Performance Exploration on Pre-implemented CNN Hardware Accelerator on FPGA

Danielle Tchuinkou Kwadjo*, Joel Mandebi Mbongue*, Christophe Bobda*
*ECE Department, University of Florida, Gainesville Fl, USA
Email: dtchuinkoukwadjo@ufl.edu, jmandebimbongue@ufl.edu, cbobda@ece.ufl.edu

I. Introduction

As the complexity of FPGA architectures increases, there is a raising need to improved productivity and performance in several computing domains such as image processing, financial analytics, edge computing and deep learning. However, vendor tools are mostly general-purpose as they attempt to provide an acceptable quality of result (QoR) on a broad set of applications, which may not exploit application/domain-specific characteristics to deliver higher QoR. In this paper, we present a divide-and-conquer design flow that enables application/domain-specific optimization on the design of convolutional neural network (CNN) architectures on Xilinx FPGAs. The proposed approach follows three fundamental steps; Step 1: Break the design down into components, Step 2: Implement these separate components, and Step 3: Efficiently generate the final design by assembling pre-built components with minimal QoR lost. Recent research has even demonstrated that such approaches may provide better QoR than that of the traditional Vivado flow in some instances [1], [2]. By pre-implementing specific components of a design, higher performance can be achieved locally and maintained to a certain extent when assembling the final circuit. This approach is supported by two main observations [1]: (1) vendor tools such as Vivado tend to deliver high performance results on small modules in a design. (2) Computing applications such as machine learning designs increase in size by replicating modules. CNN inference refers to the forward propagation of M input images through L layers. The repetition of components within CNN architectures make them suitable candidates for RapidWright implementation as the CNN sub-modules can be optimized for performance in standalone, and the achieved performance can be preserved when replicating and relocating the modules across the FPGA.

II. Proposed Design Flow

The overview of the pre-implemented flow is presented in Figure 1. The flow has two major steps that are: *function optimization* and *architecture optimization*. The *function optimization* essentially consists in performing a design space exploration of the performances that can be achieved on sub-functions. It takes into consideration some design constraints such as device, timing, floor planning, and power. If the design space exploration results in satisfiable performance, the produced netlists are saved into a database in the form

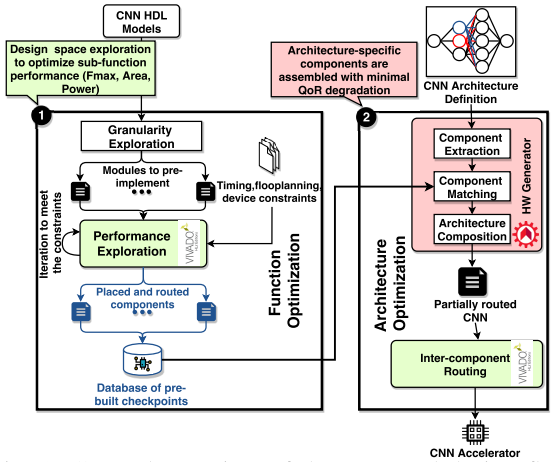

Fig. 1: General overview of the proposed design flow.

of DCPs. This step is semi-manual as the designer must choose and pre-compile the sub-functions in a design using vendor tools. It is however performed exactly once, and the saved netlists may serve in multiple designs. The *architecture optimization* is a fully automated process that aims to combine the pre-built components (the netlists saved in the *function optimization* phase) into a CNN architecture as defined by the users.

a) Function Optimization: This section describes the major steps involved in the design of optimized sub-functions.

- We start by manually building the CNN components Out of Context (OOC).
- **Strategic floorplanning:** utilizing pblock constraints allows carefully selecting the FPGA resources that will be used by each design component. It also enables the designer with the possibility to only use necessary resources.
- textbfStrategic port planning: the placement of the ports when pre-implementing modules is one of the most important steps to ensure high performance and productivity improvement.
- **Clock routing:** to accurately run the timing analysis on the OOC modules, source clock buffers must be specified using the constraint HD.CLK_SRC. Though the buffers are not inserted in the OOC modules, clock signals are partially routed to the interconnect tiles and the timing analysis tool can then run timing estimations.

- **Logic locking:** Once a module attains a desirable performance (F_{max}, area, power, etc), we lock the placement and routing to prevent Vivado from altering the design later and preserve design performance. The other advantage of locking the design is that the final inter-module routing with Vivado will only consider non-routed nets. This decreases compilation times and improves the productivity.
- **Checkpoint file generation:** pre-implemented modules are stored in the form of DCPs.

b) Architecture Optimization:

1) **Component Extraction**: The major function of the *Component Extraction* is to create a DFG from the CNN architecture definition and compose the resources needed for the CNNs hardware accelerator on FPGA. The nodes represent the components, and the edges account for the connections between them. Each node of the graph can be a component candidate. Nevertheless, consecutive nodes in the graph can be pre-implemented as one component if the data movement between them does not required a memory controller. In that case, simple handshake protocol is enough to provide node-to-node communication with simply single-source, single-sink FIFO queues with un-bounded length.
2) **Component Matching**: the API parses the DFG using a breath-first search (BFS) approach. The hardware generator that we implement with the RapidWright API loads the DCPs corresponding to the components defined in the CNN architecture definition to compose the final architecture.
3) **Architecture Composition** To achieve physical hardware re-usability, some requirements must be fulfilled: each component must implement a specific interface to communicate with the other design modules. The "source", is a dedicated memory controller that read data from a memory and feed their computing units. The second interface called "sink" controls the writing of feature maps in on-chip memory. After stitching, the blocks are placed, a DCP file is generated, then read into Vivado to complete the inter-component routing.
4) **Inter-component Routing** At this stage, the design contains all the CNN modules, with the logic and the internal routing locked. We therefore utilize Vivado for the final routing, which essentially consists in finding FPGA interconnects to implement the logic routes between components.

III. Experimental Results

Overall, Pre-implementing basic components have the potentiality of reducing resource utilization as shown in Table I. When the design is small, vivado can provide a better optimization of the resources. We present a comparison with FPGA designs that utilize a batch size of 1, and we report simultaneously latency and Frequency. In Table II, we present the performance of each component as well as the pre-implemented Lenet. Overall, Lenet achieves up to 1.16X higher frequency than the classic stream-like architecture. The first convolution reaches 562 MHz. However, with a higher number of parameters (from 156 in conv1 to 2416 in conv2), the number of multiplications increases from 117600 to 240000, and having a negative impact on the frequency. We observe the same tendency on FC1 and FC2. The frequency of the pre-built design is upper bounded by the slowest component in the design.

	CLB LUTs	CLB Registers	BRAMs	DSPs
Lenet	32021 (9.65%)	8538 (1.29%)	463 (21.44%)	144 (5.21%)
Pre-implemented Lenet	29491 (8.89%)↓	8442 (1.26%)↑	457 (21.16%)↓	144.00 (5.21%)⇔
VGG-16	282870 (85.28%)	215763 (32.53%)	854 (38.54%)	2116 (76.66)
Pre-implemented VGG-16	261321 (78.79%)↓	180754 (27.25%)↓	786 (36.39%)↓	2123 (76.92%)↑

TABLE I: FPGA Resource Utilization

Layers	Full Network	Conv1	Pool1+ ReLU1	Conv2	Pool2+ ReLU2	FC1	FC2	Our work
Freq (Mhz)	375	562	633	475	588	497	543	437 (1.16X)↑
Latency (ns)	249.7	37.33	12.93	63.46	22.51	49.32	25.05	249.10

TABLE II: LeNet Performance

	Frequency MHz	Latency (ms)
VGG	200 MHz	55.13
Component1	367 MHz	1.54
Component2	475 MHz	0.021
Component3	341 MHz	4.32
Component4	461 MHz	0.034
Component5	326 MHz	3.97
Component6	454 MHz	0.035
Component7	313 MHz	4.3
Component8	432 MHz	0.041
Component9	308 MHz	4.56
Component10	300 MHz	1.62
Component11	300 MHz	1.62
Component12	375 MHz	0.91
Our work	243 MHz (1.17 ×)↑	56.67 (1.02×)↑

Fig. 2: Performance Exploration of VGG

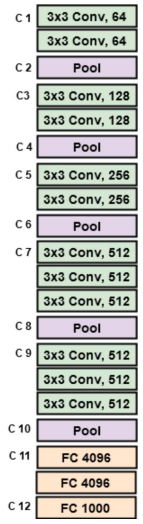

Fig. 3: VGG components

The pre-implementing VGG has 1.17 × higher frequency than the baseline VGG implementation, with a 0.53 ms higher latency (Figure 2). In contrary to LeNet, VGG has more and dense layers to place and route on the chip. When several design components must be spread around the chip, a rising issue is how to deal with fabric discontinuities such as erratic tile patterns and I/O columns. Those discontinuities increase the datapath and have a negative effect on the performance. Hence, inserting pipeline elements such as FFs on the critical path improves the timing performance, while increasing the overall latency.

References

[1] C. Lavin and A. Kaviani, "Rapidwright: Enabling custom crafted implementations for fpgas," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 133–140, IEEE, 2018.

[2] J. M. Mbongue, D. T. Kwadjo, and C. Bobda, "Automatic generation of application-specific fpga overlays with rapidwright," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, pp. 303–306, IEEE, 2019.