Exploring a Layer-based Pre-implemented Flow for Mapping CNN 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

***Abstract*—Convolutional Neural Networks are compute-intensive learning models that have demonstrated ability and effectiveness in solving complex learning problems. However, developing a high-performance FPGA accelerator for CNN often demands high programming skills, hardware verification, precise distribution localization, and long development cycles. Besides, CNN depth increases by reuse and replication of multiple layers. This paper proposes a programming flow for CNN on FPGA to generate high-performance accelerators by assembling CNN pre-implemented components as a puzzle based on the graph topology. Using pre-implemented components allows us to use the minimum of resources necessary, predict the performance, and gain in productivity since there is no need to synthesize any HDL code. Furthermore, components can be reused for a different range of applications. Through prototyping, we demonstrated the viability and relevance of our approach. Experiments show a productivity improvement of up to 69% compared to a traditional FPGA implementation while achieving over 1.75× higher Fmax with lower resources and power consumption.**

***Index Terms*—FPGA, CNN, DFG, RapidWright, Pre-implemented flow**

I. Introduction

The perpetual growth of integration capacity in FPGA technology has led to the advent of large devices capable of hosting millions of logic components and thousands of hard IP blocks. For instance, Xilinx recently released the Alveo U250 Data Center Accelerator Card powered by the UltraScale+ architecture for data center and artificial intelligence acceleration. The U250 gathers four super logic regions each containing approximately 340000 logic elements, 20MB of BRAM, 90MB of UltraRAM, and 3000 DSP slices [1]. The Intel Arria 10 in Microsoft Cloud features about 1.1 million logic elements, 3036 DSP logics, and 67MB of BRAM [2] [3]. The innovation in FPGA hardware architecture provides the basis for unprecedented flexibility and acceleration in high-performance computing and embedded system applications. It also requires CAD tools capable of extracting domain/application-specific features to better leverage the resources available in recent FPGAs. As FPGA architectures' complexity increases, there is a rising need for 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. 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.

This paper presents 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 [4]–[6]. 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. Two main observations support this approach [4]: (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. We leverage Vivado to produce highly optimized implementations for the principal modules of a design.

As motivation example, Figure 1 summarizes a few results from the work of Mandebi et al. [5]. It represents an architecture in which a block of 3×3 processing elements implementing four different applications have both been pre-implemented and built with Vivado and RapidWright. Compilation time and maximum frequency achieved by each design flow are then recorded. It shows that the pre-implemented design flow could achieve up to 37% gain in productivity and 33% higher F_{max} compared to compiling the same designs with Vivado. While little details are provided on the choice of the granularity of the pre-built components, the work proved that their pre-implementing modules could significantly improve the QoR when exploiting application/domain-specific features.

In the context of this work, we aim to explore the performance that can be achieved when utilizing RapidWright in the design flow of an FPGA accelerator for a CNN. Specifically, our contribution includes:

- **Reviewing the pre-implemented flow:** we will discuss key steps to follow to leverage RapidWright efficiently.
- **Describing CNNs architecture:** we will explore the features of state-of-art CNNs that are suitable for improvement.

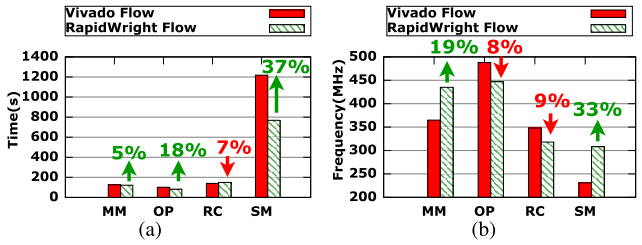


Fig. 1: Motivation example. (a) Compilation time comparison. (b) F_{max} comparison. The results from previous research show that the pre-implemented design flow with RapidWright can lead to improved productivity and QoR compared to the traditional design flow with Vivado [5] (MM=Matrix Multiplication; OP=Outer Product; RC=Robert Cross; SM=Smoothing).

- **Proposing a design flow:** we will explore different strategies that could improve the final QoR compared to the traditional design flow with Vivado.

As opposed to vendor tools that are closed source, we believe the full access to RapidWright internal features and design resources makes it suitable for design flow exploration and the implementation of targeted FPGA solutions.

II. Pre-implemented Flow with RapidWright

Out of Context Flow [7]: this design mode ensures that the placement of I/O buffers is disabled to facilitate the design of internal components of an architecture. It has several advantages: (1) it allows us to implement and analyze (resource analysis, timing analysis, power analysis, etc.) a module independently of the rest of the design. (2) it enables reusing and preserving the characteristics of placed and routed modules within a top-level design.

Pre-implementing Components: vendor CAD tools such as Vivado use heuristics for physical implementation (placement and routing). They consider the number of cells in a design, their connections, and the target FPGA device's physical architecture to generate a circuit according to specified constraints. Consequently, vendor tools generally achieve better QoR on smaller designs as the resource allocation problem addressed in the physical implementation is well-known to be NP-hard [8]. Focusing the optimization on smaller modules may therefore lead to overall QoR improvement in a design. Furthermore, several literature works have shown that pre-implementing components or macros can significantly decrease the overall FPGA compilation time with performance benefits [5], [9]. Therefore, the pre-implemented flow aims to generate high-performance implementations by reusing multiple contexts and chip locations, high-quality and customized pre-built circuits.

RapidWright [4]: is an open-source Java framework from Xilinx Research Labs that provides a bridge to Vivado back-end at different compilation stages (synthesis, optimization, placement, routing, etc.) using design checkpoint (DCP) files as illustrated in Figure 2. Once a DCP is loaded within RapidWright, the logical/physical netlist data structures and functions provided in the RapidWright APIs enable custom netlist manipulations such as cell and net instantiation, edition, and deletion. The hundreds of APIs in RapidWright make it possible to directly access/edit logic and routing resources and run some operations such as timing analysis, placement, and routing.

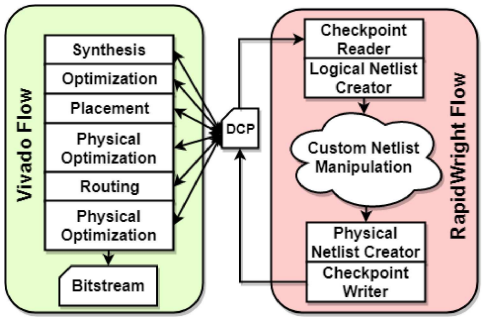


Fig. 2: Vivado and RapidWright interaction

III. Overview on CNN FPGA Architectures

CNN inference refers to the forward propagation of M input images through L layers. In recent years. Multiple CNN architectures on FPGA have been proposed. We classify these architectures into two main categories that are: *Single Instruction, Multiple Data (SIMD) accelerators* and *streaming-based accelerators*. This section highlights the potential benefits of designing FPGA-based CNN architectures with the pre-implemented flow of RapidWright and the challenges that may arise. We do not discuss any architecture implementation detail. The general computational flow in SIMD CNN accelerators [10]–[12] is to fetch feature maps and weights from external memory to on-chip buffers. These data are then streamed into computing engines composed of several processing elements (PE). At the end of the PE computations, the results are streamed back to on-chip buffers and, if necessary, to the external memory to be processed in subsequent CNN layers. Each PE is configurable and has its own computing resources, mainly using DSP blocks, and data caching features relying on on-chip registers. Computing engines are usually composed of hundreds of identical PEs that are replicated across the chip for accelerating specific layers of the CNN. This repetition of components within CNN architectures makes them suitable candidates for RapidWright implementation. 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. Accelerators with the streaming architecture always tailor the hardware regarding the target network [13], [14]. The topology of such CNN accelerators is transformed into the specified layer-by-layer execution schedule, following the structure of the DAG [15]. Shen et al. [16] note that FPGA-based acceleration used CLE [1] to process consecutive CNN layers one at a time. The intermediate results between layers can be stored in registers, memory or directly pipelined into the next layer. However, since the dimension and filter parameters

[1] Convolutional Layer Engine

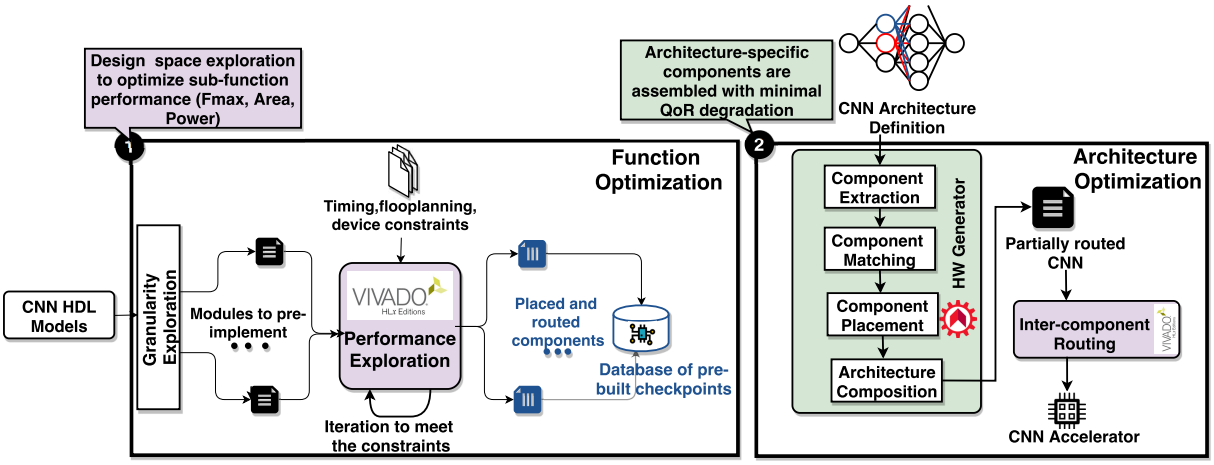


Fig. 3: General overview of the proposed design flow.

from consecutive layers might be different, using a fixed CLE for all layers leads to poor performance and poor resource utilization. For an L-layer CNN, they propose using Q CLEs, where $Q < L$, to maximize BRAM availability for each CLEs. With $Q < L$, some layers are replicated in the design, making this architecture suitable for the pre-implemented flow. In the same line of work, a streamed accelerator [13], [17] consists of sequential execution of all the layers of a given CNN. This type of architecture's main advantage is to minimize the latency caused by communication with off-chip memory and, thereby, maximize on-chip memory communication, ensuring high throughput and avoiding any latency.

IV. Proposed Design Flow

In this section, we present the design exploration steps implemented to optimize CNN components to fully exploit the benefit of our approach. The overview of the pre-implemented flow is presented in Figure 3. The flow has two major steps that are: *function optimization* and *architecture optimization*. The *function optimization* essentially consists of 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 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 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.

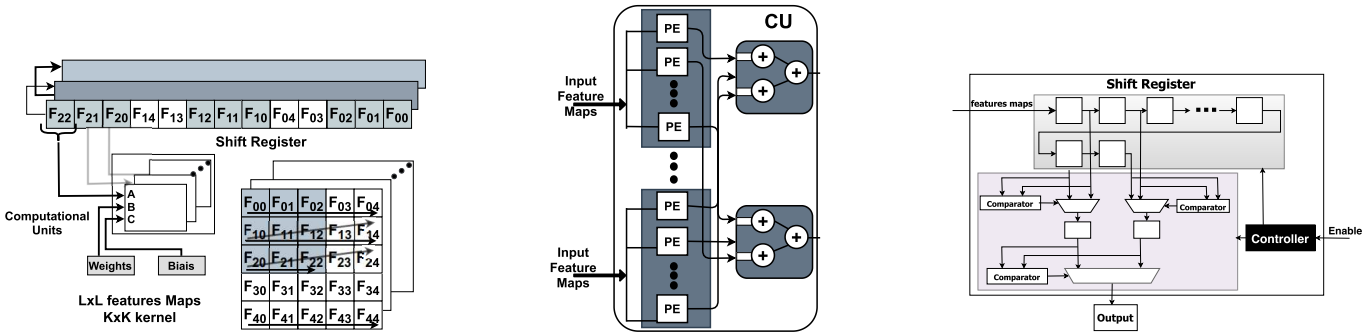
1) Granularity Exploration: The design space exploration only supports CNNs. A typical CNN is usually composed of:

- **Convolution:** The convolution layer convolves the input image with a set of learnable filters, each producing one feature map in the output image.
- **Pooling:** Max-pooling splits the input image into a set of non-overlapping rectangles and, for each of these sub-regions, outputs the maximum value.
- **Rectified-Linear:** Given an input value x, the ReLU is a simple calculation that returns the value provided as input directly x if $x > 0$ and 0 otherwise. Several ReLU functions exist and might be employed.
- **Fully Connected (FC):** Each activation of a FC layer is the result of a scalar product composed of input values, weights, and a bias.

By porting these four layers onto the FPGA, the vast majority of forward processing networks can be implemented. The modules' implementations should revolve around this minimum of granularity. Automated decomposition of user logic into leaf components is complementary future work.

2) Performance Exploration: We start by manually building the CNN components Out of Context (OOC). Figure 4 presents the main circuits of the proposed flow. The OOC flow ensures that I/O buffers and global clocks resources are not inserted into the netlists as the pre-built components are still inserted within the top-level module of the design. While efficiently designing components OOC requires hardware expertise, it is done exactly once, and the pre-built netlists may be reused in several other applications. To achieve high QoR in the performance exploration phase, a few design considerations are necessary:

- **Strategic floorplanning:** utilizing pblock constraints allows carefully selecting the FPGA resources that each design component will use. It helps to improve the module-level performance and area. Hence, the designer can only use necessary resources instead of letting the CAD tool utilize as many chip tiles as it wants. Given that Xilinx architectures generally replicate the resource



(a) 2D Convolution Circuit

(b) Architecture of a Compute Unit

(c) Architecture of a max-pooling Layer

Fig. 4: Pre-implemented Design Components

structures (CLBs, DSPs, BRAM, URAM, etc.) over an entire column of clock regions, the smaller the area of a pblock is, the more RapidWright will be capable of relocating the design components across the chip, which increases the reusability. The automated definition of pblock range is out of the scope of this work.

- **Strategic port planning:** the placement of the ports when pre-implementing modules are one of the most important steps to ensure high performance and productivity improvement. Failure to plan the location of the ports of the pre-implemented modules may result in long compilation time, poor performance, and high congestion in the design in which they are inserted.
  To preserve the QoR of the sub-modules in the final design, we should foresee the length of the nets connecting the cells at the interface of the sub-modules. However, the modules are pre-implemented independently. Hence, the CAD tool is not aware of the context in which the modules will be inserted into a design and connected to other components. A pre-implemented component may then achieve a high maximum frequency in standalone but perform poorly when inserted into a design because of very long inter-module nets. We therefore pre-implement the modules with partition pin constraints (PartPins) [7] to specify the interconnect tiles that will route the nets connecting to the other modules of a design.
- **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:** the primary goal of the performance exploration is to achieve high QoR locally. 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 productivity.
- **Checkpoint file generation:** pre-implemented modules are stored in the form of DCPs. The top-level design will then implement synthesis black-boxes that the optimized pre-built modules will fill.

The implementation here is done using vendor tools and considers several constraints such as timing and floor planning. The p-block partitioning is performed for each component according to its needs in terms of hardware resources and the physical structure of the FPGA. However, when synthesizing components OOC, there is no control over how the I/O ports are placed. With p-blocks and timing constraints, I/O ports might be contained anywhere in the p-block resulting in routing congestion and timing issues around I/O interfaces when generating the whole design as described in Sec. IV-B.

B. Architecture Optimization

In this section, we discuss the generation of a CNN accelerator based on user definition. The architecture optimization follows four major stages: component extraction, component matching, architecture composition, and inter-component routing. The following paragraphs will elaborate on each of these phases.

1) Component Extraction: From the library of pre-built components, users compose the CNNs hardware accelerator's resources on FPGA. This implies providing information about the topology and the type of layers that compose the CNN in a form that we call: "*CNN architecture definition*". In the following stage, a CNN hardware generator that we design with the RapidWright C API automatically produces the corresponding CNN accelerator. The major function of the *Component Extraction* is to parse the CNN architecture definition from the DFG specification and identify the components. It then creates a data flow graph (DFG) structure in which 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 require a memory controller. In that case, a simple handshake protocol is enough to provide node-to-node communication with simply single-source, single-sink FIFO queues with unbounded length. For

instance, the first convolution of LeNet outputs $6@28 \times 28$ features maps, and pooling outputs $6@14 \times 14$ feature maps from a 2×2 sliding windows. This architecture requires a memory controller to compose the addresses to read/store the data from/to the memory and feed the FIFOs, as shown in Figure 5. That constraint is no required for the following ReLu, and the operation can be directly applied to intermediate results of the pooling layers.

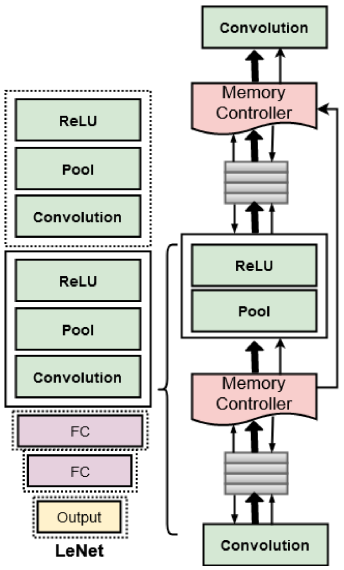


Fig. 5: Communication Interface between Components

2) Component Matching: The RapidWright application first parses the DFG using a breath-first search (BFS) approach (Algorithm 1 line 1-10). This enables efficiently discovering the components to load into the CNN architecture as well as their connectivity. We choose the BFS traversal as the DFGs representing CNN architectures are generally deeper than wider. Each node is described with a set of characteristics. For instance, a convolution is identified with information such as kernel size, the padding, and the strike for a convolution (see Figure **??**). The hardware generator that we implement with the RapidWright API loads the DCPs corresponding to the components defined in the CNN architecture definition from the pre-built checkpoint's database to compose the final architecture.

3) Architecture Composition: To achieve physical hardware reusability, some requirements must be fulfilled: each component must implement a specific interface to communicate with the other design modules. Components are pre-implemented with two interfaces. The first interface, called "source", is a dedicated memory controller that reads data from memory and feeds their computing units. The second interface, called "sink" controls the writing of feature maps in on-chip memory. Finally, since all the components implement a well-known interface, we use the Rapidwright API to create interconnections. It is done by inserting specific nets in the design's netlist to implement logic routing between the different components that communicate in the design (Algorithm 1 line 11-18). After stitching, the blocks are placed, a DCP file is generated, then read into Vivado to complete the inter-component routing.

Algorithm 1: DCP generation Algorithm with Rapidwright

Input : Design d, Graph G, Node root
Output: DCP file

```
1  let Q be a queue ;
2  mark root as discovered ;
3  Q.enqueue(root) ;
4  while Q.size() != 0 do
5      Node v = Q.dequeue() if v is the goal then
6          return v ;
7      end
8      Nodes w = G.adjacentEdges(v) ;
9      foreach edges from v to w do
10         if w is not marked then
11             pblock p = define pblock range for w;
12             addNodeToDesign(p);
13             Ports ports_v = selectPortOfInterest(v);
14             Ports ports_w = selectPortOfInterest(w);
15             foreach (ports_v, ports_w) do
16                 create nets to connect the two ports;
17             end
18             mark w as discovered ;
19             w.parent = v ;
20             Q.enqueue(w) ;
21         end
22     end
23 end
24 +
```

4) Component Placement: The placement algorithm is based on Xilinx Ultrascale architecture, which is an array of programmable logic blocks consisting of configurable logic blocks (CLB), Embedded Memory (BRAM), and multiplier (DSP) blocks. The array is surrounded by I/O Blocks allowing off-chip connections. DSP blocks and BRAMs are arranged columnar-wise and spread across the device. In this work, we aim to find a congestion-aware timing-driven placement for components of the input graph. As each component is already placed and routed, it must be replicated on the device to compose the overall architecture. Since components are pre-implemented within pblocks, the type and amount of resources used are reported. The algorithm works as follows: we recursively parse the input graph and place the first component. For each adjacent component, we assign a location on the FPGA grid. We define timing and congestion cost functions to evaluate the cost of the assigned location.

a) The timing cost: is defined by the wire length between two components.

$$timing\_cost = \sum_{i=1,i<j}^{n-1} HPWL(W_{i,j}) \tag{1}$$

Where $W_{i,j}$ is the wire length between component i and j.

b) Congestion Estimation:: for optimal routing, a placement algorithm must consider the number of resources used by each inter-component net and the interaction between them.

For instance, if all nets are limited to a relatively small portion of the chip area, the routing path request will probably be very high. A weighted sum of the number of components overlapping is used to measure congestion to take this into account.

$$cgt_{coefficient} = \#components\ overlaps\ within\ tile_i \quad (2)$$

$$cgt\_cost = \frac{\sum_{i=1,i<j}^{n-1} \frac{P_{i,j}}{H \times W}}{\sum_{i=1,i<j}^{n-1} \frac{P_{i,j}}{H \times W}} \quad (3)$$

Where $W_{i,j}$ is the wire length between component i and j.

The component placement is validated if the costs are lower than a defined threshold. Otherwise, we unplaced them for each previously placed component, find another location until the costs are satisfied.

5) Inter-component Routing: After the architecture composition, the design contains all the necessary CNN modules. Each of the design modules still has the logic and the internal routing locked. However, the RapidWright hardware generator that we implement only enables the logic routing between the components. While recent updates in the RapidWright API provide some functions to route the designs, the routing heuristics are still a work in progress and are not as mature as Vivado. Therefore, we utilize Vivado for the final routing, which essentially consists of finding FPGA interconnects to implement the logic routes created within RapidWright in a way that minimizes timing delays.

V. Experimental Results

A. Evaluation Platform and Setup

For evaluation purposes, designs are implemented on a Xilinx Kintex UltraScale+ FPGA (xcku5p-ffvd900-2-i). The hardware is generated using Vivado v2019.2 and RapidWright v2019.1. The hardware generation is conducted on a computer equipped with an Intel Corei7-9700K CPU@3.60GHz×4 processor and 32GB of RAM.

B. Benchmarks

We study two CNN architectures: LeNet [18] and VGG [19]. We run applications individually to assess achievable performances, in particular: (1) global latency, (2) Fmax and productivity, and (3) resource utilization when comparing pre-implemented components to fully implemented CNNs. For both networks, we use a valid Padding and strike of 1. Figure I presents the different characteristics of the two networks. For performance comparison, we use a stream-like architecture for both networks.

1) Lenet Architecture: It is built by replication of four main modules: (1) The convolution: this module performs the convolution computing using a systolic array architecture. The fully connecting layers are also implemented as convolution, with the kernel size equal to input data size. (2) The max pool layers, (3) The ReLU layers, (4) The memory_managment unit, jogging around the input data, and feed the computing units. The weights and biases are hardcoded in ROM. This choice has been decided out of simplicity.

	LeNet-5	VGG-16
# CONV Layer	**2**	**16**
# weights	26 K	14.7 M
# MACs	1.9 M	15.3 G
# FC Layers	**2**	**3**
# weights	406 K	124 M
# MACs	405 K	124 M
Total Weights	431 K	138 M
Total MACs	2,3 M	15.5 G

TABLE I: Computational hardware resources for state-of-art DNNs.

2) VGG-16 Architecture: VGG consists of 16 convolutional layers and is very appealing for the pre-implemented flow because of its uniform architecture. Input images are passed through a stack of convolutional layers with a fixed filter size of 3×3 and a stride of 1. There are five max-pooling filters built-in between convolutional layers. Three fully connected layers follow the stack of convolutional layers. The replicability of layers within VGG suits the pre-implemented flow. We use off-chip memory to store the coefficient data and data layout configuration files. The off-chip memory allocation is based on a Best-Fit with Coalescing algorithm. The goal of this allocator is to support defragmentation via coalescing. The principle behind this algorithm is to divide the memory into a series of memory blocks, each of which is managed by a block data structure. From the block structure, information such as the base address of the memory block, the state of use of the memory block, the block's size, the pointer on the previous block and the following can be obtained. All memory can be represented by a block structure with a double-link list.

C. Resource Utilization

Pre-implementing basic components have the potentiality of reducing resource utilization as shown in Table II. The classic implementation Lenet and VGG use respectively 9.65% and 78.79% of LUTs, 0.99%, and 32.53% of registers. The pre-implemented version of Lenet and VGG use respectively 8.89% and 0.99% of LUTs, with 78.79% and 27.25% of registers. Overall, the pre-implement networks use fewer resources than the baseline implementation. When the design is small, vivado can provide better optimization of the resources. Furthermore, when pre-implementing components, we define p-blocks, limiting the number of resources that vivado can use, forcing some area optimizations. When the design is bigger, vivado tends to maximize adaptation capacity and becomes difficult to capture all its specificities.

Lenet uses 21.44% of the BRAM available on the chip. This is simply because the weights and biases are hardcoded in ROM and use more resources. The pre-implemented Lenet (resp pre-implemented VGG) uses 0.28% less BRAM (resp. 2.56%). Vivado can optimize individual component IR without BRAM insertion while adding such resources when compiling a bigger design, which translates into higher power

	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 II: FPGA Resource Utilization

consumption. The amount of DPS is the same for LeNet implementation. However, a notice an increase of 0.26% the pre-implemented VGG. By defining p-block for each component, we sometimes provide more DSPs than needed to have enough resources to place the design. This is due to the topology of Xilinx FPGA, which are organized column-wise.

D. Productivity

When the size of the design increases, the productivity is affected. This section shows how the proposed flow can leverage component reuse to reduce both compile-time and implementation cycles. Figure 6 presents the time in seconds to generate the design checkpoint with both rapidwright and vivado. This time measure the implementation and the generation of DCP. For the Baseline Lenet and VGG, implementation time is the sum of Vivado's *opt_design*, *place_design*, *phys_opt_design* and *route_design* functions. Since components have already been implemented off-line for the pre-implemented networks, we only measure DCP generation with rapidwright and inter-component routing with vivado. The pre-implemented flow takes 16.54 min (resp. 52.87 min) to generate Lenet (resp. VGG). There is a productivity improvement of 69% for Lenet and 61% for VGG when using the pre-implemented flow. For Lenet (resp VGG), the stitching with RapiWright represents only 5% (resp. 9%) of the total time. RapidWright has minimal impact on productivity. The biggest portion of the time is used to route the nets between the p-blocks.

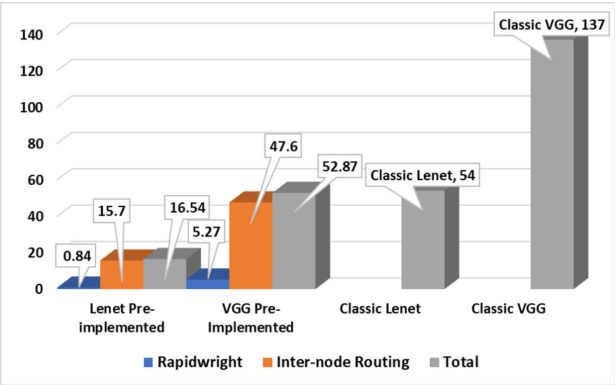


Fig. 6: Design Generation Time for implementation of LeNet and VGG with vivado and the pre-implemented flow in minutes

E. Performance

This section presents a comparison with FPGA designs that utilize a batch size of 1, and we report latency and frequency simultaneously. In Table III, we present each component's performance as well as the pre-implemented Lenet. Overall, Lenet achieves up to 1.75X higher frequency than the classic stream-like architecture, which is an improvement of over 75%. 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, negatively impacting 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.

The pre-implementing VGG has 1.31 × higher frequency than the baseline VGG implementation, with a 0.53 ms higher latency (Table 7). 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.

	Frequency MHz	Latency (ms)
VGG	200 MHz	55.13
Component 1	367 MHz	1.54
Component 2	475 MHz	0.021
Component 3	341 MHz	4.32
Component 4	461 MHz	0.034
Component 5	326 MHz	3.97
Component 6	454 MHz	0.035
Component 7	313 MHz	4.3
Component 8	432 MHz	0.041
Component 9	308 MHz	4.56
Component 10	300 MHz	1.62
Component 11	300 MHz	1.62
Component 12	375 MHz	0.91
Our work	263 MHz (1.31 ×)↑	56.67 (1.02×) ↑

Fig. 7: Performance Exploration of VGG

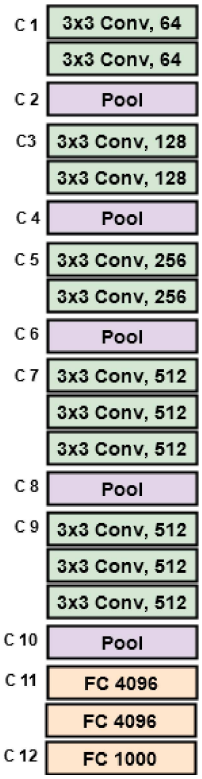


Fig. 8: VGG architecture with labelled components

To show our approach's performance, we compare our implementation of VGG-16 with state-of-the-art accelerators, as shown in Table IV. Due to differences in technology, hardware resources, and system setup, it is hard to make an

	LeNet							
Layers	**Full Network**	**Conv1**	**Pool1+ReLU1**	**Conv2**	**Pool2+ReLU**	**FC 1**	**FC 2**	**Our work**
Frequency (Mhz)	375	562	633	475	588	497	543	437 (1.75X) ↑
Latency (ns)	249.7	37.33	12.93	63.46	22.51	49.32	25.05	219.10

TABLE III: Performance Exploration of LeNet

apple to apple comparison between different implementations. However, we will list some recent works for qualitative reference in Table IV. McDanel et al. [13] have the lowest latency. They can achieve such performance because they use a Selector-Accumulator (SAC) for Multiplication-free Systolic Array. It reduces the number of operations by which 92× for VGG-16. We want to highly that the SAC implementation for a systolic array can also be used to pre-implement the components to achieve competitive results. Those results also show us that the pre-implemented flow does not significantly improve the overall latency if each component is not latency-optimized individually. In fact, the pre-implemented flow can even worsen the latency if additional FF are added on critical paths to meet timing. Nonetheless, in terms of frequency, our implementation has the highest frequency among all other implementations.

	[20]	**[21]**	**[13]**	**Our work**
FPGA chip	XC7Z045	KU460	VC707	Kintex KU060
Max. Frequency	200 MHz	200 MHz	170 MHz	263 MHz
Precision	fixed 16	fixed 16	fixed 16	fixed 16
DSP Utilization	96.2%	38%	4%	76%
Latency (ms)	-	-	2.28	42.68

TABLE IV: VGG-16 Performance Comparison with state-of-art approaches

VI. Conclusion

This paper proposes a pre-implemented flow based on a divide and conquers approach to accelerate model inference n FPGA. The flow takes as input an abstract representation of the CNN model inference to perform model mapping and design checkpoint generating, by assembling pre-implemented CNN components with rapidwright. With the pre-implemented flow, each component is implemented to reach maximum performance. Experiments and results show that our approach shows improvements in terms of latency and maximum frequency, with little to no impact on the number of resources used. There are several aspects that we can investigate to improve the current work. Particularly an optimized and automated floor planning to achieve higher performance. Furthermore, the frequency of the pre-implemented network is bounded by the slowest component of the design. We are planning to investigate optimization approaches to improve the performance of components during the function optimization stage.

References

[1] Xilinx, "Alveo u250 data center accelerator card." https://www.xilinx.com/u250, 2019.

[2] Microsoft, "Project catapult." https://www.microsoft.com/en-us/research/project/project-catapult/, 2018.

[3] Intel, "Intel arria 10 product table." https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/arria-10-product-table.pdf, 2018.

[4] 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.

[5] 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.

[6] D. T. Kwadjo and C. Bobda, "Late breaking results: Automated hardware generation of cnn models on fpgas," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–2, IEEE, 2020.

[7] Xilinx, "Hierarchical design." https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug905-vivado-hierarchical-design.pdf, 2017.

[8] G. Miranda, H. P. L. Luna, G. R. Mateus, and R. P. M. Ferreira, "A performance guarantee heuristic for electronic components placement problems including thermal effects," *Computers & operations research*, vol. 32, no. 11, pp. 2937–2957, 2005.

[9] S. Ma, Z. Aklah, and D. Andrews, "Just in time assembly of accelerators," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 173–178, 2016.

[10] X. Wei, Y. Liang, and J. Cong, "Overcoming data transfer bottlenecks in fpga-based dnn accelerators via layer conscious memory management," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2019.

[11] 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 *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 29, ACM, 2017.

[12] Y. Xing, S. Liang, L. Sui, X. Jia, J. Qiu, X. Liu, Y. Wang, Y. Shan, and Y. Wang, "Dnnvm: End-to-end compiler leveraging heterogeneous optimizations on fpga-based cnn accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[13] B. McDanel, S. Q. Zhang, H. Kung, and X. Dong, "Full-stack optimization for accelerating cnns using powers-of-two weights with fpga validation," in *Proceedings of the ACM International Conference on Supercomputing*, pp. 449–460, 2020.

[14] S. Mittal, "A survey of fpga-based accelerators for convolutional neural networks," *Neural computing and applications*, pp. 1–31, 2020.

[15] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "An automatic rtl compiler for high-throughput fpga implementation of diverse deep convolutional neural networks," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, IEEE, 2017.

[16] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 535–547, IEEE, 2017.

[17] S. Hussain, M. Javaheripi, P. Neekhara, R. Kastner, and F. Koushanfar, "Fastwave: Accelerating autoregressive convolutional neural networks on fpga," *arXiv preprint arXiv:2002.04971*, 2020.

[18] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[19] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

[20] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "Dnnbuilder: an automated tool for building high-performance dnn hardware accelerators for fpgas," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, IEEE, 2018.

[21] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.