On how to efficiently implement Deep Learning algorithms on PYNQ platform

Luca Stornaiuolo, Marco D. Santambrogio, Donatella Sciuto

Politecnico di Milano, Dipartimento di Elettronica Informazione e Bioingegneria (DEIB), Milan, Italy
{luca.stornaiuolo, marco.santambrogio, donatella.sciuto}@polimi.it

***Abstract*—Deep Learning algorithms are gaining momentum as main components in a large number of fields, from computer vision and robotics to finance and biotechnology. At the same time, the use of Field Programmable Gate Arrays (FPGAs) for data-intensive applications is increasingly widespread thanks to the possibility to customize hardware accelerators and achieve high-performance implementations with low energy consumption. Moreover, FPGAs have demonstrated to be a viable alternative to GPUs in embedded systems applications, where the benefits of the reconfigurability properties make the system more robust, capable to face the system failures and to respect the constraints of the embedded devices. In this work, we present a framework to efficiently implement Deep Learning algorithms by exploiting the PYNQ platform, recently released by Xilinx. The case study application is tested on PYNQ-Z1 board, commonly used in embedded system applications.**

I. Introduction

Deep Neural Networks (DNNs) are a set of algorithms inspired from brain neurons behavior used to recognize patterns. They are becoming a pervasive solution in a huge amount of fields, from computer vision, smart vehicles and robotics to finance, medicine and decision making. In these domains, Machine Learning approach is proved to be valuable to automate the prediction processes based on the available data and on real-time signals acquisition [1].

This, combined with the widespread of the Internet of Things (IoT) paradigm [2], opens new challenges for embedded system devices. The number of connected objects that acquire signals from the physical world, send data to related applications and get back the information to act, is increasing in quantity and variety and this increases also the whole system complexity. In this context, the embedded system devices have to face many problems, ranging from dealing with the possible system failures to adapting themselves to the physical world changes [3].

One possible solution to solve these challenges is the development of intelligent embedded system devices, able to take care of their own healing and to adapt their behavior accordingly with the environmental variations. Even if Machine Learning algorithms seem to be the right solution to face the system adaptation and its self-healing capabilities, they require a high computational power. Moreover, the embedded devices physical constraints require a low energy consumption, also to reduce the system warm and the possible derived damages [4]. For these reasons, Field-Programmable Gate Arrays (FPGAs) devices represent a suitable architecture to fit such requirements. In addition, their reconfigurable properties make the system more flexible with respect to different contexts, but also adaptable to changes within the same context.

In this paper we present a framework to help data scientists and FPGA-based IPs designers to deploy their deep learning algorithms on a Xilinx Zynq SoC by exploiting the recently released PYNQ platform. In particular, the main contributions of our work are declined as a framework able to:

1) automatically create the required interface to transfer data from the Processing System to the accelerated version of the DNN implemented on the Programmable Logic by following the dataflow paradigm;
2) help in designing the routine to deal with system failures by exploiting the behavior of DNNs and the reconfigurable properties of the system.

The rest of this paper is divided as follow: in Section II and III we describe the relation between DNNs and the Xilinx PYNQ platform and discuss related work; in Section IV we propose our framework; in Section V we discuss about our solution and the future work and Section VI contains the conclusion.

II. Background

In this section, we discuss why FPGAs are a valid solution to improve performances of Deep Learning algorithms and we analyze the advantages of the recently released Xilinx PYNQ platform within the intelligent embedded system context.

A. Deep Neural Networks and FPGAs

DNNs are a subclass of Neural Networks contains multiple hidden layers that propagate weighted sums of input data to the output layer. They also apply a non-linear function to such sums to generate an output only when a certain threshold is crossed. This behavior is inspired from brain neurons and it is used to recognize recurrent patterns between input data and the related output [5].

Although Neural Networks birth dates back to 1940s, in the recent years, the huge amount of available data together with the high compute capacity and the algorithmic techniques improvement have driven DNNs to their today success [6]. In this context, many different hardware platforms have been targeted with the aim to improve DNNs throughput and energy efficiency. Graphics Processing Units (GPUs) are the most

common employed solution to exploit hardware parallelism and execute multiple DNNs multiply-and-accumulate operations at the same time. The huge widespread of GPUs has been achieved also thanks to the investment done by the big technology companies, like NVIDIA, to create a set of frameworks and platforms to raise the level of abstraction required to exploit them. Examples are CUDA[1] and OpenCL[2] that have allowed GPUs integration in Machine Learning frameworks and libraries like Caffe[3], Dlib[4], and TensorFlow[5]. However, GPUs exploitation requires high power consumption and this represents a limitation in many Deep Learning domains where the computational devices are required to be mobile or embedded ad they need to fit rigid physical constraints.

To overcome this problem, FPGAs can largely reduce the power consumption, while remaining competitive in terms of execution time. In particular, we can exploit spatial architecture of FPGAs to take advantage of the memory hierarchy and reduce the energy costs of data accesses. Since the most efficient memory is also the most limited one, it is necessary to design the system to follow the dataflow paradigm and to get benefit from data reuse, taking into consideration that the processing of DNNs is deterministic.

B. PYNQ platform and Intelligent Embedded Systems

PYNQ[6] is a platform built on top of Xilinx Zynq SoC technology to allow software developers exploiting the Programmable Logic of the board directly from Python applications. In this way, FPGA-based IPs designers can provide their optimized functions ready to be used, with the same abstraction level of software libraries. These libraries able to exploit the heterogeneous hardware architecture of the Zynq SoC are also named hardware libraries or overlays. The boards PYNQ-Z1 is the first released board that support the PYNQ platform. The Xilinx official base overlays allows to control the 12-pin PMOD connectors, an Arduino-compatible interface and Audio/Video I/O. The energy efficiency and the flexibility of the Programmable Logic together with the Python productivity and the peripherals control make the PYNQ-Z1 board an efficient solution as embedded device.

With the advent of the Fog Paradigm part of the computation and data preprocessing is offload to the leaf nodes of the system, where signals are collected. The possibility of adding also intelligent mechanisms within this distributed scenario allows each node to learn from external stimuli, adapt to change and make decisions. These capabilities can also be used to face system failures at different hierarchical level. In particular, in the scenario where a leaf node of a fog distributed system is represented by a PYNQ-Z1 board, that acquires raw data and fast preprocesses them before sending results to the next layer of the network, the ability of recognizing failures and dealing with self-healing by reconfiguring the Programmable Logic, prevents performance reduction of the entire system and errors cascade effects.

[1]https://developer.nvidia.com/cuda-zone
[2]https://www.khronos.org/opencl/
[3]http://caffe.berkeleyvision.org/
[4]http://dlib.net/ml.html
[5]https://www.tensorflow.org/
[6]http://www.pynq.io/

III. Related work

In this section, we provide an overview of the related work that regards the DNNs implemented on embedded systems and the frameworks to help with their development.

DNNs are characterized by two different phases. The training phase, used to process already available data and to compute the model weights, requires high computational power and takes a huge amount of time to be computed. On the other hand, the inference phase, used to process the new data and find known patterns, is suitable to be implemented on embedded systems [6]. In this context, different techniques to reduce the complexity are often used [7]. As an example, binary weights are adopted in [8], where PYNQ platform is exploited to take advantages from the high performance with respect to a low power consumption, and in [9], where the integration of TensorFlow with embedded devices is performed. Together with the complexity reduction techniques, a lot of framework to help hardware designer of DNNs are released. As an example, [10] is a framework to efficiently map binarized neural networks to hardware and [11] is used to reduce the FPGA hardware resources.

IV. Proposed Approach

This section describes the main components of our framework to help data scientists and FPGA-based IPs designers to deploy their deep learning algorithms on a Xilinx Zynq SoC by exploiting the recently released PYNQ platform. Firstly, we modeled and generalized the communication from Processing System to Programmable Logic for dataflow applications in order to automatize the interfaces creation to send data to the accelerated version of a DNN and get back results. Then, we provide a routine to recover running system from a specific failure by reconfiguring the FPGA.

A. Dataflow Communication Interface creation for DNN

Input data of a DNN is the set of values representing the information to be analyzed. For instance, in computer vision domain input can be pixels of an image, in speech and language domain input can be an audio wave, in problem solving domain input can be the current state of some game.

As described in Section II, the dataflow paradigm is the best solution to improve the energy efficiency and exploit data reuse, when implementing DNNs on an FPGA. Since the processing of DNNs does not contain randomness, the dataflow design in term of data type and number of values to be exchange between Processing System and Programmable Logic is known in advance. When the FPGA-based DNN developer decides how to use each memory of the available memory hierarchy, he has to take into account that on-chip BRAM is faster and more efficient than DRAM, but it is very

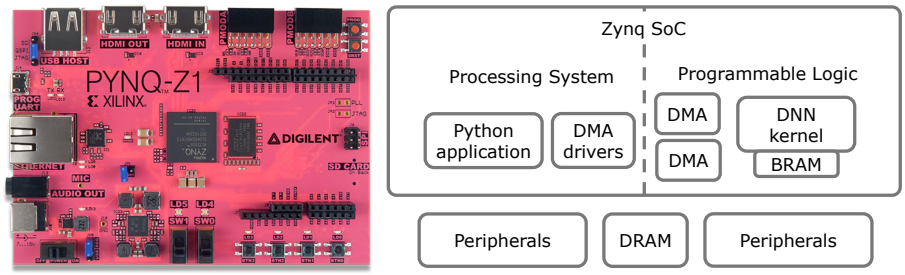

Fig. 1. Main system components overview.

limited. For this reason, the data reuse becomes very important, when some values are transferred to the Programmable Logic from the higher costly DRAM of the board.

Figure 1 shows the main system components. We can divide the framework semi-automatic creation of interfaces in three phases following a bottom-up approach:

1) **Design Level Integration**
Designing the FPGA-based version of DNN for the PYNQ-Z1 board following the dataflow paradigm required the presence of one or more Direct Memory Access (DMA) IPs to stream data from the DRAM to the computational kernel on the Programmable Logic. DMAs can be added in the Vivado block design phase and can be connected to the input interfaces of the user custom IP. To do that the FPGA-based DNN developer has to add an AXI4-Stream interface for each input and output stream of the computational kernel. The framework can auto generate such interfaces by knowing the data type of each stream. The following snippet of Vivado HLS pseudo code shows a very simple example in which the kernel reads a stream of integer points from an input stream and writes the same points to an output stream.

```
#include <hls_stream.h>

struct data_struct{
  int data;
  bool last;
};

void computational_kernel(
        hls::stream<int> &s_in,
        hls::stream<data_struct> &s_out
) {
#pragma HLS INTERFACE axis port=s_in
#pragma HLS INTERFACE axis port=s_out
#pragma HLS INTERFACE ap_ctrl_none port=return

    data_struct out_data;
    for (int i = 0; i < NUMBER_OF_POINTS; i++)
#pragma HLS PIPELINE II=1
         out_data.data = s_in.read();
         if (i == (NUMBER_OF_POINTS - 1))
            out_data.last = 1;
         else
            out_data.last = 0;
            s_out.write(out_data);
      }
}
```

The first two HLS INTERFACE pragmas define the AXI4-Stream type of the kernel parameters. The last HLS INTERFACE pragma allows the computational kernel to start as soon as the data on the input stream are available, without the needing of programmatically start the IP computation. The specific output data structure will be automatically mapped to a set of pins that are able to transfer data and inform the system when the last point is sent, so that the receiver can close the transfer connection. Finally, the NUMBER_OF_POINTS variable depends on the kernel design and in DNN computation is known in advance.

2) **Processing System Level Integration**
The second step of our framework interfaces generation consists of integrating the drivers to command the DMAs directly from the Processing System. We exploit the Python/C API to achieve the fastest possible performance when exposing the drivers functions to the user application. The following snippet of C pseudo code shows how to manage the stream transfers by exploiting two instanced DMAs connected respectively with the input and the output streams of the computational kernel.

```
// ==== OPEN STREAM TO WAIT RESULTS ====
XAxiDma_SimpleTransfer(
        DMAinstance2,
        (uint32_t *) FPGA_receive_buffer,
        NUMBER_OF_POINTS,
        DMA_FROM_FPGA);

// ==== SEND POINTS ====
XAxiDma_SimpleTransfer(
        DMAinstance1,
        (uint32_t *) FPGA_send_buffer,
        NUMBER_OF_POINTS,
        DMA_TO_FPGA);

// ==== WAIT ALL POINTS ARE SENT ====
while (XAxiDma_Busy(DMAinstance1, DMA_TO_FPGA))
    ;
```

```
// ==== WAIT ALL RESULTS ARE RECEIVED ====
while (XAxiDma_Busy(DMAinstance2, DMA_FROM_FPGA))
    ;
```

3) **Python Level Integration**
At the user application level, the Python data types need to be cast to satisfy the requirements of the computational kernel. For this reason, we exploit the NumPy library together with the CFFI Python module to deal with data casting. Moreover, a data verification process is added to avoid that the algorithm is called with the wrong inputs. This avoid computational kernel crashes due to wrong data representations. The following snippet of Python pseudo code shows how to manage the data casting and how to add a possible input format check.

```
def fpga_kernel(input_data):
    if len(input_data) != NUMBER_OF_POINTS:
        raise ValueError("Wrong input dimensions")
    else:
         buff_dma_in = cffi.FFI().cast("int *")
         buff_dma_out = cffi.FFI().cast("int *")

        cffi.FFI().memmove(buff_dma_in, input_data)

        fpga_kernel_driver_interface(
           DMAinstance1,
           DMAinstance2,
           buff_dma_in,
           buff_dma_out
        )

    received_data = cffi.FFI().buffer(buff_dma_out)
    results = frombuffer(received_data, dtype=int32)

        return results
```

B. System Failure Recovery

The inference phase of DNNs, unlike the training one that requires high computational needs and a long execution time, can be part of real-time data processing at the embedded systems level. Taking into consideration the integration with the PYNQ platform explained in the previous section, it is possible to identify at the application level the average execution time of the DNN optimized kernel, given a fixed input/output dimension and fixed data types. This value can be obtained empirically with a calibration phase or can be set as a threshold from a domain expert user. The proposed routine measures the execution time of each computational kernel function call and based on some accuracy thresholds can recognize some kernel errors or interruptions. If an anomaly is detected, the routine kills the current computational kernel function call, reconfigures the Programmable Logic and starts the next input processing. If a list of subsequent anomalies are detected, the routine can generate a system alarm or slightly modify the thresholds to try to understand the failure nature.

V. Discussion and Future Directions

Together with the related work, the solution proposed in this paper represents a starting step to create a solid framework-based infrastructure to integrate FPGAs within the Deep Learning field. Since this process is started earlier for GPUs, nowadays they represent the most used solution, even if other hardware architectures could be integrated at different applications levels, bringing benefits to the whole system.

As future work, we want to allow framework exploiting the partial configuration property of the Programmable Logic to better face system failures. Moreover, we are going to improve the failures predictive model by integrating machine learning techniques in the proposed routine. Finally, we are planning to integrate the framework with the most used Python libraries for Deep Learning models implementation.

VI. Conclusions

In this paper, we presented a framework to integrate Deep Learning algorithms on the PYNQ-Z1 at the embedded system level. In particular the proposed solution help data scientists and hardware developers to generate the required interfaces to interact with the FPGA-based implementation of the DNN algorithm and integrate it within the PYNQ platform. This is done through the generalization of the DNN data management within different domains. Finally, we proposed a routine to face embedded system failure at the Programmable Logic level.

References

[1] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1.

[2] K. Ashton *et al.*, "That 'internet of things' thing," *RFID journal*, vol. 22, no. 7, pp. 97–114, 2009.

[3] G. Ditzler, M. Roveri, C. Alippi, and R. Polikar, "Learning in non-stationary environments: A survey," *IEEE Computational Intelligence Magazine*, vol. 10, no. 4, pp. 12–25, 2015.

[4] C. Alippi, G. Anastasi, M. Di Francesco, and M. Roveri, "Energy management in wireless sensor networks with energy-hungry sensors," *IEEE Instrumentation & Measurement Magazine*, vol. 12, no. 2, 2009.

[5] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[6] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[7] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[8] "Bnn-pynq," https://github.com/Xilinx/BNN-PYNQ/.

[9] "Binary networks from tensorflow to embedded devices," https://github.com/jonathanmarek1/binarynet-tensorflow.

[10] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 65–74.

[11] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.