electronics

Article

A Heterogeneous Inference Framework for a Deep Neural Network

Rafael Gadea-Gironés *, José Luís Rocabado-Rocha, Jorge Fe and Jose M. Monzo

Institute for Molecular Imaging Technologies (I3M), Universitat Politècnica de València, 46022 Valencia, Spain; jororoc@alumni.upv.es (J.L.R.-R.); jmonfer@upvnet.upv.es (J.M.M.)
* Correspondence: rgadea@eln.upv.es

Abstract: Artificial intelligence (AI) is one of the most promising technologies based on machine learning algorithms. In this paper, we propose a workflow for the implementation of deep neural networks. This workflow attempts to combine the flexibility of high-level compilers (HLS)-based networks with the architectural control features of hardware description languages (HDL)-based flows. The architecture consists of a convolutional neural network, SqueezeNet v1.1, and a hard processor system (HPS) that coexists with acceleration hardware to be designed. This methodology allows us to compare solutions based solely on software (PyTorch 1.13.1) and propose heterogeneous inference solutions, taking advantage of the best options within the software and hardware flow. The proposed workflow is implemented on a low-cost field programmable gate array system-on-chip (FPGA SOC) platform, specifically the DE10-Nano development board. We have provided systolic architectural solutions written in OpenCL that are highly flexible and easily tunable to take full advantage of the resources of programmable devices and achieve superior energy efficiencies working with a 32-bit floating point. From a verification point of view, the proposed method is effective, since the reference models in all tests, both for the individual layers and the complete network, have been readily available using packages well known in the development, training, and inference of deep networks.

Keywords: convolutional neural networks; heterogeneous computation; systolic arrays; FPGA

Citation: Gadea-Gironés, R.; Rocabado-Rocha, J.L.; Fe, J.; Monzo, J.M. A Heterogeneous Inference Framework for a Deep Neural Network. *Electronics* **2024**, *13*, 348. https://doi.org/10.3390/electronics13020348

Academic Editors: Shuping Zhao, Jie Wen, Chao Huang, Bob Zhang and Luca Patanè

Received: 7 December 2023
Revised: 9 January 2024
Accepted: 11 January 2024
Published: 14 January 2024

1. Introduction

Recently, the development of technologies based on artificial intelligence and machine learning algorithms has surged. These technologies have a wide range of applications, from image classification and object recognition for medical diagnoses and quality control to text generation.

However, these technologies rely heavily on machines with considerable computing power, so processing is usually carried out on cloud servers to facilitate access to the end user. If we want to bring artificial intelligence closer to electronics, this approach is not suitable due to latency, power consumption, and size constraints.

In response to this and thanks to improvements in the latest processors, a new approach known as artificial intelligence "on the edge" has appeared. The philosophy of this approach consists of carrying out the processing of the data acquired in a node within the node itself so it can act once the results have been obtained, allowing the electronics to be independent [1].

Therefore, it is possible to find new architectures whose objective is to achieve the same functionalities as the rest of the technologies but in exchange for reducing the results' accuracy and complexity.

In this work, we focus on convolution-based architectures because they are a clear trend in the current landscape of edge computing. This type of architecture allows us to extract features or characteristics from the input images, classify them, or detect different objects.

Darknet or VGG are examples of these architectures, as is an interesting option known as SqueezeNet v1.1. SqueezeNet is a new version of [2] that maintains the same accuracy and reduces the computational cost by a factor of 2.4.

We can observe the use of SqueezeNet in applications such as: self-driving cars [3], face-mask detection [4], weather classification [5], fire detection [6,7], garden insect recognition [8], and indoor obstacle classification [9]. In many of these applications, the key element is to be able to rely on a solution that can perform prediction, recognition, classification, and detection tasks with resource-constrained systems that provide maximum operational autonomy without compromising the accuracy achieved. We have implemented SqueezeNet v1.1, a convolutional neural network, on low-cost FPGA SOC platforms, such as the DE10-Nano development board, which is our primary contribution. This system includes an ARM HPS processor that works with the acceleration hardware. This enables us to compare solutions based solely on software (PyTorch 1.13.1) and create heterogeneous inference solutions that take advantage of the best options from both software and hardware flows.

To reduce design time, we resort to high-level compilers (HLS) by which the abstraction level is increased to generate the synthesis of the hardware by a C/C++ code. One of the most interesting options provided by the manufacturer that allows us to take advantage of the FPGAs' capabilities to optimize the results with energy efficiency and low latency is the Intel FPGA SDK for OpenCL. With this tool, we can implement our image-sorting algorithm, inferring acceleration hardware with a parallelism capability as set by the OpenCL standard inside the low-cost development board, DE10-Nano.

A conceptual diagram summarizing the main goal is shown in Figure 1.

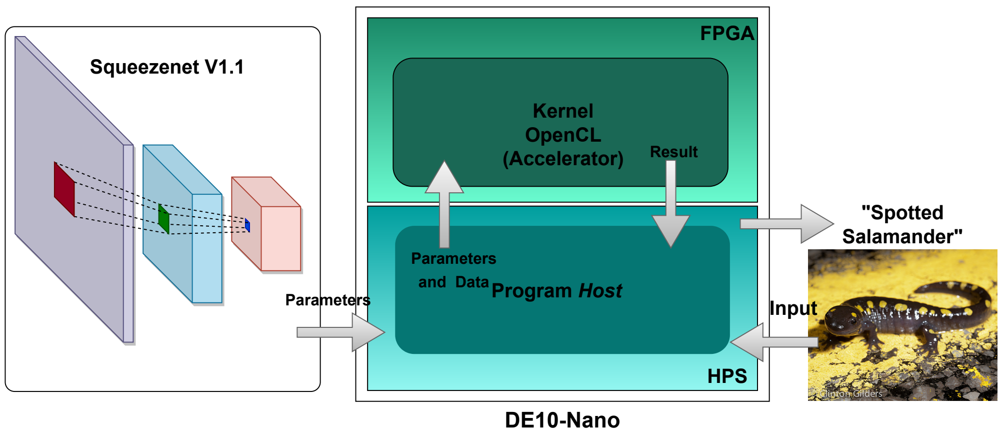

Figure 1. Conceptual diagram of the main objective.

Methodologically speaking, this work's main contribution consists of basing the whole verification flow on Python using Jupyter notebooks that access our accelerators through PyopenCL. This technology is already quite common in working with GPUs but is rarely used in low-cost FPGAs in which the host is exercised by an ARM hard macro.

The remainder of this article is organized as follows. Section 2 reviews CNN implementations performed on low-cost FPGAs. In Section 3, the elements used in the proposed design and verification methodology are determined. In Section 4, the implementation results obtained for a CNN are analyzed by comparing them with other available environments. Conclusions are presented in Section 5.

2. SqueezeNet Implementations Review

In the first subsection, we review the fundamentals of the convolutional layers, which are fundamental to understanding the topology of the SqueezeNet convolutional network. Above all, we consolidate the terminology used throughout the article and establish the theoretical foundations.

Other types of layers commonly used in convolutional architectures and the selected model, SqueezeNet V1.1, are also discussed.

Once we have all the pieces of the puzzle that make up this type of network, we discuss the main structural element that gives essence to this type of network, the fire module.

We then review works that work with this type of network, with low-cost FPGA devices that can develop AI at the edge.

Finally, we discuss the OpenCL standard and its two main approaches to address different types of problems from the point of view of FPGA solutions.

2.1. Convolution Layer

The convolution layer is normally used to process input or feature maps whose spatial distribution is two-dimensional, as in the case of an image whose size is usually expressed by pixel width and height, $H_{pixel} \times W_{pixel}$.

However, images usually have different channels that contain color information in the format $H_{Red-Green-Blue}$. Therefore, the inputs to the convolution layers usually have the format $H_{in} \times W_{in} \times CH_{in}$, which are the height, width, and channel depth of the input feature map, respectively.

Consequently, the 2D convolution layers are formed by several filters of dimension $H_k \times W_k \times CH_{in}$ to maintain the agreement with the input data. Regarding the size of the filters, $H_k \times W_k$, we can find different combinations with the common sizes 1×1, 3×3, and 5×5. Another parameter that is part of the convolutions is called the bias. It consists of a scalar value for each convolution filter.

Figure 2 visually represents the dimensions, with their nomenclature, of a feature map and a filter used in a 3×3 convolution.

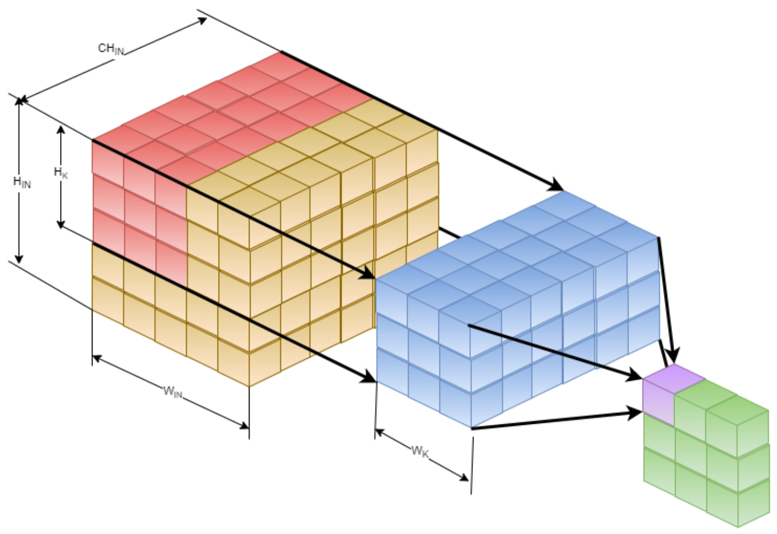

Figure 2. Representation of CH_{in} feature maps and CH_{in} kernels with the nomenclature of its dimensions.

The result of the convolution corresponds to the sum of the product of the filter with the input data element by element. The filter slides through the size of the input data, producing an output value for each displacement. If there is more than one channel, the filter is applied to its corresponding channel, and all channels' results are summarized.

Assume that a $H_{in} \times W_{in} \times CH_{in}$ input and a single filter $H_k \times W_k \times CH_{in}$. The first element of the output corresponds to the following:

$$out(0,0) = bias + \sum_{c=0}^{CH_{in}} \sum_{i=0}^{W_k} \sum_{j=0}^{H_k} filter(c,i,j) \cdot feature(c,i,j) \quad (1)$$

The size of the output depends mainly on the size of the kernel, the sliding factor or stride, and an optional setting known as padding that allows a frame, usually of zeros, to be added around the input.

The input and filter being square with sizes in_{size} and k_{size}, respectively, we can calculate the size of the output out_{size} as:

$$out_{size} = \frac{in_{size} + 2 \cdot padding - k_{size}}{stride} + 1 \tag{2}$$

where *padding* is the size of the added frame, and *stride* is the positions the filter moves over the input data.

The number of channels of the output CH_{out} depends on the number of filters the convolution applies.

Figure 3 shows a visual example of a convolution layer operation. We observe an input feature map with a size in_{size} of 5 and a filter with a k_{size} of 3, since this is a 3×3 convolution, with a *stride* and padding of 1.

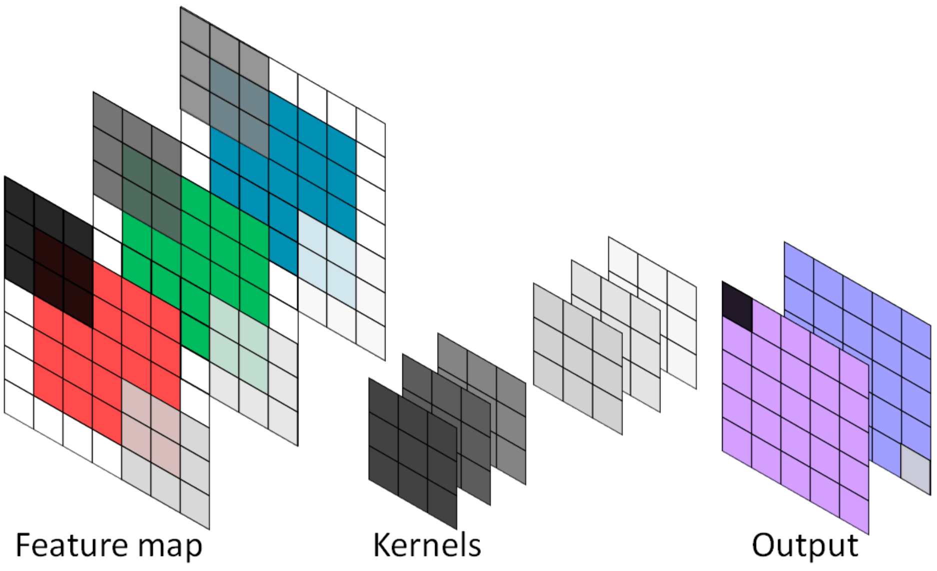

Figure 3. Example of 3×3 convolution with an input image of dimensions $5 \times 5 \times 3$ and two *filters* of dimensions $3 \times 3 \times 3$ with a *padding* and *stride* of 1.

Applying Formula (2), we can verify that the output size out_{size} is 5. Also, since we have two filters, there are 2 output channels (CH_{out}), obtaining an output dimension of $5 \times 5 \times 2$.

To conclude, it is necessary to comment that a widespread practice is to apply an activation function to the results obtained in the convolution and to apply nonlinearities to the data, allowing the network to tackle more complex problems. The most common is ReLU, whose piecewise function is found in Equation (3).

$$ReLU(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

2.2. Layers of Type Pool

There are two main types of layers, max-pooling and average-pooling (both present in SqueezeNet v1.1), depending on the type of data extraction they perform. These layers have a procedure similar to convolutions, consisting of a filter of a given size that slides through the input data, performing the corresponding operation. In addition, both the stride and the padding can be configured. However, they do not affect the channel dimensionality of the input feature map.

The max-pool layer extracts the input data with the highest value within the kernel size, while the average-pool layer calculates the average of the characteristics of the input data.

2.3. SqueezeNet

SqueezeNet, [2], is a CNN architecture whose main objective has been to reduce the number of parameters without significantly affecting the classified images' precision level. Reducing the number of parameters in the convolutional network makes its implementation feasible in FPGA or embedded systems. Moreover, it allows the parameters to be stored in the on-chip memory, reducing the bottleneck generated by the memory access bandwidth.

To do this, in [2], we present the fire modules that make up the global architecture. These modules consist of a Squeeze layer, corresponding to a 1 $\times$ 1 convolution, whose output connects directly with an Expand layer, formed by a mixture of 1 $\times$ 1 and 3 $\times$ 3 convolutions, whose concatenated results generate the module output.

Using a squeeze layer before the 3 $\times$ 3 convolution allows one to reduce the number of input channels and the number of kernels of each filter. Figure 4 shows the layout of the different convolutions in a fire module.

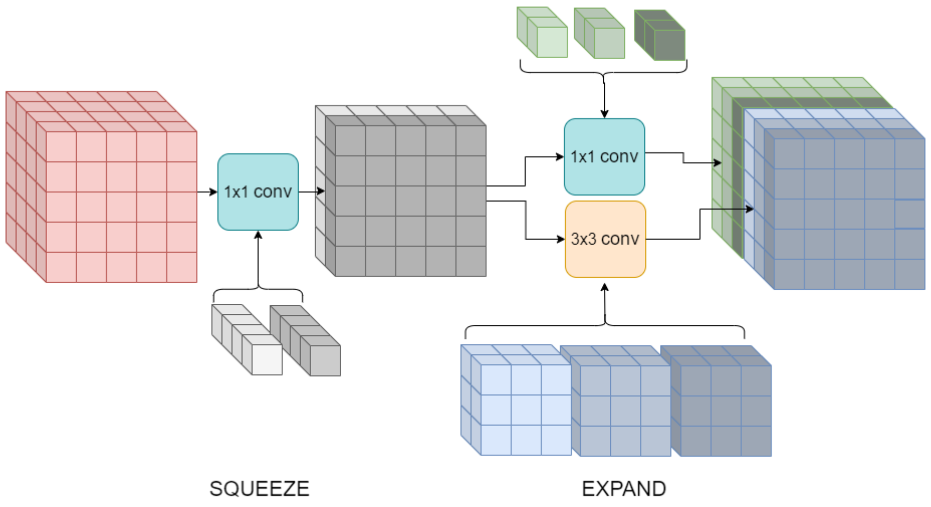

Figure 4. Structure of the fire module made up of the different convolution filters. For example, in this figure, we have a squeeze factor, $S_{1\times1}$, of 2; an expand factor of the 1 $\times$ 1 convolution, $E_{1\times1}$, of 3; and an expand factor of the 3 $\times$ 3 convolution, $E_{3\times3}$, of 3.

Subsequently, a new version of SqueezeNet, SqueezeNet v1.1, was released in the official repository [2]. This version modifies the original architecture, keeping the fire modules, slightly reducing the number of parameters, and obtaining a 2.4 reduction in the computational cost without losing accuracy.

A comparison between the versions of the error classifications of the pretrained model with Imagenet images is available in the documentation of the PyTorch open-source machine learning library [10]. The results are shown in Table 1.

Table 1. Comparative table of the prediction errors of the SqueezeNet versions with the Imagenet dataset. Data obtained from [10].

Structure of the Model	Error Top-1	Error Top-5
SqueezeNet V1.0	41.90%	19.58%
SqueezeNet V1.1	41.81%	19.38%

We can see the structure and configuration of SqueezeNet v.1.1 in Figure 5 and Table 2. It is a succession of layers (14). If we break down each layer into its constituent units (Figure 5), we need to implement four fundamental elements: a 1 $\times$ 1 convolution, a 3 $\times$ 3 convolution, and max-pool and average-pool operations. Of course, the reuse of these units requires that the implementations be reconfigurable to adapt to the different sizes and parameters listed in Table 2.

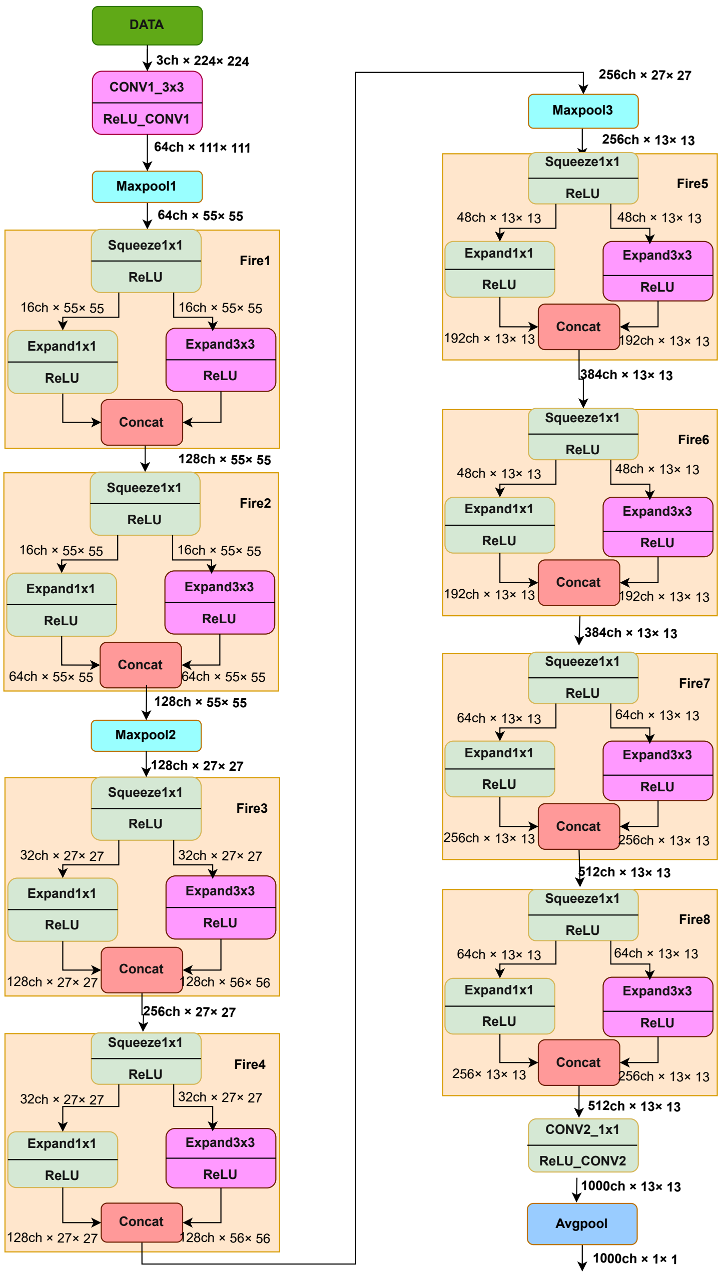

Figure 5. SqueezeNet v1.1 structure.

Table 2. The architecture of the SqueezeNet v1.1 model [10].

Layer Name/Type	**Output Size**	**Size/Stride of Filter (No Fire Modules)**	$S_{1\times1}$	$E_{1\times1}$	$E_{3\times3}$
Input image	224 × 224 × 3				
conv1	111 × 111 × 64	3 × 3/2 (× 64)			
maxpool1	55 × 55 × 64	3 × 3/2			
fire1	55 × 55 × 128		16	64	64
fire2	55 × 55 × 128		16	64	64
maxpool2	27 × 27 × 128	3 × 3/2			
fire3	27 × 27 × 256		32	128	128
fire4	27 × 27 × 256		32	128	128
maxpool3	13 × 13 × 256	3 × 3/2			
fire5	13 × 13 × 384		48	192	192
fire6	13 × 13 × 384		48	192	192
fire7	13 × 13 × 512		64	256	256
fire8	13 × 13 × 512		64	256	256
conv2	13 × 13 × 1000	1 × 1/1 (×1000)			
avgpool1	1 × 1 × 1000	13 × 13/1			

2.4. Related Work

Shortly after the appearance of the SqueezeNet model, David Gschwend [11] implemented the model on a low-cost Zynqbox development platform that included a Xilinx Zynq-7000 FPGA combined with an ARM Cortex-A9. The implemented solution was called ZynqNet and was divided into two parts.

Modifications were made to the original SqueezeNet model to adapt it to FPGA requirements. One of the most notable modifications was eliminating the max-pool layers, following the philosophy of "all convolutional networks", where the network is composed only of convolutional layers. To obtain the same result by reducing the output size, they added a stride of two in the convolution immediately after that. However, in SqueezeNet, this layer always consists of a 1 × 1 convolution, which would result in information loss. Therefore, they replaced this first layer with a 3 × 3 filter. They also implemented the final layer of type average pool, which could be considered a convolution whose filters are composed of values $kernel_{size}^{-1}$.

Another modification worth mentioning is the resizing of the layers, making their height and width powers of two. This is notable because this type of change is common in several implementations, which condition the type of CNN topology developed, achieving better latency values in exchange. On the other hand, in implementing the accelerator, the authors opted to use a single-precision floating point for data representation to maintain compatibility with their implementation of ZynqNet in GPU.

In addition, they leaned on the use of cache memories to facilitate memory reads and writes and also chose to unwind the loops of the 3 × 3 convolutions, thus achieving a top-5 error on Imagenet of 15.4% using a 32-bit floating-point data representation. We highlight that the accelerator was optimized to perform 3 × 3 convolutions, and it is possible that any other configuration would require additional logic, resulting in a low utilization of multiple-accumulate operations. It is worth noting that they performed their implementation using Vivado's HLS.

Work similar to ZynqNnet was carried out in [12], resulting in EdgeNet. This proposed convolutional network was implemented on a DE10-Nano board, and its proposed accelerator consisted of a configurable computing block. This unit was designed to work inversely, with the accelerator input equivalent to an expand layer and the output being the squeeze layer. This allowed fewer input channels to be accessed, as the input of the computation unit was mostly the result of a 1 × 1 convolution of the squeeze layer. The unit's output was also the squeeze layer. Additionally, the unit was able to access pool-type layers depending on the configuration of the data path.

It should be noted that the results were obtained with a representation of the data and parameters using 8-bit floating points (five bits of exponent and two of mantissa) of the parameters resulting from the training of [11], obtaining a reduction of 6% in accuracy in the classifications and achieving 51% of top-1 accuracy with Imagenet.

Continuing with the quantification of the parameters of convolutional neural networks, we find again in [13] an implementation of SqueezeNet v1.1 where, prior to the design of the accelerator, the authors performed a study of the effect of quantification on the model's accuracy.

They observed that by using the 8-bit integer data type, they considerably reduced both the FPGA resources and the memory access, obtaining a loss of 0.69% and 0.72%, resulting in 57.49% and 79.90%, the first and fifth in accuracy, respectively.

It is worth mentioning that this implementation was carried out on the low-cost DE10-Nano board using the OpenCL standard and HLS.

In addition, they incorporated additional logic that allowed them to read the input data, filter the convolution, and write the results into the global memory. Also, because the ReLU activation function generates feature maps with results equal to zero, they introduced a specific control with which they avoided performing operations with null input values.

Continuing with implementations using reduced fixed-point precision, we have the work of [14]. They used a Zynq-7020 for the implementation based on the ZC702 development board. The authors employed 8 bits for the parameters (weights and bias) and 16 bits for the activations and carried out fixed-point arithmetic. This yielded an execution time of 333 ms. Their energy efficiency measurements were obtained from Vivado's Xilinx Estimator Power (XPE) and resulted in a 2275-watt consumption. The Xilinx HLS tool was used for the design, which required two kernels for implementation. In particular, 186 DSPs were used in its implementation.

Similar works to those with Zynq-7020 can be found in [15], using HLS but working with 32-bit floating point. They reached execution times of 1 second, and curiously, with coupling degrees similar to those previously mentioned, they obtain 7.95 watts of power consumption again using XPE.

Later, in [16], the researchers developed a project on the SOC DE10-Nano board, the implementation of SqueezeNet v1.1 conferring parallelism at a multithreaded level, using the HLS of OpenCL.

They provided detailed documentation of the steps followed to carry out their design. They included different versions where applying optimization techniques of kernels as the coalescent memory access and the loop unrolling reduced the time required in the network's prediction. Reviewing the implementation, it is observed that using coalescent memory through the float4 data structure limits the number of channels of each convolution layer since they must be a multiple of four due to the number of data elements that are read in a coalescent way.

3. Methodology and Architectures

We developed three architectures. We developed a Python-based method using Jupyter notebooks for design and verification. Our kernels developed in Opencl were verified with host programs developed in Python that contextualized and communicated with the kernels through PyopenCL. This allowed us to design and verify our OpenCL implementation to use as a reference model in each of the developed layers and in each of the structures that linked them (fires, blocks, and the complete network) with the PyTorch solution. This verification was performed with test benches developed in Jupyter notebooks at both the emulation level, the cosimulation level with an HDL simulator, and the physical level in the FPGA itself.

In this way, we could debug the execution of the host and the implementation of the kernel at the same time since:

- The SqueezeNet model is easily instantiated, thanks to PyTorch, making it a good reference model.
- The results of each node of the PyTorch model can be converted to NumPy arrays, facilitating a layer-by-layer comparison for design debugging.
- The host implementation is more flexible and user-friendly via PyOpenCL.

The first architecture that is described allows us to know the basic algorithm needed for the different layers of the SqueezeNet and present an implementation based on simple-task kernels. In the next two, we use systolic implementations to apply the concepts of "task-parallelism" available in HLS, such as OpenCL. For each of these architectures, we establish comparisons in resources used and performance with the hardware architectures that have so far offered the best performance and that are based on NDRange kernels with the same board. We also establish a comparison with solutions developed on the DE10 nano board but resorting only to the combined use of dual-core ARM Cortex-A9 plus fpu NEON by using the PyTorch Python package.

3.1. Simple Task Architecture

3.1.1. Convolution 1 × 1

We can see in Algorithm 1 the equations involved in convolution 1 × 1 and in Figure 6 a graphic detail for the first iteration of the loop contained between lines 3–13 of this algorithm. As we can see, the fact that the pointers are not performing operations considerably speeds up execution time. Moreover, the only method that manages to considerably optimize the loop's pipeline is the implementation of a shift register, with version 6 being the best-optimized kernel of the 1 × 1 convolution layer.

Algorithm 1 Convolution 1 × 1

```
Input: $in\_channels$ (parameter)
Input: $filte\_size$ (parameter)
Input: $in\_size$ (parameter)
Input: $in\_img$ (buffer read only)
Input: $filter\_weight$ (buffer read only)
Input: $filter\_bias$ (buffer read only)
Output: $out\_img$ (buffer)
1:  for $filter\_index \leftarrow 0$ To $filter\_size$ do
2:      $bias \leftarrow filter\_bias[filter\_index]$
3:      for $j \leftarrow 0$ To $(in\_size \times in\_size)$ do
4:          $tmp \leftarrow bias$
5:          for $k \leftarrow 0$ To $(in\_channels)$ do
6:              $tmp \leftarrow in\_img[k \times in\_size \times in\_size + j] \times filter\_weight[k + filter\_index \times in\_channels] + tmp$ ;
7:          end for
8:          if $tmp > 0$ then
9:              $out\_img[ij + in\_size \times in\_size \times filter\_index] \leftarrow tmp$
10:         else
11:             $out\_img[ij + in\_size \times in\_size \times filter\_index] \leftarrow 0$
12:         end if
13:     end for
14: end for
```

We can see in Table 3 different implementations of the kernel corresponding to the 1 × 1 convolution. We discarded version 2 due to the resources needed to implement it and its high ii. Although it is the version with the shortest execution time, it is not compatible with the synthesis of the kernel with all SqueezeNet layers.

In Table 4, we find the results obtained from the synthesis of the hardware of the 1 × 1 convolution (V6) with the inference of a shift register. Again, we obtain the execution time required for the execution of a 1 × 1 convolution with an input feature map of 13 × 13 × 512 with 1000 filters.

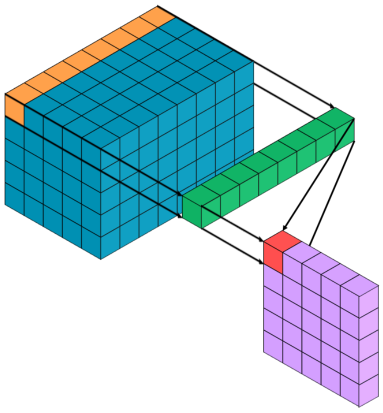

Figure 6. Detail of convolution 1 × 1 with $CH_{in} = 8$, $CH_{out} = 1$, and $in_{size} = 5$.

Table 3. Results of execution times and initial interval, ii, of different versions of the *kernel* conv1 × 1.

	Latency (s)	**Initial Interval (ii)**
V0	3.4381	16
V1	1.7096	16
V2	1.0423	31
V5	2.4772	16
V6	1.2000	1

Table 4. Compilation and execution results of conv1 × 1 V6.

	Single Task	**NDRange [16]**	**PyTorch**
ALUTs	16,421	11,536	-
Registers	28,536	21,064	-
Logic resources	12,413/41,910 (30%)	6962/41,910 (24%)	-
DSP blocks	9/112 (8%)	20/112 (18%)	-
Memory bits	351,936/5,662,720 (6%)	1,092,608/5,662,720 (19%)	-
RAM blocks	101/553 (18%)	168/553 (30%)	-
Fmax (MHz)	116.13	120.81	-
Latency (s)	1.2000	0.3689	0.1823

3.1.2. Convolution 3 × 3

We can see in Algorithm 2 the equations involved in convolution 3 × 3 and in Figure 7 a graphic detail for first iteration of the loop contained between lines 5-23 of this algorithm.

The first thing that stands out about this implementation and the Ndrange [16] with which it is compared is that they are implementations in which the filter size is fixed (3 × 3), as well as the padding (1) and stride (1).

The results obtained from synthesizing the hardware of the 3 × 3 convolution with shift register inference are shown in Table 5, even though it is not in the final OpenCL code. The execution time required for the execution of a 3 × 3 convolution with a 13 × 13 × 64 input feature map with 256 kernels is also shown. This corresponds to a single-layer *expand* configuration of 7 or 8 fire modules.

Algorithm 2 Convolution 3 × 3

```
Input: $in\_channels$ (parameter)
Input: $in\_size$ (parameter)
Input: $pad$ (parameter)
Input: $stride$ (parameter)
Input: $out\_size$ (parameter)
Input: $filter\_size$ (parameter)
Input: $in\_img$ (buffer read only)
Input: $filter\_weight$ (buffer read only)
Input: $filter\_bias$ (buffer read only)
Output: $out\_img$ (buffer)
1:  $out\_img \leftarrow start\_channel \times out\_size \times out\_size + out\_img$
2:  for $filter\_index \leftarrow 0$ To $filter\_size$ do
3:      $bias \leftarrow filter\_bias[filter\_index]$
4:      for $i \leftarrow 0$ To $(out\_size)$ do
5:          for $j \leftarrow 0$ To $(out\_size)$ do
6:              $tmp \leftarrow bias$
7:              for $k \leftarrow 0$ To $(in\_channels)$ do
8:                  for $l \leftarrow 0$ To 3 do
9:                      $h \leftarrow i \times stride + l - pad$
10:                     for $m \leftarrow 0$ To 3 do
11:                         $w \leftarrow j \times stride + m - pad$
12:                         if $(h >= 0)\&(h < in_size)\&(w >= 0)\&(w < in_size)0$ then
13:                             $tmp \leftarrow in\_img[k \times in\_size \times in\_size + h \times in_size + w]$
                                  $\times filter\_weight[9 \times k + 3 \times l + m] + tmp$
14:                         end if
15:                     end for
16:                 end for
17:             end for
18:             if $tmp > 0$ then
19:                 $out\_img[i \times out\_size + j] \leftarrow tmp$
20:             else
21:                 $out\_img[i \times out\_size + j] \leftarrow 0$
22:             end if
23:         end for
24:     end for
25:     $filter\_weight \leftarrow input\_channels \times 9 + filter\_weight$
26:     $out\_img \leftarrow out\_size \times out\_size + out\_img$
27: end for
```

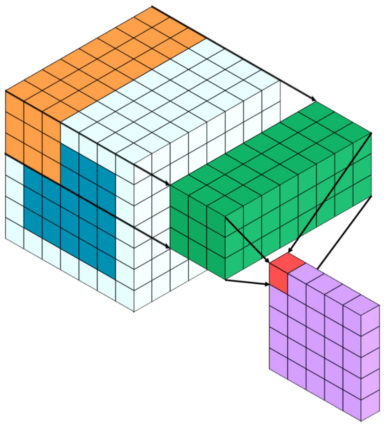

Figure 7. Detail of Convolution 3 × 3 with $CH_{in} = 8$, $CH_{out} = 1$, $in_{size} = 5$, pad =1, and stride = 1.

Table 5. Compilation and execution results of kernel conv3 × 3.

	Single Task	NDRange [16]	PyTorch
ALUTs	30,109	255,353	-
Registers	52,305	38,286	-
Logic resources	23,699/41,910 (57%)	20,028/41,910 (48%)	-
DSP blocks	29/112 (26%)	29/112 (26%)	-
Memory bits	982,288/5,662,720 (17%)	2,831,104/5,662,720 (50%)	-
RAM blocks	202/553 (37%)	403/553 (73%)	-
Fmax	104.85	113.23	-
Latency (s)	0.0723	0.0149	0.1795

3.1.3. Max-Pool

We can see in Algorithm 3 the equations involved in the max-pool layer and in Figure 8 a graphic detail for the first iteration of the loop contained between lines 3–14 of this algorithm in two different channels. This implementation works with a fixed filter size (3 × 3), with a padding of 0, and a stride of 1. Therefore, it must work with out_{size} according to the dimensions of in_{size} and these prefixed parameters.

In this section, the resources of DE10-nano are visualized after compilation with the OpenCL HLS. The results presented in Table 6 correspond to the execution of the kernel that implements the max-pool layer with the maxpool1 configuration of the SqueezeNet v1.1 architecture.

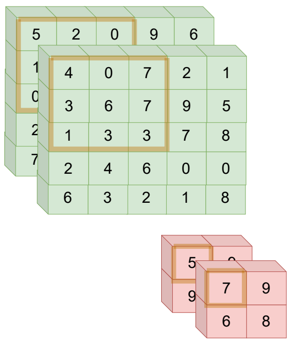

Figure 8. Detail of max-pool layer with $CH_{size} = 2$, $in_{size} = 5$ and $out_{size} = 2$, pad = 0, and stride = 2.

Algorithm 3 Max-pool

```
Input: $in\_size$ (parameter)
Input: $out\_size$ (parameter)
Input: $channel\_size$ (parameter)
Input: $in\_img$ (buffer read only)
Output: $out\_img$ (buffer)
 1: for $channel\_index \leftarrow 0$ To $channel\_size$ do
 2:     for $i \leftarrow 0$ To $(out\_size)$ do
 3:         for $j \leftarrow 0$ To $(out\_size)$ do
 4:             $tmp \leftarrow 0$
 5:             for $l \leftarrow 0$ To $3$ do
 6:                 for $m \leftarrow 0$ To $3$ do
 7:                     $value \leftarrow in\_img[(i \times 2 + l) \times in\_size + j \times 2 + m]$
 8:                     if $value > tmp$ then
 9:                         $tmp \leftarrow value$
10:                     end if
11:                 end for
12:             end for
13:             $out\_img[i \times out\_size + j] \leftarrow tmp$
14:         end for
15:     end for
16:     $in\_img \leftarrow in\_img \times in\_img + in\_img$
17:     $out\_img \leftarrow out\_size \times out\_size + out\_img$
18: end for
```

Table 6. Compilation and execution results of the kernel max-pool.

	Single Task	**NDRange [16]**	**PyTorch**
ALUTs	6274	6318	-
Registers	10,757	12,317	-
Logic resources	5432/41,910 (13%)	6032/41,910 (14%)	-
DSP blocks	8/112 (7%)	12/112 (11%)	-
Memory bits	183,644/5,662,720 (3%)	234,864/5,662,720 (4%)	-
RAM blocks	68/553 (12%)	58/553 (10%)	-
Fmax	122.24	118.51	-
Latency (s)	0.0308	0.0642	0.0947

3.1.4. Average-Pool

We can see in Algorithm 4 the equations involved in the average-pool layer and in Figure 9 a graphic detail for two iterations of the loop contained between lines 1–14 of this algorithm. The Table 7 presents the results obtained by the OpenCL code synthesis of the average-pool layer, taking into account that the filter size is fixed (13 $\times$ 13) and coincides with in_{size}. This layer corresponds to the last stage of the selected architecture and gives us the result of the classification.

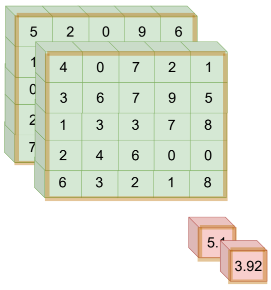

Figure 9. Detail of average-pool with $CH_{size} = 2$, $in_{size} = 5$, and $out_{size} = 2$.

Algorithm 4 Average-pool

```
Input: $in\_img$ (buffer read only)
Output: $out\_img$ (buffer)
 1: for $class\_index \leftarrow 0$ To 1000 do
 2:     $tmp \leftarrow 0$
 3:     for $i \leftarrow 0$ To 169 do
 4:         $tmp \leftarrow bias$
 5:         for $i \leftarrow 0$ To $(in\_channels)$ do
 6:             $tmp \leftarrow in\_img[169 \times class\_index + i] + tmp$ ;
 7:         end for
 8:         $out\_im[class\_index] \leftarrow tmp/169$
 9:     end for
10: end for
```

Table 7. Compilation and execution results of the average-pool kernel.

	Single Task	**NDRange [16]**	**PyTorch**
ALUTs	6178	6045	-
Registers	9303	8492	-
Logic Resources	4736/41,910 (11%)	4633/41,910 (11%)	-
DSP blocks	4/112 (4%)	4/112 (4%)	-
Memory bits	182,798/5,662,720 (3%)	209,162/5,662,720 (4%)	-
RAM blocks	59/553 (11%)	49/553 (9%)	-
Fmax	133.16	117.28	-
Latency (s)	0.0028	0.0034	0.0057

3.1.5. Complete Model

If we join the four kernels described above and implement the complete network described in Figure 5, we can see in Table 8 how the lower efficiency achieved in the 1×1 and 3×3 convolutions makes the continuous reuse of these kernels result in a much lower performance than other hardware solutions referenced and our software implementations based on PyTorch libraries.

In terms of resources used, the solution is comparable to other solutions with a performance four times higher. Thus, despite working with a simple-task solution that is, in principle, more efficient with FPGA devices, the reality is that the results are disappointing.

3.2. Systolic Unroll Architecture

Fundamental contributions in these two areas have already been provided by our group in this architecture, as follows.

1. Systolic generation

- The first contribution is the implementation of the convolution layers in pairs rather than individually. With this technique, we can improve the communication between layers by performing it internally in the IP developed through channels. The SqueezeNet fire processing unit (SFPU) is designed to facilitate the integration of the squeeze layer and the expand layer as a single entity. This implementation is intended to be versatile enough to be used multiple times to implement the various fire units that exist in the convolutional network (see Table 2).
- The SFPU should be fractionable so that we can implement 3 × 3 and 1 × 1 convolutions in isolation when necessary (see layers conv1 and conv3 in Table 2). This implementation should increase the possibilities available in the published solutions. It should support filter sizes other than 3 × 3 and with configurable padding and stride.
- The systolic architecture must also be flexible enough to implement the Max-pool and Average-pool layers with configurable filter sizes and characteristics.
- Finally, it must be compatible with the possibility of implementing dense layers, even if SqueezeNet does not have this type of layer. With this approach, we could solve the final stages of CNN (LeNet, AlexNet, VGGNet).

2. Implementation of the systolic architecture using OpenCL
    - Use of a system of task implementation, in which the projected processing units are assumed by a function with the property autorun kernels. From now on, this element is referred to as the projected processing element (PE).
    - Use of functions for data entry and data extraction based on two techniques: buffers and queues.
    - Use of two control units from which it distributes its control signals to the remaining kernels.
    - Use of control flow and data flow through channels (exclusive to IntelFPGA OpenCL) that play a key role in the synchronization of all processing units.

Table 8. Compilation results and executions of the full OpenCL code implementing the SqueezeNet v1.1 model. A comparison is also made with previous work.

	Single Task Ours	**PyTorch Ours**	**NDRange [16]**	**ZynqNet [11]**	**Edgenet [12]**	**Int8 [13]**	**SqJ [14]**
FPGA	Cyclone V	Cyclone V	Cyclone V	Zynq 7000	Cyclone V	Cyclone V	Zynq 7000
ALUTs	46,615	-	446,951	-	-		
Registers	826,347	-	736,337	137k	-	-	306,554
Logic resources	37k/42k (87%)	-	37k/42k (87%)	154k/218k (70%)	-	110k	
DSP blocks	50/112 (45%)	-	65/112 (58%)	739/900 (82%)	-	-	186/200 46%
Memory bits	1503k/5662k (27%)	-	4096k/5662k (72%)	-	-	-	
RAM blocks	383/553 (69%)	-	553/553 (100%)	996/1090 (91%)	-	-	
Fmax (MHz)	106.6	-	103.07	200	100	101.7	100
Latency fire (ms)	389	140	74	-	-	-	
Latency block 3 (ms)	1203	379	331	-	-	-	
Global latency (ms)	4484	1231	1012	977	110	121	333

The architecture of the implementation is shown in Figure 10. The vertical processing units (PE) are basically in charge of the squeeze stage, and the horizontal processing units

(PE) are in charge of the expand stage. However, the latter has a higher complexity and can realize the average-pool and max-pool layers.

This structure is reused 13 times for the complete network implementation (Figure 11). The buffers (red blocks) that communicate with the producer and consumer blocks visible in Figure 10 are fundamental. The efficiency of the communication between them and the device kernels and the global RAM of the DE10-nano is critical.

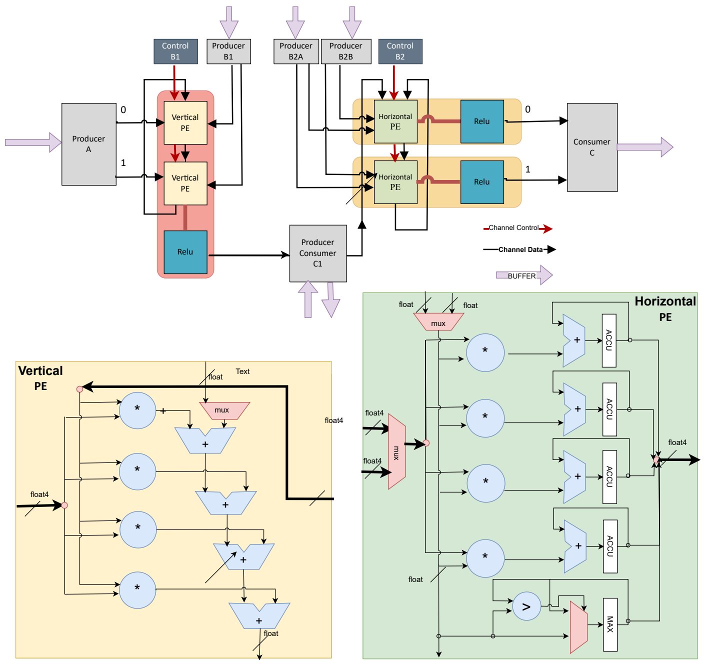

Figure 10. Systolic unroll architecture.

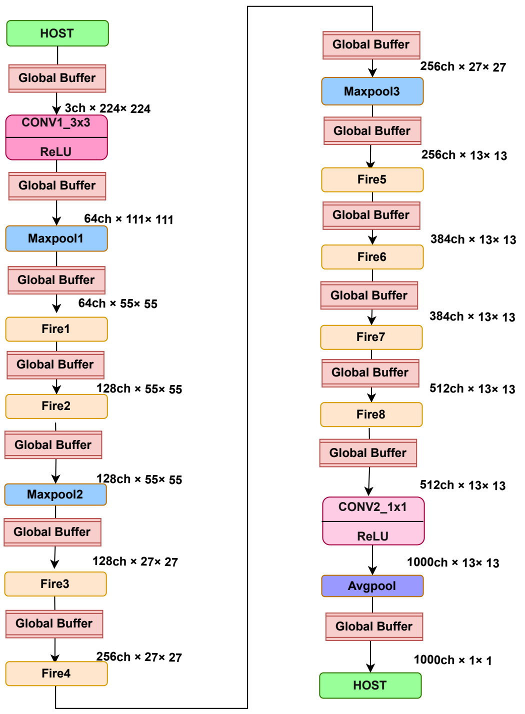

Figure 11. Organization of the reuse of the unroll architecture for SqueezeNetv1.1.

3.3. Systolic Roll Architecture

The folded architecture we describe in Figure 12 is very similar to the previous one, although a better efficiency is achieved in communication between Fire1–Fire2, between Fire3–Fire4, between Fire5–Fire6, and between Fire7–Fire8, as this is now done through channels between kernels. Of course, the control is more complicated, but this type of folding may allow us in the future to have more possibilities with the implementation of dense layers. We can clearly see how communication improves in Figure 13. On the other hand, the unroll architecture is more adaptable than the roll architecture as it does not necessitate that the numbers of vertical and horizontal processing units be the same. This is because it does not involve looping between outputs and inputs.

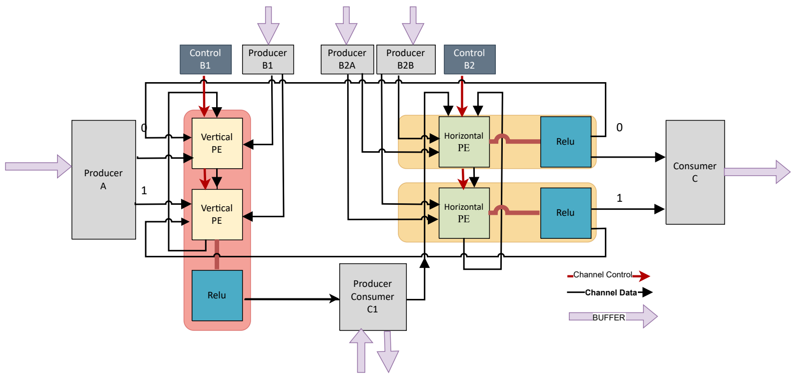

Figure 12. Systolic roll architecture.

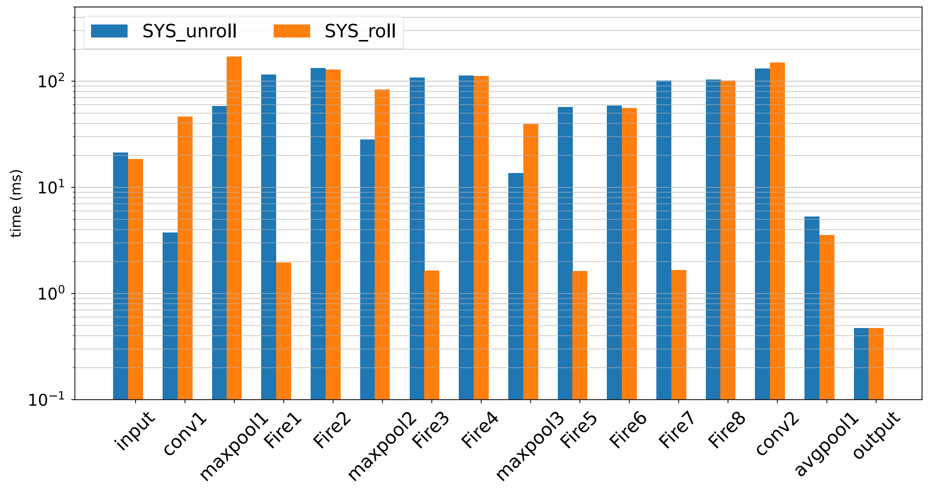

Figure 13. Fire level comparison between roll version and unroll version.

3.3.1. Optimization: Use of RTL Libraries

Our group has developed our own libraries to implement basic operations (output = A × B + C and Accu = A × B + Accu) for different technologies (Cyclone V and Arria 10). These libraries have been very effective in replacing the default implementation by the OpenCL compiler ([17,18]). Therefore, we also used them in this SqueezeNet implementation. The results can be seen in Figure 14, and there is no doubt that our low-level implementations significantly improve the results when applied in deep networks. This figure focuses on the implementation of the convolutions in which we used these functions. As the network depth increases, this optimization clearly reflects the improvement achieved, as shown in Figure 15.

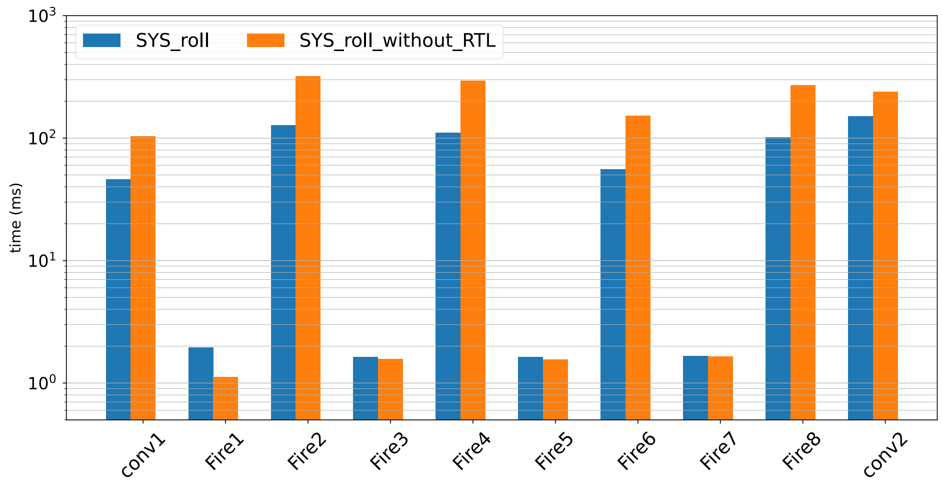

Figure 14. Fire level comparison between roll version without RTL and roll version with RTL.

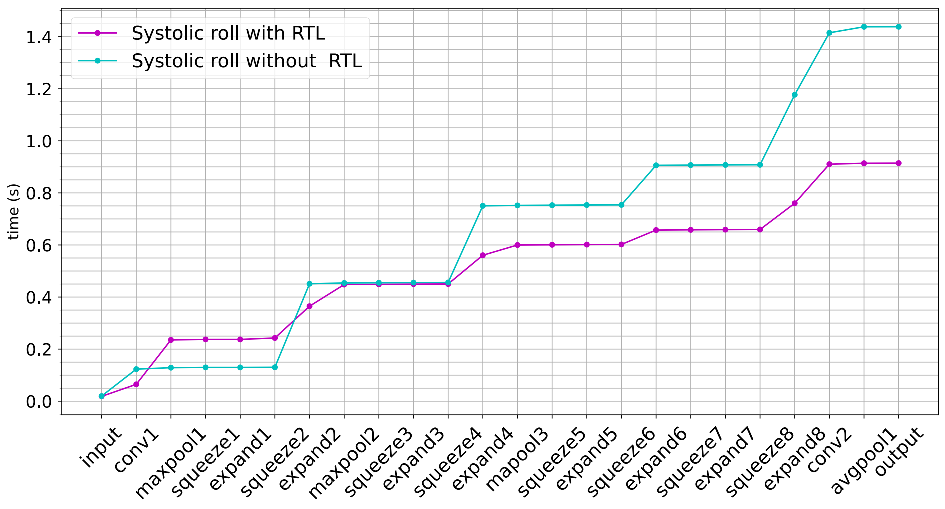

Figure 15. Comparison of latency using RTL libraries and not using RTL libraries.

3.3.2. Optimization: Use of Heterogeneous Solutions

A closer look at Figure 13 shows a degradation in the implementation of the max-pool layers when we made the architectural change between the roll version and the unroll version. This circumstance, together with the need to reduce the number of resources used in the FPGA, led to the implementation of the max-pool layer using a simple-task kernel and the implementation of average-pool by the host through PyTorch. Transfers between the host and FPGA kernels are not feasible solutions inside a deep network, but they are feasible at the beginning and the end of the network since both ends are in the scope of the host. The consequence of this optimization appear in the results as the mixed architecture.

3.3.3. Optimization: Communication of Data

We performed all communications between the host and the kernel (through buffers) and between the kernels (via channels) with data in the IEEE 754 single-precision format. If we replace this data communication with the IEEE 754 half-precision format, we can increase the efficiency of our implementation considerably because we can double the multiplication-and-add units of our processing units (PE). Let us say that we can increase our processing capacity (equally realized with the IEEE 754 single-precision format) at the cost of data communication accuracy.

4. Results and Performance Evaluation

4.1. Performance Evaluation

In AI-at-the-edge applications, the latency of the system and consequently the throughput of the application are usually the most sought-after performance indicators.

It is not unexpected that the number of frames per second (FPS) is a widely used metric for evaluating the effectiveness of computer vision applications. This measure was used in the present study to compare our results with those of other studies. Nevertheless, it is essential to remember that this parameter is heavily dependent on the network topology, which includes the input layer that would have the size of the original image.

It is more intriguing and does not depend on the topology to determine the throughput as a function of the number of network parameters processed per second, as it is generally accepted that each synaptic connection requires one parameter (one MAC operation).

It should be specified whether these operations are performed in fixed or floating point and the exact type of representation used. We can assess the effect on accuracy at different levels, but it will always depend on the application used.

The importance of energy efficiency is especially relevant when AI-at-the-edge processing is performed on embedded devices with limited battery capacity [19]. This is usually measured in terms of the number of operations per joule, but in this work, mW/Mps, or energy per operation, was used. This is a significant value when comparing different manufacturers. The only issue with this metric is determining how the power has been evaluated (estimated or measured); the values found in many studies are estimates of the design environments with FPGAs (Vivado or Quartus), and it can often be difficult to ascertain what has been done with the power consumed by the processor system (PS) in systems-on-chips (SOCs).

Finally, it is essential to consider the flexibility and architectural adaptation of our implementation. Other implementations may be superior in terms of metrics but require a time-consuming hardware recompilation for any topology change. Additionally, implementations that depend on group size (such as NDrange kernel-based OpenCL implementations) may be efficient with a batch of input vectors, but this is not a realistic situation in inference. Therefore, our implementation is advantageous in terms of flexibility and architectural adaptation.

4.2. Resources

We show in Table 9 the resources used in each of the developed architectures. It is not important as an element of comparison since what is really important is that we can implement it on the DE10-nano device. However, it is useful as a reference element when we want to compare the performance of each solution in the following subsection.

Table 9. Compilation results of the full OpenCL code implementing the SqueezeNet v1.1 model.

	Unroll Architecture (Section 3.2)	**Roll Architecture (Section 3.3.1)**	**Mix Architecture (Section 3.3.2)**	**Mix-Hybrid Architecture (Section 3.3.3)**
ALUTs	48,454	42,878	44,596	58,680
Registers	75,051	66,240	69,582	72,497
Logic resources	37,463/41,910 (89%)	31,682/41,910 (87%)	32,426/41,910 (77%)	37,656/41,910 (90%)
DSP blocks	57/112 (51%)	61/112 (54%)	61/112 (54%)	77/112 (69%)
Memory bits	3,670,872/5,662,720 (65%)	3,627,956/5,662,720 (64%)	2,795,292/5,662,720 (49%)	1,764,372/5,662,720 (31%)
RAM blocks	553/553 (100%)	553/553 (100%)	502/553 (91%)	393/553 (71%)
Fmax (MHz)	108.31	98.39	97.05	95.43

4.3. Performance

At this point, we show the speed results achieved by our proposed systolic architectures. For this purpose, we show the complete run-time of the SqueezeNet network version v1.1 (Figure 16) and a detail of the execution times of each phase of the network (Figure 17). In each of the two figures, we also include for comparison a pure software solution developed with PyTorch (dual-core ARM Cortex-A9 plus fpu NEON) on the same device and a hardware solution based on NDrange from those referenced in the literature and using the same device.

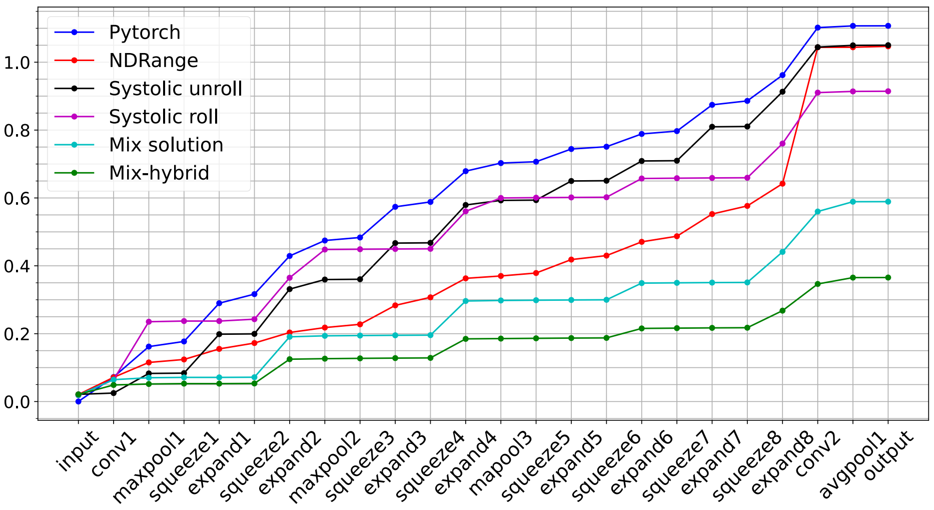

Figure 16. Comparison of full network inference with accumulated times.

It can be seen for each of the proposed architectures where the possible points of improvement are located. For example, in our best implementation (mix solution), the bottlenecks are found in the expand layers of even fire blocks, together with the final 1 × 1 convolution (which we call conv2).

It can also be observed that the final 1 × 1 convolution could also be assumed by the software through PyTorch. Examining the graphs (Figures 16 and 17), it is evident that our approach resulted in a significant improvement compared to prior solutions.

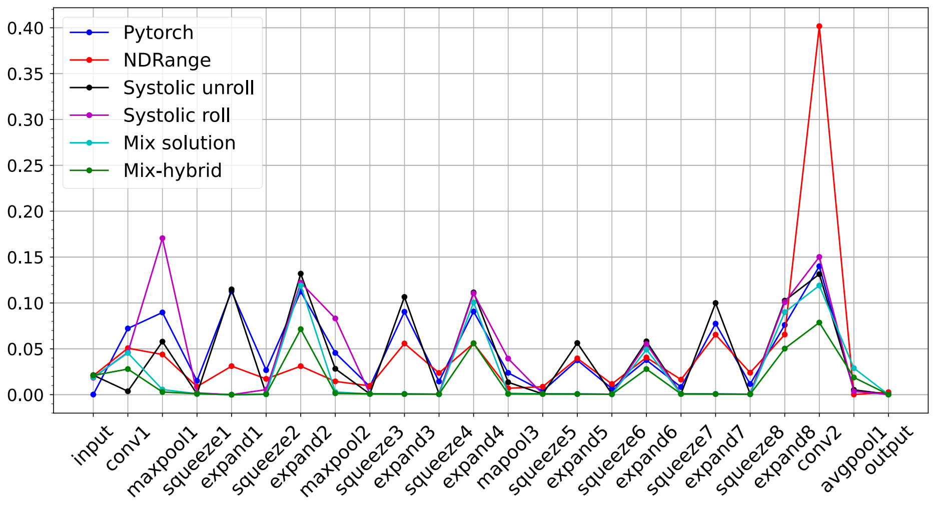

Figure 17. Comparison of full network inference with phase times.

4.4. Energy Efficiency

As we have said before, it is important to make the same comparison in terms of energy efficiency. For this purpose, we include Table 10.

Table 10. Compilation results and executions of the full OpenCL code implementing the SqueezeNet V1.1 model. A comparison is also made with previous work.

	Single Task (Section 3.1)	**NDRange [16]**	**PyTorch**	**Unroll (Section 3.2)**	**Roll (Section 3.3.1)**	**Mix (Section 3.3.2)**	**Hybrid (Section 3.3.3)**
Energy per inference (Joules)	32.4	10.8	3.6	7.2	10.8	5.04	2.88
Power (mw)	7523.55	10,641.33	3716.07	6418.91	7928.37	7626.50	7447.42
mw/Mps	27,427.31	8780.49	2926.83	5853.65	8780.48	4097.56	2341.46

It is evident that the last of our architectures is more energy efficient than implementing the algorithm solely with the software solution (PyTorch). However, from an application perspective, the execution time is of great importance, and our systolic architectures implemented in OpenCL have enabled us to reduce these execution times. This energy efficiency study raises doubts about the suitability of many hardware–software codesign solutions from an energy efficiency standpoint in low-cost devices.

4.5. Discussion

The HPSs available in this type of resource-limited devices provide very high performance thanks to the use of the vector accelerators they have implemented (Neon in this case) and hardly justify the use of the FPGA part to perform additional acceleration except when we need to free the microprocessor for other tasks. We managed to improve energy efficiency with our kernels but at the cost of no longer having resources in the FPGA that might be necessary for the generation and acquisition of signals needed for a dedicated instrument.

We would like to emphasize the effectiveness of HLS compilers. We have discovered that incorporating our own RTL libraries into them can significantly enhance their performance. Additionally, we have noticed that the more granular the system (which leads to a greater number of kernels that must be connected through channels), the more the efficiency of dataflow systems decreases. The solution may lie in the communication system between kernels (channels, pipes, or streaming interfaces); however, it does not appear that they can be tailored or designed at the RTL level.

5. Conclusions

This paper presented a workflow for the implementation of deep neural networks that combined the flexibility of HLS-based networks with the architectural control features of HDL-based flows. OpenCL was the main tool used in this workflow, as it provides a structural approach and a high level of hardware control, which is especially important when dealing with systolic architectures.

From a verification point of view, the proposed method based on Pyopencl–Jupyter Notebook is effective since the reference models in all tests, both for the individual layers and the complete network, have been readily available using packages well known in the development, training, and inference of deep networks (PyTorch).

From a results point of view, we managed to lower the inference of SqueezeNet v1.1 by 0.4 s working on all calculations in single-precision float (32 bits), and we managed to outperform in energy efficiency the results measured in solutions that did not use the programmable logic infrastructure available in the package.

Of course, the results are easily surpassed when working with more resource-intensive devices, but they become unfeasible in AI-at-the-edge solutions. We have seen that in deep networks, the only solution is the continuous reuse of a flexible architecture, and the intermediate step through global memories is absolutely necessary. This reuse, together with the intervention of the global memory, considerably reduces the implementation results of deep networks with respect to shallow networks [17].

The importance of the relationship between layers in deep neural networks is clear and must be thoroughly examined. Our next steps will be to enhance the interlayer transfers by improving the interconnection between the FPGA and the SDRAM connected to the hard processor system (HPS).

Author Contributions: Conceptualization, J.F.; Methodology, R.G.-G. and J.L.R.-R.; Validation, J.L.R.-R.; Investigation, R.G.-G. and J.L.R.-R.; Resources, J.M.M.; Writing—original draft, R.G.-G.; Writing—review & editing, J.M.M.; Supervision, J.F. and J.M.M. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by grant PID2020-116816RB-100 from Ministry of Science, Innovation and Universities (MCIU) of Spain.

Data Availability Statement: Data is contained within the article.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Singh, R.; Gill, S.S. Edge AI: A survey. *Internet Things Cyber-Phys. Syst.* **2023**, *3*, 71–92. [CrossRef]
2. Iandola, F.N.; Han, S.; Moskewicz, M.W.; Ashraf, K.; Dally, W.J.; Keutzer, K. SqueezeNet: AlexNet-level accuracy with 50× fewer parameters and <0.5 MB model size. *arXiv* **2016**, arXiv:1602.07360. Available online: http://arxiv.org/abs/1602.07360 (accessed on 10 May 2023).
3. Lee, H.J.; Ullah, I.; Wan, W.; Gao, Y.; Fang, Z. Real-Time Vehicle Make and Model Recognition with the Residual SqueezeNet Architecture. *Sensors* **2019**, *19*, 982. [CrossRef] [PubMed]
4. Kwaghe, O.P.; Gital, A.Y.; Madaki, A.; Abdulrahman, M.L.; Yakubu, I.Z.; Shima, I.S. A Deep Learning Approach for Detecting Face Mask Using an Improved Yolo-V2 With Squeezenet. In Proceedings of the 2022 IEEE 6th Conference on Information and Communication Technology (CICT), Gwalior, India, 18–20 November 2022; pp. 1–5. [CrossRef]
5. Fang, C.; Lv, C.; Cai, F.; Liu, H.; Wang, J.; Shuai, M. Weather Classification for Outdoor Power Monitoring based on Improved SqueezeNet. In Proceedings of the 2020 5th International Conference on Information Science, Computer Technology and Transportation (ISCTT), Shenyang, China, 13–15 November 2020; pp. 11–15. [CrossRef]
6. Zhang, J.; Zhu, H.; Wang, P.; Ling, X. ATT Squeeze U-Net: A Lightweight Network for Forest Fire Detection and Recognition. *IEEE Access* **2021**, *9*, 10858–10870. [CrossRef]
7. Tsalera, E.; Papadakis, A.; Voyiatzis, I.; Samarakou, M. CNN-based, contextualized, real-time fire detection in computational resource-constrained environments. *Energy Rep.* **2023**, *9*, 247–257. [CrossRef]
8. Yang, Z.; Yang, X.; Li, M.; Li, W. Automated garden-insect recognition using improved lightweight convolution network. *Inf. Process. Agric.* **2023**, *10*, 256–266. [CrossRef]
9. Huang, Q. Weight-Quantized SqueezeNet for Resource-Constrained Robot Vacuums for Indoor Obstacle Classification. *AI* **2022**, *3*, 180–193. [CrossRef]
10. Team, P. Pytorch Vision SQUEEZENET Model. Available online: https://pytorch.org/hub/pytorch_vision_squeezenet (accessed on 1 April 2023).
11. Gschwend, D. ZynqNet: An FPGA-Accelerated Embedded Convolutional Neural Network. *arXiv* **2020**, arXiv:2005.06892. Available online: http://arxiv.org/abs/2005.06892 (accessed on 1 April 2023).
12. Pradeep, K.; Kamalavasan, K.; Natheesan, R.; Pasqual, A. EdgeNet: SqueezeNet like Convolution Neural Network on Embedded FPGA. In Proceedings of the 2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS), Bordeaux, France, 9–12 December 2018; pp. 81–84. [CrossRef]
13. Zhao, J.; Yin, Z.; Zhao, Y.; Wu, M.; Xu, M. Scalable FPGA-Based Convolutional Neural Network Accelerator for Embedded Systems. In Proceedings of the 2019 4th International Conference on Computational Intelligence and Applications (ICCIA), Nanchang, China, 21–23 June 2019; pp. 36–40. [CrossRef]
14. Mousouliotis, P.G.; Petrou, L.P. SqueezeJet: High-Level Synthesis Accelerator Design for Deep Convolutional Neural Networks. In Proceedings of the Applied Reconfigurable Computing. Architectures, Tools, and Applications, Santorini, Greece, 2–4 May 2018; Voros, N., Huebner, M., Keramidas, G., Goehringer, D., Antonopoulos, C., Diniz, P.C., Eds.; Springer Nature: Cham, Switzerland, 2018; pp. 55–66. [CrossRef]
15. Arora, M.; Lanka, S. Accelerating SqueezeNet on FPGA. Available online: https://lankas.github.io/15-618Project (accessed on 20 September 2023).

16. Er1cZ. Deploying_CNN_on_FPGA_using_OpenCL. 2017. Available online: https://github.com/Er1cZ/Deploying_CNN_on_FPGA_using_OpenCL (accessed on 1 January 2023).
17. Gadea-Gironés, R.; Fe, J.; Monzo, J.M. Task parallelism-based architectures on FPGA to optimize the energy efficiency of AI at the edge. *Microprocess. Microsyst.* **2023**, *98*, 104824. [CrossRef]
18. Gadea-Gironés, R.; Herrero-Bosch, V.; Monzó-Ferrer, J.; Colom-Palero, R. Implementation of Autoencoders with Systolic Arrays through OpenCL. *Electronics* **2021**, *10*, 70. [CrossRef]
19. Sze, V.; Chen, Y.H.; Yang, T.J.; Emer, J.S. How to Evaluate Deep Neural Network Processors: TOPS/W (Alone) Considered Harmful. *IEEE Solid-State Circuits Mag.* **2020**, *12*, 28–41. [CrossRef]