downsample layer, which involves a pointwise convolution in the short branch of the skip connection, the compiler applies the Loop Merge optimization. It consists in merging the two convolutions acting on the same tensor into a single pipeline, allowing the layers to share the same window buffer and thus avoiding duplication of the input tensor data.

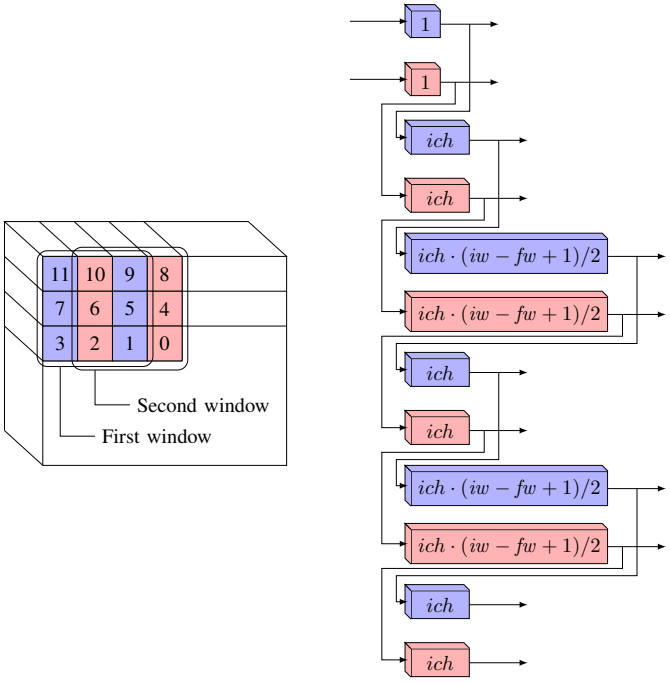


Fig. 8: Buffer partitioning in case of a 3×3 window, with $ow^{par} = 2$. The separate flows of activations belonging to the first and second windows are represented in red and blue.

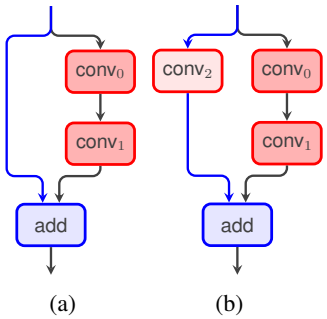


Fig. 9: *Resnet20* and *Resnet8* residual blocks. (a) and (b) depict the residual block without and with the downsampling convolution in the skip connection.

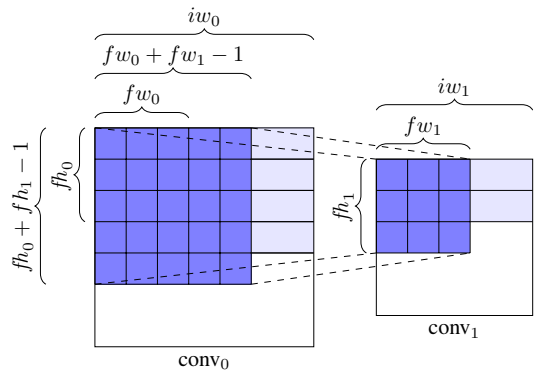
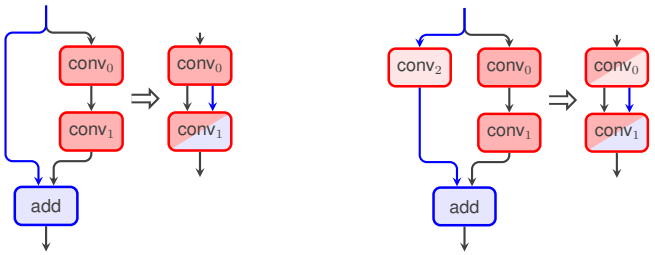


Fig. 10: Receptive field of the conv$_1$ window. For clarity, the ich dimension is omitted.



(a) (b)

Fig. 11: Complete graph optimization of residual blocks. (a) and (b) depict the residual block without and with the downsampling convolution in the skip connection.

A final transformation removes the sum of the value coming from the short branch by connecting it as an additional contribution to the second convolution of the long branch, merging the two operations in the same pipeline. The value from the skip branch is used to initialize the accumulator register and the addition is removed from the network graph.

Please note that the add operation is always merged with convolutions. As a result, the addition is done in the same pipeline of the convolution and adopts the same unrolling factor (ow^{ops}) of the conv operator.

These transformations ensure that the two input streams of the conv$_1$ are produced simultaneously, preventing computation stalls. conv$_0$ writes the skip connection stream as soon as the convolution computation starts and at the same rate as the convolution output tensor.

Fig. 11 shows the initial and final representations after applying the previously described optimizations to a network graph of a residual block with and without downsampling. Blocks with different colors highlight merge results of two operations together, such as conv$_1$ and add in Fig. 11a or conv$_2$ and conv$_0$ in in Fig. 11b.

F. Design space exploration

The framework targets low-power embedded devices, with the main focus on maximizing throughput via optimal resource allocation, and with power consumption being less of a concern.

The proposed methodology uses the *dataflow* paradigm with optimal stream sizing to prevent stalling. Hence the accelerator throughput is constrained only by the throughput of the slowest concurrent process.

The throughput of each task is influenced by the number of computations needed to process a single input frame (loop iterations) and its computational parallelism (unroll factor). Therefore the *nn2fpga* compiler uses an optimization model to find the optimal unroll factors for each layer, which maximizes the throughput of the accelerator under DSP and memory resource constraints. Considering the set G of all convolution and pool layers in the network, the optimization model is formulated as:

$$\begin{aligned} &\min_{S \in \mathbb{N}^+} S \\ \text{subject to} \quad &\forall i \in G \quad L_i^c \leq S \\ &\forall i \in G \quad L_i^w \leq S \\ &\sum_{i \in G} D_i \leq \text{DSP} \\ &\sum_{i \in G} M_i \leq \text{MEM} \end{aligned} \tag{11}$$

where S represents the latency of the slowest concurrent task processing its input tensor, which determines the overall network throughput. L_i^c and L_i^w are respectively the latencies of the computation and window buffer tasks associated to layer i, D is the number of DSPs utilized, and M is the number of memory banks required. MEM and DSP are constants representing the available resources on the target FPGA. As an example, consider the case of a generic convolution layer i.

The number of computations is then defined as:

$$c_i = oh_i \cdot ow_i \cdot och_i \cdot ich_i \cdot fh_i \cdot fw_i. \tag{12}$$

Since the parameter c_i is fixed and depends on the chosen network architecture, the layer latency can be adjusted by parallelizing computations. Considering the pseudocode in code 1, the computation parallelism c_i^{par} is:

$$\begin{aligned} &c_i^{par} = ow_i^{par} \cdot och_i^{par} \cdot ich_i^{par} \cdot fh_i \cdot fw_i \\ \text{with} \quad &ow_i^{par} \mid ow_i, \;\; ow_i^{par} \in \mathbb{N}^+ \\ &och_i^{par} \mid och_i, \;\; och_i^{par} \in \mathbb{N}^+ \\ &ich_i^{par} \mid ich_i, \;\; ich_i^{par} \in \mathbb{N}^+ \end{aligned} \tag{13}$$

which corresponds to the overall unroll factor of the internal loops of the convolution, specifically the number of 2D windows computed simultaneously. The filter size is defined by the model, hence the parameters that can be optimized are ow_i^{par}, och_i^{par}, and ich_i^{par}.

The latency in clock cycles is then modeled as:

$$L_i^c = \frac{c_i}{c_i^{par}} + \epsilon = \frac{oh_i \cdot ow_i \cdot och_i \cdot ich_i}{och_i^{par} \cdot ow_i^{par} \cdot ich_i^{par}} + \epsilon \tag{14}$$

where ϵ is the additional latency given by the pipeline depth. Throughout the paper ϵ is ignored, as it is negligible compared to the other terms.

The parallelization parameters are also impacting the latencies of the window buffer processes feeding the computation

tasks. Similar to convolutions, the latency of a window buffer task is defined as:

$$L_i^w = \frac{ich_i \cdot ih_i \cdot iw_i}{ich_i^{par} \cdot ow_i^{par}} \tag{15}$$

which is the number of elements in the input tensor divided by the number of elements shifted in parallel.

Each parallelism factor has a different impact on the resource utilization on the board. Parallelizing over ow demands less memory bandwidth for convolution parameters, since the same weights are reused to compute more output activations simultaneously. However, it requires more LUTs to store input activations, given the partitioning of the input window buffer into multiple channels. Conversely, parallelizing over och increases the memory bandwidth requirements for layer parameters but reduces window buffer parallelization. Both approaches (ow, och) can leverage DSP packing, as they enable one operand to be reused for computing multiple partial results. The third form of parallelization, over ich, proves useful for DSP chaining, as the accumulator is shared for all the partial results, saving resources.

Given the parallelism of the convolution task, the required DSPs can be modeled as:

$$D_i = \frac{och_i^{par} \cdot ow_i^{par} \cdot ich_i^{par} \cdot fh_i \cdot fw_i}{pack_i} \tag{16}$$

where $pack_i$ is the number of operations packed in a single DSP.

The on-chip memory requirements for a task are influenced by the bandwidth requirements as well as the overall task parameters. The number of memory banks required in parallel to fulfill the memory bandwidth requirements can be modeled as:

$$M_i^w = \left\lceil \frac{ich_i^{par} \cdot och_i^{par} \cdot fw_i \cdot fh_i \cdot bw_i}{bank_w} \right\rceil \tag{17}$$

in which bw_i is the bit-width of the parameters while $bank_w$ is the width of the on-chip memory port.

Then the total number of memory banks required to store the filters of the convolution task is:

$$M_i = M_i^w \cdot \left\lceil \frac{ich \cdot och}{ich_i^{par} \cdot och_i^{par}} \cdot \frac{1}{bank_d} \right\rceil \tag{18}$$

with $bank_d$ equal to the depth, i.e. the number of words available in each bank.

Formulated as above, Eq. (16) and Eq. (18) are nonlinear due to the ceiling function and the packing factor. Nevertheless, by enumerating the search space, it is possible to linearize these constraints. Enumerating the search space entails generating all the possible combinations of parallelism factors for each layer. This might seem like a huge search space, but in practice it can be explored efficiently, as discussed below.

Unroll factors are restricted to integers divisors of their respective dimension. From theory it is known that the number of distinct divisors for most integers n is lower than $log(n)$ [44]. Since we are taking into account tuples of three elements, the number of potential combinations grows in the worst case as $log(n)^3$ which is less than linear with respect to the input size. Hence the search space is limited and is further bounded by the actual number of resources available on the boards. Thus complete enumeration is feasible in practice for real CNNs and real FPGA, as shown in Section IV.

TABLE III: Resources of the boards used for our experiments.

Board	FPGA part	LUT	FF	BRAM	DSP	URAM
Ultra96	xczu3eg	70560	141120	216	360	0
Kria KV260	xczu5eg	117120	234240	144	1248	64
ZCU102	xczu9eg	274080	548160	912	2520	0

Since full enumeration is realistic, it is then possible to evaluate a priori the resource requirements for each combination, effectively removing the non-linearity from the problem. By assigning a binary variable to each combination, and imposing only one true variable per layer, problem (11) can be rewritten as a binary integer programming problem.

Since the problem focuses only on maximizing throughput, some layers may end up being over-parallelized. The *nn2fpga* compiler adopts a second step of optimization to reduce the parallelism of non-bottleneck layers, thus freeing resources. This step aims to ease the implementation of the accelerator, as it reduces the complexity of the design and speeds up the Vivado implementation process. It enables placement and routing to succeed, since these steps typically require resource utilizations of at most 70% to complete successfully with a good clock frequency. It uses the same formulation as in Eq. (11), but this time the latency S is fixed to the value found by the throughput optimization, and DSP and MEM are minimized.

IV. Experimental Results

The accelerators generated by *nn2fpga* are evaluated on the CIFAR-10 and Imagenet datasets, that exemplify common embedded machine vision applications. The former consists of 32×32 RGB images, while the latter of 224×224 RGB images. NN models targeting CIFAR-10 are trained using Brevitas QAT for 400 epochs with a batch size of 256, using the stochastic gradient descent (SGD) optimizer and cosine annealing as the learning rate scheduler. The MobilenetV2 model is based on a pre-trained model and it is quantized using Brevitas PTQ, without performing fine-tuning. QAT is not used in this case because it requires very significant computational resources during training, and the focus of this paper is on efficient HLS implementation of a given quantized network, rather than on network architecture exploration or quantization. The implementation flow uses Xilinx Vitis HLS 2023.2 for RTL code generation and Vivado 2023.2 for implementation on the Ultra96-v2, Kria KV260 and ZCU102 boards. Table III details the available resources for these three boards.

The *nn2fpga* compiler uses the PULP [45] library to solve binary integer programming problems with its default solver. Table IV shows the execution time of the solver for various combinations of model and board, and the reported times are contained, thus proving the claims made at the end of Section III-F.

The inference throughput (FPS, $\mathrm{Gops/s}$) and latency (ms) measured on the boards are detailed in Table VI, while the final resource utilization is reported in Table VII. Both tables show highlighted in bold the results of *nn2fpga*.

TABLE IV: Binary integer programming solver execution time.

Model	Board	Bit	Variables	Constraints	Time (s)
ResNet8	KV260	4	523	37	0.10
ResNet8	Ultra96	8	407	37	0.06
ResNet8	KV260	8	523	37	0.07
ResNet20	Ultra96	8	1107	85	0.09
ResNet20	KV260	8	1471	85	0.94
Mobilenet	ZCU102	8	6142	217	0.97

In the first section of Table VI, the *nn2fpga* implementation of ResNet20 is compared with the AdderNet and ResNet20 implementations presented in [46], both running on the Kria KV260. The Addernet is included in this comparison since, from the point of view of the architecture, it is a ResNet20 with multiplications replaced by additions. Our implementation achieves throughput improvements (Gops/s) of 2.88× and 1.94× with respect to the ResNet20 and the Addernet in [46]. Also, the latency is reduced by 3.84× and 1.96× respectively.

We compare on KRIA KV260 the *nn2fpga* ResNet8 with the implementation from Vitis AI [31], achieving a throughput improvement of 6.8× and a latency improvement of 28.1×. The third section of Table VI shows the results of a 4-bit implementation of a ResNet8, compared to the FINN version detailed in [31]. The comparison shows how *nn2fpga* is able to efficiently use FPGA resources even at lower bit precision, achieving a speedup of 4.53× and improving latency by 4.27×.

To further prove the flexibility of *nn2fpga*, Table VI also reports the performance of ResNet8 and ResNet20 deployed on the smaller Ultra96. There are no other implementations of these models on this board in the literature for comparison.

Finally, our implementation of MobilenetV2 on the ZCU102 is compared with the Vitis AI implementation [47] and with a custom RTL model [48]. Our implementation achieves a throughput of 2115 fps with a frequency of 214 MHz, and an accuracy of 71.73 %. This represents a 10% speedup over a state-of-the-art highly optimized manual RTL implementation, showing that HLS can actually improve over manual design, thanks to the faster exploration of the design space. While the maximum frequency that we achieved is lower than that proposed by [48], our solution demonstrates higher overall throughput, due to better utilization of the available resources provided by the ZCU102, as shown by the high DSP utilization that we achieved. The slight difference in accuracy of the models is attributable to the different quantization methods used, as well as the accuracy of the pre-quantization model.

To demonstrate the efficiency of the proposed graph optimizations beyond their dataflow benefits, we conducted an ablation study to highlight the reduction in on-chip activation storage required for the tested networks. Table V shows the resource used to store activations depending on the chosen implementations. ResNets can leverage both optimizations thanks to the presence of both downsample and no-downsample types of residual bottlenecks. In contrast, MobileNetV2 can only benefit from Temporal Reuse, as its inverted residual bottleneck does not contain convolutions in the short branch. Moreover, Temporal Reuse has a limited impact on resource savings for the inverted residual bottleneck, since the first convolution in the block is point-wise and thus does not increase the receptive field's dimension.

V. Conclusion

This work introduces a novel framework tailored for CNNs, specifically for skip-connection based networks. It supports the most commonly used operations for classic CNNs, including convolutions, fully connected (linear) layers, batch normalization, ReLU activation functions, max/average pooling, and skip connections. It is also fairly platform-independent since it is based on heavily templatized layer models and comes with a BIP-based optimization method to maximize throughput under resource constraints. This allows it to be used with various FPGA platforms, including the embedded ones that we used for our experimental evaluation. The static dataflow pipelined architecture minimizes buffering resources for networks with skip connections.

The effectiveness of the framework is validated by experiments on ResNet8 and ResNet20 using the CIFAR-10 dataset and on MobilenetV2 using the ImageNet dataset. The framework employs Vitis HLS and Vivado for hardware implementation on Kria KV-260, Ultra96-v2, and ZCU102 boards. Considering state-of-the-art NNs implemented on FPGAs, the comparison shows that the resource-efficient implementation of the residual layer, combined with resource-aware design space exploration, achieves implementations that Pareto dominate with respect to throughput and latency all state-of-the-art designs from the literature, when implemented on the same FPGAs.

In summary, the proposed framework shows potential as an alternative to commonly used residual network accelerators, delivering greater throughput and energy efficiency than the state-of-the-art, with optimized hardware resource utilization.

In the future we plan to support additional network layer types and new types of operations, such as the one required to support the head for detection and segmentation tasks, to further improve and extend the framework usefulness.

Acknowledgment

This work was partially supported by the Key Digital Technologies Joint Undertaking under the REBECCA Project with grant agreement number 101097224, receiving support from the European Union, Greece, Germany, Netherlands, Spain, Italy, Sweden, Turkey, Lithuania, and Switzerland. It was also partially supported by the Spoke 1 on Future HPC of the Italian Research Center on High-Performance Computing, Big Data and Quantum Computing (ICSC) funded by MUR Mission 4 – Next Generation EU. This work was also supported in part by Partenariato Esteso - RESTART "RESearch and innovation on future Telecommunications systems and networks, to make Italy more smART" - PE00000001.

References

[1] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: Analysis, applications, and prospects," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 12, pp. 6999–7019, 2022.

TABLE V: On-chip activation resource reduction achieved through various graph optimizations. ResNets are generated for KRIA KV260, while MobileNetV2 for ZCU102. All kernels are synthesized at 300MHz in out-of-context mode, with results obtained from the synthesis report of Vivado.

	No opt.			Temporal Reuse			Loop Merge			Temporal Reuse + Loop Merge		
Model	kLUT	kFF	BRAM	kLUT	kFF	BRAM	kLUT	kFF	BRAM	kLUT	kFF	BRAM
ResNet8	14.7	11.2	8.5	14.4	11.2	8.5	14.2	11.1	7.5	13.3 (-9.5%)	11.1 (-0.9%)	7.5 (-11.7%)
ResNet20	27.2	18.1	8.5	25.9	17.8	6.0	26.7	17.9	8.5	25.6 (-6.0%)	17.7 (-2.2%)	6.0 (-29.4%)
MobileNetV2	82.0	44.1	102.5	79.7	43.7	102.5	82.0	44.1	102.5	79.7 (-2.8%)	43.7 (-0.9%)	102.5 (-0.0%)

TABLE VI: Performance comparison with SOTA accelerators. Power was measured using sensors on the power rails and is representative of the entire board consumption. Throughput was evaluated with a batch size of 400 for ImageNet and 1000 for CIFAR10.

Model	**Dataset**	**FPGA**	**Bit**	**Freq.** (MHz)	**Throughput** (FPS)	**Throughput** (Gops/s)	**Latency** (ms)	**Power** (W)	**Accuracy** (%)
ResNet20† [46]	CIFAR10	KV260	8	200	N/A	214	1.221	1.07†	90.8
AdderNet† [46]	CIFAR10	KV260	8	200	N/A	317	0.624	1.52†	89.9
ResNet20	CIFAR10	KV260	8	250	7601	616	0.318	6.10	91.3
ResNet8 Vitis AI [31]	CIFAR10	KV260	8	200	4458	109	1.293	6.42	89.2
ResNet8	CIFAR10	KV260	8	250	30153	773	0.046	6.67	88.7
ResNet8 FINN [31]	CIFAR10	KV260	4	225	13475	330	0.154	5.89	85.9
ResNet8	CIFAR10	KV260	4	250	61035	1526	0.036	6.12	86.9
ResNet20	CIFAR10	Ultra96	8	214	3254	264	0.807	1.04	91.3
ResNet8	CIFAR10	Ultra96	8	214	12971	317	0.111	0.56	88.7
MobileNetV2 Vitis AI [47]	ImageNet	ZCU102	8	281	765	N/A	N/A	N/A	67.67
MobileNetV2 [48]	ImageNet	ZCU102	8	333	1910	N/A	N/A	N/A	72.98
MobileNetV2	ImageNet	ZCU102	8	214	2115	1403	2.061	13.5	71.73

† The description of how to measure power consumption is not explicitly provided in [46]. Accounting for just the board idle consumption measured with `xmutil`, such a measure is already above the values reported in the referenced paper.

TABLE VII: Resource utilization comparison. Reported data are collected from the Vivado report after place and route.

Model	**FPGA**	**Bit**	**kLUT**	**kLUTRAM**	**kFF**	**DSP**	**BRAM**	**URAM**
ResNet20 [46]	KV260	8	41.8 (35.7 %)	17.6 (30.1 %)	34.0 (14.5 %)	545 (43.7 %)	40.0 (27.7 %)	N/A
AdderNet [46]	KV260	8	67.4 (57.6 %)	22.2 (38.6 %)	43.2 (19.1 %)	609 (48.8 %)	40.0 (27.7 %)	N/A
ResNet20	KV260	8	65.0 (55.5 %)	13.0 (22.6 %)	81.7 (34.9 %)	636 (51.0 %)	60.5 (42.0 %)	12 (18.7 %)
ResNet8 VITIS AI [31]	KV260	8	25.6 (21.8 %)	N/A	33.7 (14.4 %)	110 (8.8 %)	8.8 (8.8 %)	18 (28.1 %)
ResNet8	KV260	8	55.4 (47.3 %)	9.4 (16.4 %)	70.1 (29.9 %)	767 (61.5 %)	63.5 (44.1 %)	0 (0 %)
ResNet8 FINN [31]	KV260	4	81.4 (69.5 %)	N/A	87.6 (37.4 %)	N/A	28.5 (19.8 %)	N/A
ResNet8	KV260	4	40.4 (34.5 %)	10.3 (17.9 %)	64.8 (27.7 %)	794 (63.6 %)	58.0 (40.3 %)	0 (0 %)
ResNet20	Ultra96	8	54.4 (77.1 %)	10.2 (35.6 %)	57.6 (40.8 %)	318 (88.3 %)	89.5 (41.4 %)	0 (0 %)
ResNet8	Ultra96	8	46.4 (65.8 %)	6.2 (21.5 %)	45.1 (32.0 %)	360 (100 %)	54.0 (25 %)	0 (0 %)
MobileNetV2 VITIS AI [47]	ZCU102	8	N/A	N/A	N/A	N/A	N/A	N/A
MobileNetV2 [48]	ZCU102	8	170.4 (62.2 %)	N/A	154.3 (28.2 %)	1283 (50.9 %)	691 (75.8 %)	0 (0 %)
MobileNetV2	ZCU102	8	139.4 (50.9 %)	63.3 (44.0 %)	162.8 (29.7 %)	1797 (71.3 %)	912 (100 %)	0 (0 %)

[2] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *ArXiv e-prints*, 11 2015.

[3] P. Dhilleswararao, S. Boppu, M. S. Manikandan, and L. R. Cenkeramaddi, "Efficient hardware architectures for accelerating deep neural networks: Survey," *IEEE Access*, vol. 10, pp. 131 788–131 828, 2022.

[4] Q. Zhang, M. Zhang, T. Chen, Z. Sun, Y. Ma, and B. Yu, "Recent advances in convolutional neural network acceleration," *CoRR*, vol. abs/1807.08596, 2018. [Online]. Available: http://arxiv.org/abs/1807.08596

[5] 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. [Online]. Available: https://arxiv.org/abs/1704.04861

[6] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices." *CoRR*, vol. abs/1707.01083, 2017. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1707.html#ZhangZLS17

[7] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (Canadian Institute for Advanced Research)." [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[8] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of 2016 IEEE Conference on Computer Vision and Pattern Recognition*, ser. CVPR '16. IEEE, Jun. 2016, pp. 770–778. [Online]. Available: http://ieeexplore.ieee.org/document/7780459

[10] O. Weng, A. Khodamoradi, and R. Kastner, "Hardware-efficient residual networks for fpgas," 2021. [Online]. Available: https://arxiv.org/abs/2102.01351

[11] N. Fasfous, M. R. Vemparala, A. Frickenstein, E. Valpreda, D. Salihu, J. Höfer, A. Singh, N.-S. Nagaraja, H.-J. Voegel, N. A. Vu Doan, M. Martina, J. Becker, and W. Stechele, "Anaconga: Analytical hw-cnn co-design using nested genetic algorithms," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022, pp. 238–243.

[12] C. Liu, P. Chen, B. Zhuang, C. Shen, B. Zhang, and W. Ding,

"SA-BNN: state-aware binary neural network," in *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, 2021, pp. 2091–2099. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/16306

[13] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu *et al.*, "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 11, no. 3, pp. 1–23, 2018.

[14] H. Qin, R. Gong, X. Liu, X. Bai, J. Song, and N. Sebe, "Binary neural networks: A survey," *Pattern Recognition*, vol. 105, p. 107281, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0031320320300856

[15] J. Choi, S. Venkataramani, V. V. Srinivasan, K. Gopalakrishnan, Z. Wang, and P. Chuang, "Accurate and efficient 2-bit quantized neural networks," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 348–359, 2019.

[16] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. H. W. Leong, M. Jahre, and K. A. Vissers, "FINN: A framework for fast, scalable binarized neural network inference," *CoRR*, vol. abs/1612.07119, 2016. [Online]. Available: http://arxiv.org/abs/1612.07119

[17] Xilinx Inc., *Vitis High-Level Synthesis User Guide*, 2022. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2022_2/ug1399-vitis-hls.pdf

[18] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, and G. Boudoukh, "Can fpgas beat gpus in accelerating next-generation deep neural networks?" in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 5–14. [Online]. Available: https://doi.org/10.1145/3020078.3021740

[19] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2017, pp. 1–6.

[20] Y. Yu, T. Zhao, K. Wang, and L. He, "Light-opu: An fpga-based overlay processor for lightweight convolutional neural networks," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. Association for Computing Machinery, 2020, p. 122–132. [Online]. Available: https://doi.org/10.1145/3373087.3375311

[21] L. Petrica, T. Alonso, M. Kroes, N. Fraser, S. Cotofana, and M. Blott, "Memory-efficient dataflow inference for deep cnns on fpga," 2020. [Online]. Available: https://arxiv.org/abs/2011.07317

[22] AMD-Xilinx, "Vitisai develop environment," 2023. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html

[23] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[24] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035.

[25] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM International Conference on Multimedia*, ser. MM '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 675–678. [Online]. Available: https://doi.org/10.1145/2647868.2654889

[26] Xilinx, "Dpuczdx8g for zynq ultrascale+ mpsocs product guide (pg338)," [Online]. Available: https://docs.xilinx.com/r/en-US/pg338-dpu/Core-Overview, January 2023.

[27] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, "Pruning and quantization for deep neural network acceleration: A survey," *Neurocomputing*, vol. 461, pp. 370–403, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231221010894

[28] S. A. Alam, D. Gregg, G. Gambardella, T. Preusser, and M. Blott, "On the rtl implementation of finn matrix vector unit," *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 6, nov 2023. [Online]. Available: https://doi.org/10.1145/3547141

[29] A. Ushiroyama, M. Watanabe, N. Watanabe, and A. Nagoya, "Convolutional neural network implementations using vitis ai," in *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*, 2022, pp. 0365–0371.

[30] M. Machura, M. Danilowicz, and T. Kryjak, "Embedded object detection with custom littlenet, finn and vitis ai dcnn accelerators," *Journal of Low Power Electronics and Applications*, vol. 12, no. 2, 2022. [Online]. Available: https://www.mdpi.com/2079-9268/12/2/30

[31] F. Hamanaka, T. Odan, K. Kise, and T. V. Chu, "An exploration of state-of-the-art automation frameworks for fpga-based dnn acceleration," *IEEE Access*, vol. 11, pp. 5701–5713, 2023.

[32] G.-M. Liang, C.-Y. Yuan, M.-S. Yuan, T.-L. Chen, K.-H. Chen, and J.-K. Lee, "The support of mlir hls adaptor for llvm ir," in *Workshop Proceedings of the 51st International Conference on Parallel Processing*, ser. ICPP Workshops '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3547276.3548515

[33] H. Ye, H. Jun, H. Jeong, S. Neuendorffer, and D. Chen, "Scalehls: a scalable high-level synthesis framework with multi-level transformations and optimizations: invited," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1355–1358. [Online]. Available: https://doi.org/10.1145/3489517.3530631

[34] A. Pappalardo, "Xilinx/brevitas," 2022. [Online]. Available: https://doi.org/10.5281/zenodo.3333552

[35] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," 2021.

[36] Q. Jin, J. Ren, R. Zhuang, S. Hanumante, Z. Li, Z. Chen, Y. Wang, K. Yang, and S. Tulyakov, "F8net: Fixed-point 8-bit only multiplication for network quantization," in *International Conference on Learning Representations*, 2022. [Online]. Available: https://openreview.net/forum?id=_CfpJazzXT2

[37] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713.

[38] A. Pappalardo, Y. Umuroglu, M. Blott, J. Mitrevski, B. Hawks, N. Tran, V. Loncar, S. P. Summers, H. Borras, J. Muhizi, M. Trahms, S.-C. Hsu, and J. M. Duarte, "QONNX: Representing Arbitrary-Precision Quantized Neural Networks," in *4th Workshop on Accelerated Machine Learning (AccML) at HiPEAC 2022 Conference*, 6 2022. [Online]. Available: https://accml.dcs.gla.ac.uk/papers/2022/4thAccML_paper_1(12).pdf

[39] Y. Umuroglu, H. Borras, V. Loncar, S. Summers, and J. Duarte, "fastmachinelearning/qonnx," 06 2022. [Online]. Available: https://github.com/fastmachinelearning/qonnx

[40] E. W. Yao Fu and A. Sirasao, "8-bit dot-product acceleration (wp487)," 2017. [Online]. Available: https://docs.xilinx.com/v/u/en-US/wp487-int8-acceleration

[41] "Convolutional Neural Network with INT4 Optimization on Xilinx Devices," 2020. [Online]. Available: https://docs.amd.com/v/u/en-US/wp521-4bit-optimization

[42] K. Goetschalckx and M. Verhelst, "Breaking high-resolution cnn bandwidth barriers with enhanced depth-first execution," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 323–331, 2019.

[43] A. Araujo, W. Norris, and J. Sim, "Computing receptive fields of convolutional neural networks," *Distill*, 2019, https://distill.pub/2019/computing-receptive-fields.

[44] G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*, 4th ed. Oxford, 1975.

[45] I. Dunning, S. Mitchell, and M. O'Sullivan, "Pulp: A linear programming toolkit for python," 2011. [Online]. Available: https://optimization-online.org/?p=11731

[46] Y. Zhang, B. Sun, W. Jiang, Y. Ha, M. Hu, and W. Zhao, "Wsq-addernet: Efficient weight standardization based quantized addernet fpga accelerator design with high-density int8 dsp-lut co-packing optimization," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3508352.3549439

[47] "Vitis AI Model Zoo," 2023. [Online]. Available: https://xilinx.github.io/Vitis-AI/3.0/html/docs/workflow-model-zoo.html

[48] W. Jiang, H. Yu, and Y. Ha, "A high-throughput full-dataflow mobilenetv2 accelerator on edge fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 5, pp. 1532–1545, 2023.