Pareto optimal design space exploration for accelerated CNN on FPGA

Enrico Reggiani, Marco Rabozzi, Anna Maria Nestorov, Alberto Scolari, Luca Stornaiulo, Marco D. Santambrogio
Politecnico di Milano, Milan, Italy
{enrico2.reggiani annamaria.nestorov}@mail.polimi.it,
{marco.rabozzi, alberto.scolari, luca.stornaiuolo, marco.santambrogio}@polimi.it

***Abstract*—Convolutional Neural Networks (CNNs) are at the base of many applications, both in embedded and in server-class contexts. While Graphics Processing Units (GPUs) are predominantly used for training, solutions for inference often rely on Field Programmable Gate Arrays (FPGAs) since they are more flexible and cost-efficient in many scenarios. However, existing approaches fall short to accomplish several conflicting goals, like efficiently using resources on multiple platforms while retaining deep configurability and allowing a quick Design Space Exploration (DSE) towards the best solution. This paper proposes a solution composed of highly configurable kernels designed for resources time-sharing with an analytical model of their resource/performance characteristics. Building on such models, we propose an Integer Linear Programming (ILP)-based approach to effectively identify pareto optimal kernel configurations in terms of throughput and resource consumption. We evaluate our DSE on two state-of-the-art CNNs, showing how it identifies hundreds of pareto optimal solutions in less than a minute. Guided from the DSE configurations of the AlexNet network, we quickly identified a candidate design for a Xilinx Virtex-7 XC7VX485T FPGA and achieved a peak throughput of 4.05 ms per image, while we measured a maximum estimation error of 6.69% with respect to the proposed analytical models.**

I. Introduction

CNNs have proved to be an effective class of algorithms for features extraction and classification, whose application spread across a wide range of markets, from embedded systems to data centers and High Performance Computing (HPC) systems. To sustain CNNs ever-increasing computation demands, heterogeneous architectures are becoming the main approach, as they offer more parallelism and fine-grained data reuse opportunities with respect to Central Processing Units (CPUs). Accelerators based on GPUs achieve high speedups with respect to CPUs [1] and are largely employed during CNN training, where their performance and programmability allow to quickly iterate through the training phases. Instead, for inference, where the CNN structure is fixed, GPUs power demands become a bottleneck in many scenarios where energy, power, and thermal budgets are limited; these limitations foster the demand for other highly efficient heterogeneous solutions. Recent evaluations have shown several advantages of implementing CNNs on FPGAs [2] and Application Specific Integrated Circuits (ASICs) [3], where the former offer an appealing tradeoff between energy efficiency and programmability, which is especially important where applications frequently evolve over time and the scale makes inflexible ASIC solutions impractical. For these reasons, FPGA-based acceleration of CNNs has become a lively research topic in both the embedded [4] and the HPC domains [5]–[8].

Several aspects are key for implementing CNNs on FPGAs. With respect to hardware resources, *scalability* ensures the design to be easily portable to a large variety of Field Programmable Gate Array devices, whose resources have to be used with maximum *efficiency*. With respect to the system designer, these solutions should have high *configurability* to cover the large range of CNN applications, which vary along several dimensions like network structure, hyperparameters, data types etc. These goals usually open up a very large design space whose exploration can be overly time-consuming, thus causing the final design choices to be sub-optimal; therefore, an ideal solution should also rely on suitable *models* for automated DSE.

To meet these guidelines, this work proposes a comprehensive solution that brings the following contributions:

1) an architecture to accelerate the feature extraction stage of CNNs that focuses on scalability, using time sharing of computing resources to minimize the number of resources used for each convolution
2) a methodology to achieve high design configurability while retaining productivity by using parameterizable Hardware Description Language (HDL) kernels for the compute-intensive parts and High-Level Synthesis (HLS) kernels for the control parts
3) a performance and a resource model to quickly evaluate different architecture configurations
4) an effective pareto optimal DSE to quickly explore the network performance on different target devices

To explain these contributions, the text is organized as follows. Section II reviews the main related work, highlighting state-of-the-art solutions for CNN acceleration to FPGA. Section III details the proposed architecture, discussing the design choices of the proposed kernels and their features. Section IV shows the resource and performance model of these kernels, which is used in section V to perform the DSE and derive the Pareto-optimal configurations for the system. Section VI evaluates our solution with two state-of-the-art CNNs, showing how it effectively targets multiple applications. Finally, section VII discusses achievements, limitations and future works.

II. Related work

This Section describes different design methodologies in the literature to compute CNN inference on FPGA, leveraging both manual and automatic strategies to perform the DSE.

The authors of [4] aim to accelerate state-of-the-art CNN models for large-scale image classification targeting embedded FPGAs. To overcome the off-chip memory bandwidth bottleneck, due to the number of CNN parameters and slow memories, they introduce an automatic data quantization flow and apply a singular value decomposition approach to significantly reduce the fully connected weights footprint. However, when compared with the theoretical estimates, their on-board VGG-SVD benchmark, targeting a Xilinx Zynq XC7Z045, experiences a 47% performance degradation due to limited efficiency of the off-chip memory bandwidth. To alleviate off-chip memory access bottlenecks, [9] proposes an analytical model to balance the performance of computational, on-chip and off-chip resources, providing an optimal match between their usage. To enhance the on-chip memory bandwidth, they also propose a bi-dimensional interconnection between on-chip memory and computational units.

While [4] and [9] employ manual analytical models to perform the CNN DSE, the authors in [5], [10]–[12] propose automatic tool flows to infer the best network configuration. In [10], the authors use a bi-dimensional systolic array pattern to implement an end-to-end automation flow for CNNs targeting FPGAs. The systolic array network is composed of a grid of pipelined processing units, leveraging meshed local connections to better exploit FPGAs routing resources achieving higher working frequencies. This work uses a two-phase DSE. The first phase relies on performance and resource models to identify the best pre-designed template. Whereas, the second phase uses the hardware generation flow of the target device to reach the best performance. Such flow automatically generates the host code and bitstream starting from a user-written code and pragmas. The work is evaluated on an Arria 10 GT 1150 board against AlexNet [13], achieving a throughput of 360.4 GFLOPS, and VGG-16 [14], where they obtained 460.5 GFLOPS for the floating point implementation and 1171.3 GOPS for the fixed point implementation. Zhang et al. [12] propose a DSE technique to identify the optimal architectural solutions, in terms of both performance and resource usage, over a large design space and leveraging on both the roofline model [15] and the Computation to Communication Ratio (CTC). The design space is created by applying to the described network a set of optimizations such as on-chip reuse buffering and loop transformations, e.g. tiling, pipelining and unrolling. Implementing AlexNet on a Xilinx Virtex-485T at 100MHz, this work achieves 1.33GFLOPS.

In [11] the authors implement the 2D Winograd algorithm [16] to improve their FPGA-based CNNs design. In particular, the authors use this algorithm to generate a tile of elements in the output feature map with a reduced resource usage than other implementations, thus obtaining better results while employing the same number of resources. To balance this computational performance increment, they design line-buffers to cache the feature maps before and after the Winograd transformation. Then, they propose a design space exploration and use it to derive an automatic tool flow. Their implementation of AlexNet and VGG-16, implemented on a Xilinx ZC706 platform, reaches 1006.4 GOPS and 2940.7 GOPS respectively when using 200 MHz as a working frequency.

To improve the energy-efficiency of FPGA-based CNN accelerators, Zhang et al. [5] propose a pipelined FPGA cluster, where seven FPGAs are connected through a ring topology. Each link is composed by a serial link whose bandwidth is 750 MB/s. Then, the authors propose an analytical model to easily map CNNs on a multi-FPGA architecture. Such a model is able to determine the best implementation depending on the selected objective – i.e. energy, latency or throughput –. The results of different AlexNet and VGG-16 implementations, where different implementations have different objectives, show a maximum throughput of 825.6 GOPS and 1280.3 GOPS relying on a cluster of four and six Xilinx Virtex VX690T respectively.

CNNs represent a computationally intensive class of algorithms, whose structure and topology are constantly evolving and used in a wide range of different context, from embedded systems to data centers. As a result, an efficient DSE is crucial to guarantee both performance and reusability of the designed architecture on different platforms. For this reason, manual [4], [9] and platform-specific [5], [10]–[12] DSEs, easily limit the heterogeneity CNNs acceleration require. In this paper, we propose an architectural independent approach to compute CNNs inference on FPGA. Specifically, we propose a scalable and productive design methodology, together with an automatic pareto optimal DSE, able to quickly estimate the network performance for different target FPGAs.

III. Proposed CNN architecture

This Section details the methodology we employ for the acceleration of the CNNs inference on FPGAs. The proposed solution combines the benefit of HDL and HLS design flows to easily setup whichever CNN on the target accelerator. Moreover, data quantization and time-sharing are exploited to minimize resource utilization and allow the implementation of CNN inference on a wide number of FPGAs families.

In the context of CNNs accelerations, convolutional layers represent the most critical computational kernel, as they contain most of the network operations, and as a consequence, they consume most of the accelerator resources. An inaccurate design of the convolutional layer easily leads to degradation of the overall system performance. Moreover, the convolutional layer latency often dictates the performance of the entire accelerator, as it usually represents the computational bottleneck. For these reasons, it is crucial to fully exploit the parallel architecture offered by FPGA devices to minimize the convolution latency. Generally speaking, *convolutional layers* are always followed by an *activation layer*, and, sometimes, by a *pooling layer*. When the network computes quantized data, a *requantization layer* is included between the activation and the pooling layer. In this work, the aforementioned quartet – or trio, if the pooling layer is absent – is named *coarse-layer* and, as detailed in Section III-E, the features extraction stage of a CNN can be viewed as a chain of pipelined coarse-layers. The proposed methodology can leverage the same hardware resources to implement several subsequent *coarse-layers*, namely *stage*. Going deeper in the hierarchy, a *coarse-*

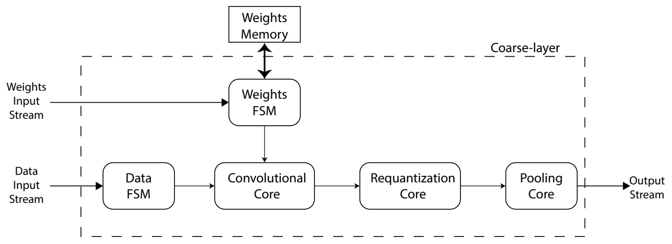

Fig. 1. Architecture of the accelerator coarse-layer.

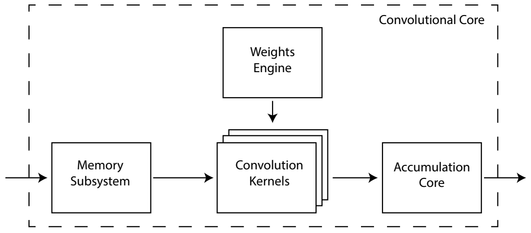

Fig. 2. Architecture of the Convolutional Core.

layer is consists in the components shown in Figure 1:

- The *Convolutional Core*, whose function is to compute the convolutions on the input feature-maps and to perform the accumulations to calculate the output feature-maps values. This core is in essence devoted to implementing the *convolutional layer*, excluding bias addition.
- The *Pooling Core*, which performs sub-sampling on the features maps to implement the *pooling layer*.
- The *Data Movement Finite State Machine (FSM)*, a set of cores that orchestrate data movement as well as storage of the intermediate results.
- The *Requantization Core*, whose function is to requantize the accumulated output from the Convolutional Core to bring it in the range of the output feature-maps, performing also bias addition and non-linear activation, essentially implementing the bias addition of the *convolutional layer* as well as the *activation layer*.

Each of the aforementioned components will be further analyzed in the remainder of this Section.

A. Convolutional Core

The convolutional core contains most of the CNN operations, and results to be computationally intensive even when high parallelism impose a substantial data movement from the off-chip memory. The outcome of this core is a volume, composed of a set of bi-dimensional matrices, named output feature-maps. Specifically, the core input feature-maps are convolved with a set of $N \times N$ filters and the resultant volume is summed element-wise to obtain a single bi-dimensional output feature-map. The described computational pattern keeps room for two principal parallelism opportunities, named *Intra-FM* and *Intra-layer* parallelisms. *Intra-FM parallelism* relies on the data dependencies absence to compute the output feature-map; as a result, a set of convolutions are computed concurrently and a reduction tree is used to create the output feature-map element. *Intra-layer parallelism* exploits data reuse to convolve the same layer input feature-maps with different weights-set to compute several output feature-maps in parallel. The above-mentioned parallelism paradigms can easily saturate the FPGA computational resources; even when low-precision fixed point arithmetic is adopted, Digital Signal Processings (DSPs) are not able to fulfill the available parallelism: consequently, a fine-grained tuning of the convolutional layer parallelism dictates both the overall performance and the parallelism to infer on the other components. To exploit both Intra-FM and Intra-layer parallelisms while saving DSPs resources, this methodology implements a *time-sharing* approach to compute each convolution with one DSP primitive. Each DSP is used as Multiply and Accumulate (MACC) unit to perform an element-wise multiplication between the input feature-map and the correspondent filter while accumulating these partial results. To allow a single DSP to perform this computation, the input datapath flows at a sub-multiple of the clock frequency, whose value changes according to the filter dimension. This solution offers a substantial DSPs count reduction and can exploit larger Intra-layer and Intra-FMs parallelism degree, resulting in less off-chip data movement to process the network. The proposed hardware design relies on parameterized HDL-based cores that allow specifying the following parameters: filter and input feature-map dimensions, bit-width of data and weights buses, padding and stride factors, Intra-FM and Intra-layer parallelisms.

The Convolutional Core architecture is represented in Figure 2. A tailored *Memory Subsystem* feeds a set of *Convolution Kernels*, guaranteeing a data-flow communication pattern, while a *Weights Engine* loads and distributes the weights windows among each *Convolution Kernel*, whose output is hierarchically accumulated by the *Accumulation Core* to create the corresponding output feature-maps. A detailed description of each component follows.

a) Memory Subsystem: The CNN convolutional layers require concurrent access to several neighbor elements of the input feature-maps: to perform the convolution, these elements need to be slid and processed over a filter. The presented methodology relies on bi-dimensional line buffers to give concurrent access on all the filter elements while performing a data flow computation. Specifically, an input channel streams sequentially the elements of the input feature-maps, while a chain of FIFOs is used to cache a suitable number of rows and output concurrent elements, preserving data locality. The concept of on-chip buffering implemented in the Memory Subsystem is built above the micro-architecture proposed in [17]. The implemented Memory Subsystem exploits Intra-FM parallelism, as the bit-width of its data-bus can be set to buffer several input feature-maps concurrently. Intra-layer parallelism is instead extracted from the Memory Subsystem outputs, as the same data are used to compute different output feature-maps. All datapaths of these blocks flow at a sub-multiple of the clock frequency to allow all the Convolution Kernels to compute data through the *time-sharing technique*. When the stride factor is greater than one, the updating frequency dynamically changes depending on the spatial location of the elements within the FIFO chains: this optimization allows to freeze the FIFO chains only when its output has to be processed by the Convolution Kernels to produce valid elements, while running at a full frequency to skip dummy elements windows. To orchestrate the data stream, a controller is in charge of handling FIFOs read and write enables.

b) Convolution Kernels: The Memory Subsystem outputs are consumed by a set of Convolution Kernels that perform the element-wise multiplications between the input feature-maps elements and weights, as well as their accumulations. This computation is performed by MACC units and Multiplexers (MUXs): both these functional blocks work at a full frequency and leverage the datapath slowness to consume data while sharing computational resources. In particular, each clock cycle of the time-sharing period is used to multiply the *i-th* element of the input window; the MUX output is fed to the MACC unit, which performs the multiplication with the corresponding weight and accumulates the partial result. After $N \times N$ clock cycles, where $N \times N$ is the filter size, the accumulator outputs the final result and resets its content. When using Intra-layer parallelism, the same data are streamed to different MACCs and multiplied with different weights.

c) Weights Engine: In CNN convolutional layers, the same weights window is slid over the whole input feature map. This computational pattern allows improving data locality by saving the weights window in small memories until the input feature map is completely processed. This solution requires a high peak bandwidth: when a new set of feature-maps have to be computed, slowness in loading memories-weights implies computation stalls, which impact on throughput. Moreover, depending on Intra-FM and Intra-layer parallelisms, the number of weights loaded in each iteration widely varies. To avoid stalls, weights are loaded and saved on-chip before the Memory Subsystem starts to forwards valid data to the Convolution Kernels. To properly orchestrate the weights loading, a tailored *gearbox* engine has been implemented. The *gearbox* automatically distributes data from off-chip to on-chip memories, providing weights to each MACC unit until the input feature-maps are completely processed. If the off-chip memory bandwidth is not enough to avoid computational stalls, the Memory Subsystem freezes, and stops producing data to the Convolution Kernels until the Weights Engine completes weights loading. From that moment, the Weights Engine starts writing data to the MACC units, guaranteeing synchronism and coherency with the input datapath.

d) Accumulation Core: The results of several Convolution Kernels are summed element-wise to create the output feature-map elements. To this purpose, the proposed methodology exploits a hierarchical accumulator strategy, where each hierarchy level is in charge of performing a partial data reduction until the output feature-map element is generated. Accumulators are generated through top levels parameters to match the Intra-FM and Intra-layer parallelisms.

B. Pooling Core

The proposed Pooling Core architecture can be seen as a simplified version of the Convolutional Core one, as both cores share the same memory access pattern. A FIFO chain collects the output feature-maps produced by one of the Requantization Core described in Section III-C, while its output is consumed by tailored *Pooling Kernels*, which compute the maximum value among the window. Unlike the Convolution Kernel, the Pooling Kernel does not require specific computational resources, as it just needs one comparator working at a full frequency to find the maximum value of the window.

C. Data Movement Cores

The data movement cores are in charge of handling data movement inside each stage. They are in essence FSMs, where each state is determined by the current coarse-layer they are computing. These cores are implemented using Vivado HLS, and are thus described using a high-level language – i.e. C++ –. For each state, nested for-loops handle data transfers to and from the different cores, counting the total number of elements and number of iterations to perform before the computation for the coarse-layer is completed. In particular, two main cores can be identified, namely *Data FSM* and *Weights FSM*. The *Data FSM* is interposed between two subsequent convolutional layers to perform data collection and reordering, as well as to preprocess data for the subsequent Convolution Kernel, as data between adjacent layers need to be normalized. Specifically, data coming from the previous coarse-layer are saved on-chip and distributed to the Convolution Kernel. Data reordering is necessary when the Convolutional Core computes a fraction of output feature-maps at a time; therefore, data at the input of the *Data FSM* are composed by a fragmented output feature-maps and must be properly reordered to feed the following coarse-layer correctly. The *Weights FSM* takes weights from the off-chip memory and simply perform an element-wise normalization of the Convolutional Core weights.

D. Requantization Core

As discussed in the literature [18], CNN inference with lower-precision quantized networks can achieve results comparable to floating point arithmetic, with an accuracy loss of only a few percentage points. Quantized networks are beneficial for FPGA devices, as fixed-point arithmetic is lighter than the floating point one, in terms of both latency and resource consumption. Moreover, data quantization reduces the memory footprint and memory bandwidth requirements, because it exploits data types with reduced bit-width. As a direct consequence of these considerations, this work leverages 8-bit integer arithmetics to perform the CNN computations, which is also the choice of several industry leaders [19], [20]. Nevertheless, in order to effectively work with 8-bit quantized data, a Requantization Core is necessary in order to correctly requantize the output feature-maps to avoid overflows. The requantization core is placed after the accumulation hierarchy and performs data requantization, Rectified Linear Units (ReLU) activation and biases addition. Indeed, from the Convolution Kernel input and the Accumulator outputs, data bit-widths have to be increased to avoid overflow during the computation. Therefore, once the convolution is performed and the output feature-maps are created, the Requantization Core rescales these elements at their original bit-widths. Lastly, it is important to highlight that this core might consume a part

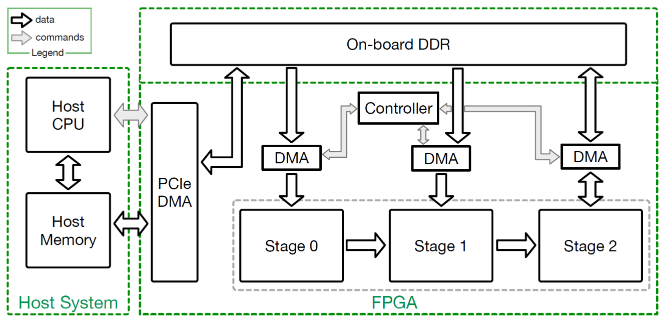

Fig. 3. High-level overview of the proposed hardware accelerator.

of the available DSPs, depending on what the HLS tool infers during HLS synthesis. However, DSP utilization can be tuned using HLS directives and it represents a small fraction of the Convolutional Core DSP utilization.

E. CNN Hardware Accelerator

The proposed architecture follows a classical host-accelerator scheme where the *host* – i.e. the CPU – is responsible for the initial data pre-processing – if needed – and the final results gathering after hardware acceleration takes place. Data is first sent to the accelerator through a Peripheral Component Interconnect Express (PCIe) interface, in order to copy both weights and input data to the FPGA external memory. The computation starts when all the weights are transferred and the first image is streamed to the accelerator. It is worth to note that weights are sent to the accelerator only at the beginning of the first computation; from that moment on, they reside on the FPGA external memory and a controller is in charge of sending them to the Weights Engine periodically. This solution allows to easily process multiple input frames in a batch, which is a common scenario in real applications, where models are used to process a collection – sometimes a continuous stream – of frames. As a result, the proposed acceleration methodology implements *Inter-layer parallelism*: such parallelism is realized when the accelerator executes layers in a pipelined fashion, which intuitively means that while layer l is computing on a frame, layer $l-1$ is computing on the subsequent frame in the batch. This is the general idea beyond *Inter-layer parallelism*, though the actual implementation differs. Indeed, our architecture allows to cluster subsequent layers into *stages* according to resource and performance requirements. Each pipeline stage s implements one or more subsequent coarse-layers and have a certain total latency τ_s. The total latency to process a single frame will, therefore, be $\sum_{s\in S}\tau_s$ as the frame needs to pass through all the stages to be fully processed. However, as for a classical pipeline, at steady state each stage s will be processing a different input frame, therefore resulting in an increase of the throughput since the time-per-frame will be given by the slowest stage, i.e. $\max_{s\in S}\tau_s$. Figure 3 depicts the overall system, showing the pipelined architecture of the accelerator along with the aforementioned components.

IV. Resources and performance model

Convolutional Cores are the key components of the design, as they are the most resource-demanding and the actual performance bound. Therefore, determining how many of them to use – i.e. the number of *stages* – and their degrees of internal parallelism, takes precedence over the other parts of the accelerator. Pooling Cores, data movements FSMs and Requantization Cores will simply be derived consequently. Moreover, their impact on the resulting execution time will be negligible. Indeed, the larger part of the total operations count is given by the Convolutional Core which is the larger contributor to the total stage latency. Moreover, DSPs usage is often the most constraining resource of the Convolutional Cores. For this reason, the proposed model relies on its estimation for a good assessment of the feasibility of the entire design. It is worth reminding that thanks to the proposed time-sharing, just one DSP is needed to perform an entire convolution, independently from the filter dimension. For a given Convolutional Core c, the total DSP usage is a function of the chosen Intra-FM parallelism δ and Intra-layer parallelism κ:

$$DSP_c(\delta_c,\kappa_c)=\delta_c\cdot\kappa_c \tag{1}$$

To estimate the total number of clock cycles τ_c of a given Convolutional Core c, the following equation can be used:

$$\tau_c(\delta_c,\kappa_c)=\frac{(\tau_{compute}+\tau_{load})\cdot|F_{l-1}|\cdot|F_l|}{\delta\cdot\kappa} \tag{2}$$

where $|F_{l-1}|$ and $|F_l|$ represent the cardinalities of respectively the input and output feature-maps, $\tau_{compute}$ is the number of cycles for computing the rows and columns of a single input/output feature-map pair, while τ_{load} are the cycles spent in loading data for each input/output feature-map pair. Overall, $\tau_{compute}$ can be evaluated as:

$$\tau_{compute}=m_l\cdot n_l\cdot\rho_l^2 \tag{3}$$

where (m_l,n_l) represent the size – rows m and columns n – of the output features maps and ρ is the kernel size. On the other hand, τ_{load} can be expressed as:

$$\tau_{load}=m_{l-1}\cdot n_{l-1}-m_l\cdot n_l \tag{4}$$

where (m_{l-1},n_{l-1}) represent the size – rows m and columns n – of the padded input feature-maps. The total execution time T_c will be then simply given by dividing τ_c by the target frequency H_c: $T_c=\tau_c/H_c$. Notice that the term ρ_l^2 stems from the fact that the proposed implementation uses time-sharing.

V. Design space exploration

In this Section, we describe our pareto optimal design space exploration that allows exploring CNN architectures that are both optimal in terms of throughput and DSPs resource consumption. The proposed design space exploration focuses on the Convolutional Cores, as they represent the key components in determining resource and performance of the final design.

A. Stages characterization

First of all, given the set of Convolutional Cores C within the CNN, such cores can be grouped in a set of stages S. A stage $s \in S$ consists in a set of Convolutional Cores $C_s \subseteq C$. Every stage $s \in S$ has to satisfy a number of requirements needed for achieving an efficient implementation of the stage:

1) To simplify routing of the final hardware design, we require the Convolutional Cores of the stage to be connected one after the other, i.e. to form a chain of cores.
2) Since the same hardware is shared by the Convolutional Cores within the stage, all the Convolutional Cores $c \in C_s$ must have the same kernel size ρ_s, the same Intra-FM parallelism δ_s and the same Intra-layer parallelism κ_s.
3) For best efficiency, we require that, for all $c \in C_s$, the Intra-FM parallelism δ_s is a submultiple of $|F^c_{l-1}|$ and that the Intra-layer parallelism κ_s is a submultiple of $|F^c_l|$.

Overall, a stage $s \in S$ can be uniquely identified by the tuple $s = (C_s, \delta_s, \kappa_s)$. Therefore, by taking into account the previously defined constraints, for a CNN containing $|C|$ convolutional layers, there exists at most $\mathcal{O}(|C|^2 \cdot |F^{max}_{l-1}| \cdot |F^{max}_l|)$ different stages, where $F^{max}_{l-1} = \max_{c \in C}\{|F^c_{l-1}|\}$ and $F^{max}_l = \max_{c \in C}\{|F^c_l|\}$. Hence, since the number of different stages within a CNN has a manageable size, we can enumerate all possible stages and compute the corresponding latency and DSP resource consumption of each stage. In particular, since the Convolutional Cores share the same hardware, the number of DSPs of a stage $s = (C_s, \delta_s, \kappa_s) \in S$ is:

$$DSP_s = \delta_s \cdot \kappa_s \tag{5}$$

On the other hand, the stage latency is the sum of the latencies of the Convolutional Cores within the stage:

$$\tau_s = \sum_{c \in C_s} \tau_c(\delta_s, \kappa_s) \tag{6}$$

B. System characterization

The final system contains a subset A of all the possible stages S. Indeed, each stage $s \in S$ implements a subset of the Convolutional Cores C_s and the final accelerator must contain a single implementation of each Convolutional Core. In order to define the feasible system solutions and optimize them in terms of resource consumption and performance, we define an ILP model. First, we characterize a system implementing the target CNN with the binary variables $x_s \forall s \in S$, where $x_s = 1$ if and only if stage s is used in the final system (i.e. if and only if $s \in A$). Then, for a feasible system, we require that each Convolutional Core is implemented by exactly one stage:

$$\sum_{s \in S | c \in C_s} x_s = 1 \quad \forall c \in C \tag{7}$$

Furthermore, in order to obtain an efficient hardware implementation, we also require that each pair of subsequent Convolutional Cores $c_i \in s_1$ and $c_{i+1} \in s_2$, are such that either κ_{s_1} is a multiple of δ_{s_2} or δ_{s_2} is a multiple of κ_{s_1}:

$$\begin{aligned} x_{s_1} + x_{s_2} \leq 1 \; \forall s_1, s_2 \in S \mid c_i \in s_1, c_{i+1} \in s_2 \wedge \\ (\kappa_{s_1}/\delta_{s_2} \vee \delta_{s_2}/\kappa_{s_1}) \end{aligned} \tag{8}$$

Algorithm 1 pareto optimal design space exploration

```
1: $m \leftarrow$ createILPModel($S$)
2: $\phi^{max} \leftarrow +\infty$                    $\triangleright$ the current initiation interval bound
3: $P \leftarrow \emptyset$
4: repeat
5:     $m.w_\phi \leftarrow 0,\ m.w_\Delta \leftarrow 1$                    $\triangleright$ minimize DSPs
6:     $m.\phi^{max} \leftarrow \phi^{max}$                    $\triangleright$ bound on initiation interval
7:     $m.\Delta^{max} \leftarrow +\infty$
8:     $m$.optimize()
9:     $feasible \leftarrow m$.isFeasible()
10:    if $feasible$ then
11:        $\Delta \leftarrow m$.getObjectiveValue()
12:        $m.w_\phi \leftarrow 1,\ m.w_\Delta \leftarrow 0$   $\triangleright$ minimize initiation interval
13:        $m.\Delta^{max} \leftarrow \Delta$                    $\triangleright$ bound on DSPs
14:        $m.\phi^{max} \leftarrow +\infty$
15:        $m$.optimize()
16:        $\phi^{max} \leftarrow m$.getObjectiveValue()
17:        $P \leftarrow P \cup (\phi^{max}, \Delta, m$.getSolution())
18:        $\phi^{max} \leftarrow \phi^{max} - 1$   $\triangleright$ reduce initiation interval bound
19: until $feasible$
20: return $P$
```

As previously discussed, the proposed architecture is a coarse grain pipeline composed of many stages. Hence, the initiation interval ϕ of the pipeline, which determines its throughput, is bounded by the slowest stage. Hence, we can introduce the variable ϕ in our ILP model and the corresponding constraint:

$$\phi \geq \tau_s \cdot x_s \; \forall s \in S \tag{9}$$

On the other hand, the total amount of DSPs Δ used by the system can be computed as:

$$\Delta = \sum_{s \in S} x_s \cdot DSP_s \tag{10}$$

C. pareto optimal exploration

In order to identify all the pareto optimal solutions in terms of DSP resource consumption Δ and initiation interval ϕ, we optimize multiple ILP models in which we alternate minimizations of resources and minimization of initiation interval. For this purpose, we define two new parameters: Δ^{max} and ϕ^{max} and add the following constraints to our ILP model:

$$\Delta \leq \Delta^{max}, \; \phi \leq \phi^{max} \tag{11}$$

and, as objective function, we minimize a linear combination of the initiation interval and DSP resource consumption:

$$argmin \; w_\phi \cdot \phi + w_\Delta \cdot \Delta \tag{12}$$

The exploration works as described in Algorithm 1. The procedure starts by finding the solution that requires the least amount of DSPs Δ, then, solves a second ILP model whose target is to minimize the initiation interval ϕ under the constraint of using at most Δ DSPs resources, found from the first solution. In this fashion, we guarantee that there exists no solution with less DSPs and the same initiation interval, or with the same DSPs and smaller initiation interval. Hence

TABLE I
PERFORMANCE OF THE PARETO OPTIMAL DSE

CNN	Conv. layers	Exec. time [s]	Design space size	Pareto optimal solutions
AlexNet	5	37.47	$2.78 \cdot 10^{10}$	91
VGG-16	13	38.23	$4.72 \cdot 10^{32}$	233

TABLE II
CONFIGURATION OF ALEXNET, EXTRACTED FROM THE DSE, IMPLEMENTED ON A XILINX VIRTEX-7 XC7VX485T FPGA.

Stage	Conv. layers	Intra-FM parallelism	Intra-layer parallelism	DSPs	Initiation Int. [clock cycles]
S0	1	3	96	288	430985
S1	2	32	32	1024	599664
S2	3, 4, 5	128	8	1024	756000

the solution found is pareto optimal and is saved in the set of pareto optimal solutions P. Then, the process repeats by searching for a solution having an initiation interval of at most $\phi - 1$. Hence, the next iteration of the algorithm can only find a different pareto optimal solution with smaller initiation interval and higher DSP resource requirements. The process iterates until no more feasible solutions can be found. This happens when all the available parallelism is exploited and the initiation interval cannot be reduced further.

Overall, the proposed algorithm allows identifying all the available pareto optimal solutions before implementing the design on any specific FPGA. Hence, the designer can leverage on this approach to either identify a suitable FPGA board for achieving the desired performance or, to seek for the best possible implementation that fits within a target FPGA.

VI. EXPERIMENTAL EVALUATION

In this Section, we validate both the automated pareto optimal DSE and the proposed CNN architecture. In particular, with respect to the DSE, we considered two well-known CNNs, namely AlexNet [13] and VGG-16 [14]. The DSE has been performed on a Intel Core i7-4870HQ CPU @ 2.50GHz processor, while Gurobi 8.1 [21] has been used for solving the ILP models. Table I reports the execution time of the DSE, the number of identified solutions, as well as the size of the design space being explored. It is worth noting that the DSE requires less than 40 seconds for both networks. For the VGG-16 case, the DSE identifies all the 233 pareto optimal solutions out of the $4.72 \cdot 10^{32}$ possible network implementations (the size of the design space has been computed with dynamic programming on all the combination of stages S). Finally, the identified pareto optimal solutions are shown in Figure 4 and Figure 5. As an example, from Figure 4, we can see that the best performing solution requires 3,872 DSPs with an estimated initiation interval of 430,985 cycles. In this configuration, the performance is limited by the first convolutional layer, which is the bottleneck and for which the proposed CNN architecture cannot leverage additional parallelism.

In order to further validate the approach, we implemented in hardware a configuration of AlexNet derived from the

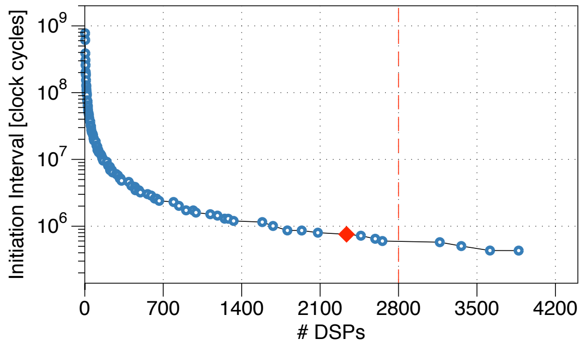

Fig. 4. Pareto optimal solutions of AlexNet identified by the DSE. The figure highlights the implemented AlexNet configuration - red diamond -, as well as the total number of DSP slices available on the target FPGA - dotted line -.

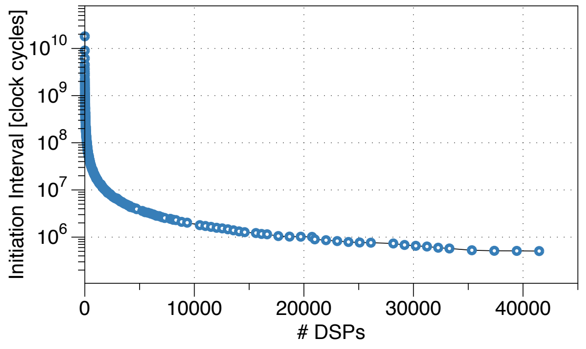

Fig. 5. Pareto optimal solutions of VGG-16 identified by the DSE.

DSE on a Xilinx Virtex-7 XC7VX485T FPGA, and we measured the estimation error of the DSE. We chose the AlexNet network as it exhibits more heterogeneity in terms of configuration parameters compared to VGG-16. Hence it allows to better validate both the model reliability and the architecture flexibility. As Figure 4 shows, most of the pareto optimal solution inferred by the DSE can be implemented on the target FPGA, which is equipped with $2,800$ DSPs slices. Among others, we choose the implementation described in Table II and highlighted in Figure 4, as it represents a good trade-off between parallelization degrees and overall resource utilization. As reported in Table II, we split the network into three *stages*, which represent three subsequent blocks of the computational pipeline described in Section III-E. Specifically, while the first and the second stages consist of a single *coarse-layer*, the last stage exploit the clustering capability discussed in Section III to implement the last three convolutional layers with the same hardware resources. Table II also shows the initiation interval of each *stage*, highlighting the third stage

TABLE III
RESOURCE CONSUMPTION OF THE IMPLEMENTED ALEXNET NETWORK TARGETING THE XILINX VIRTEX-7 XC7VX485T FPGA.

LUTs	Flip-Flops	DSPs	BRAMs
191,451 (63%)	390,556 (64%)	2,383 (85%)	1,511 (73%)

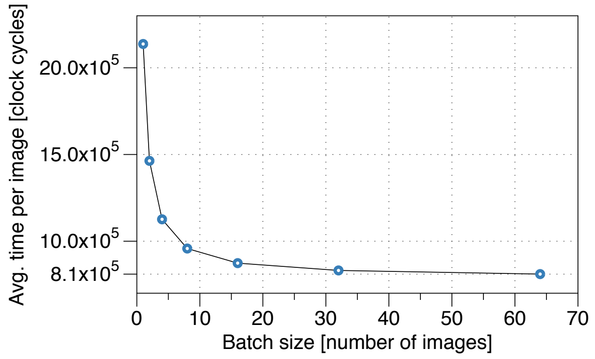

Fig. 6. Throughput of the AlexNet network with different batch sizes.

to be the network computational bottleneck for the selected configuration. As a result, when a batch of images is processed sequentially, and the computational pipeline is completely filled by the batch, the selected configuration should peak at a throughput of 756,000 clock cycles per image. Figure 6 reports the on-board run-time of the network for several batch sizes. At steady-state - i.e. for a batch of 64 images -, Figure 6 shows an experimental throughput of 810,233 clock cycles per image. Hence, the analytical model obtained an estimation error of 6.69% with respect to the experimental measure. Consequently, as the design target frequency is 200 MHz, the proposed implementation provides a throughput of 4.05 ms per image. The resource utilization of the selected AlexNet configuration is reported in Table III. As claimed in Section IV, DSPs slices are the most used FPGA resources. Compared to the theoretical estimation of Table II, 47 additional DSPs are required, which are mainly used to perform data re-quantization after the Convolutional Core.

VII. Conclusions

This paper proposes an architecture to evaluate the CNN feature-extraction stage on FPGA, along with reliable performance and resource models to drive an automatic pareto optimal DSE. The hardware architecture employs a dataflow-like computational pattern to maximize throughput, as well as a time-sharing technique to reduce the number of DSPs required to perform each convolution. To improve the design productivity, the proposed work relies on a mix of HDL and HLS written modules to easily compose the target CNN. The pareto optimal DSE allows to quickly identify candidate implementations for FPGA devices with different resource availability. The proposed DSE has been tested on both AlexNet and VGG-16 and was able to identify up to 233 pareto optimal solution is 38.23 seconds. Finally, starting from the DSE results, we implemented AlexNet on a Xilinx Virtex-7 XC7VX485T FPGA, achieving a throughput of 4.05 ms per image, with a performance estimation error of $6,69\%$. Future work will extend the analytical models to consider resources other than DSPs and estimate memory transfer bottlenecks. Finally, we are working to address multi-FPGA scenarios.

References

[1] D. Strigl, K. Kofler, and S. Podlipnig, “Performance and scalability of gpu-based convolutional neural networks,” in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, Feb 2010, pp. 317–324.

[2] “Microsoft project brainwave,” https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/, accessed: 2018-03-07.

[3] N. P. Jouppi, C. Young, N. Patil, D. Patterson, and G. A. E. al, “In-datacenter performance analysis of a tensor processing unit,” 2017. [Online]. Available: https://arxiv.org/pdf/1704.04760.pdf

[4] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, “Going deeper with embedded fpga platform for convolutional neural network,” in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’16. New York, NY, USA: ACM, 2016, pp. 26–35. [Online]. Available: http://doi.acm.org/10.1145/2847263.2847265

[5] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, “Energy-efficient cnn implementation on a deeply pipelined fpga cluster,” in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, ser. ISLPED ’16. New York, NY, USA: ACM, 2016, pp. 326–331. [Online]. Available: http://doi.acm.org/10.1145/2934583.2934644

[6] N. Voss, M. Bacis, O. Mencer, G. Gaydadjiev, and W. Luk, “Convolutional neural networks on dataflow engines,” in *2017 IEEE International Conference on Computer Design (ICCD)*, Nov 2017, pp. 435–438.

[7] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, “Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks,” in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2016, pp. 1–8.

[8] N. Raspa, G. Natale, M. Bacis, and M. D. Santambrogio, “A framework with cloud integration for cnn acceleration on fpga devices,” in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2018, pp. 170–177.

[9] J. Zhang and J. Li, “Improving the performance of opencl-based fpga accelerator for convolutional neural network,” in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’17. New York, NY, USA: ACM, 2017, pp. 25–34. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021698

[10] 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)*, June 2017, pp. 1–6.

[11] L. Lu, Y. Liang, Q. Xiao, and S. Yan, “Evaluating fast algorithms for convolutional neural networks on fpgas,” in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2017, pp. 101–108.

[12] 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*, ser. FPGA ’15. New York, NY, USA: ACM, 2015, pp. 161–170. [Online]. Available: http://doi.acm.org/10.1145/2684746.2689060

[13] “The alexnet architecture,” https://www.researchgate.net/figure/An-illustration-of-the-architecture-of-AlexNet, accessed: 2018-08-07.

[14] “The vgg-16 architecture,” https://www.science-data.com/blog/quantifying-entropy-in-images-a-prototype-with-neural-networks, accessed: 2018-08-07.

[15] S. Williams, A. Waterman, and D. Patterson, “Roofline: An insightful visual performance model for multicore architectures,” *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1498765.1498785

[16] A. Lavin, “Fast algorithms for convolutional neural networks,” *CoRR*, vol. abs/1509.09308, 2015. [Online]. Available: http://arxiv.org/abs/1509.09308

[17] M. Bacis, G. Natale, E. D. Sozzo, and M. D. Santambrogio, “A pipelined and scalable dataflow implementation of convolutional neural networks on fpga,” in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2017, pp. 90–97.

[18] S. Anwar, K. Hwang, and W. Sung, “Fixed point optimization of deep convolutional neural networks for object recognition,” in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, April 2015, pp. 1131–1135.

[19] R. Krishnamoorthi, “Quantizing deep convolutional networks for efficient inference: A whitepaper,” *CoRR*, vol. abs/1806.08342, 2018.

[20] “Nvidia. 2018. tensorrt,” https://developer.nvidia.com/tensorrt, accessed: 2018-09-07.

[21] L. Gurobi Optimization, “Gurobi optimizer reference manual,” 2018. [Online]. Available: http://www.gurobi.com