Journal of Parallel and Distributed Computing 167 (2022) 123–135

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

Towards a component-based acceleration of convolutional neural networks on FPGAs

Danielle Tchuinkou Kwadjo *, Erman Nghonda Tchinda, Joel Mandebi Mbongue, Christophe Bobda

Electrical and Computer Engineering Department, University of Florida, Gainesville, 32603, FL, USA

ARTICLE INFO

Article history:
Received 26 July 2021
Received in revised form 21 January 2022
Accepted 24 April 2022
Available online 6 May 2022

MSC:
0000
1111

Keywords:
FPGA
Data flow graph
CNN inference
Pre-implemented flow

ABSTRACT

In recent years, Convolution Neural Networks (CNN) have been extensively adopted in broad Artificial Intelligence (AI) applications and have demonstrated ability and effectiveness in solving learning problems. However, developing high-performance hardware accelerators on Field Programmable Gate Array (FPGA) for CNNs often demands skills in hardware design and verification, accurate distribution localization, and long development cycles. Besides, the depth of CNN architectures increases by reusing and replicating several layers. In this work, we take advantage of the replication of CNN layers to achieve improvement in design performance and productivity. We propose a programming flow for CNNs 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 minimum of resources, predict the performance, and gain in productivity since there is no need to synthesize any Hardware Description Language (HDL) source code. Furthermore, the pre-implemented components are reused for different range of applications, reducing the engineering time. Through prototyping, we demonstrate 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 higher energy efficiency.

1. Introduction

In recent years, the implementation of Convolutional Neural Networks (CNN) on Field Programmable Gate Arrays (FPGAs) has drawn considerable attention as the need for more efficiency and accuracy is mitigated by the rapid increase in computational cost. CNNs achieve higher quality of result (QoR) at the cost of significant computing and memory requirements due to their deep topological structures, complicated neural connections, and massive data to process [15,7]. As a result, to keep up with the versatile performance needs of CNN applications in several domains like image and video processing, hardware accelerators such as Application-Specific Integrated Circuits (ASICs), FPGAs, and Graphics Processing Units (GPUs) are increasingly used to improve the throughput of the CNN network. Though ASICs and GPUs have long been the default solution to speed up computation in high-performance computing platforms, FPGAs are now a rising trend, mainly due to their performance/watt advantage over GPUs and flexibility over ASICs [3,8]. Furthermore, the continuous 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 [47,27,12]. 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 computer-aided design (CAD) tools capable of extracting application and domain-specific features to leverage the resources available in high-end FPGAs. As the complexity of FPGA architectures increases, so is the 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 primarily general-purpose. They attempt to provide acceptable results on various applications, which may not exploit application and domain-specific characteristics to deliver higher QoR.

This paper presents a divide-and-conquer design flow that enables application and domain-specific optimization on the design of CNN architectures using Xilinx FPGAs. The proposed approach follows three fundamental steps. The first step consists in dividing the design into smaller components. The granularity of the sub-components is left at the designer's choice. Next, each com-

* Corresponding author.
E-mail address: dtchuinkoukwadjo@ufl.edu (D. Tchuinkou Kwadjo).

https://doi.org/10.1016/j.jpdc.2022.04.025
0743-7315/

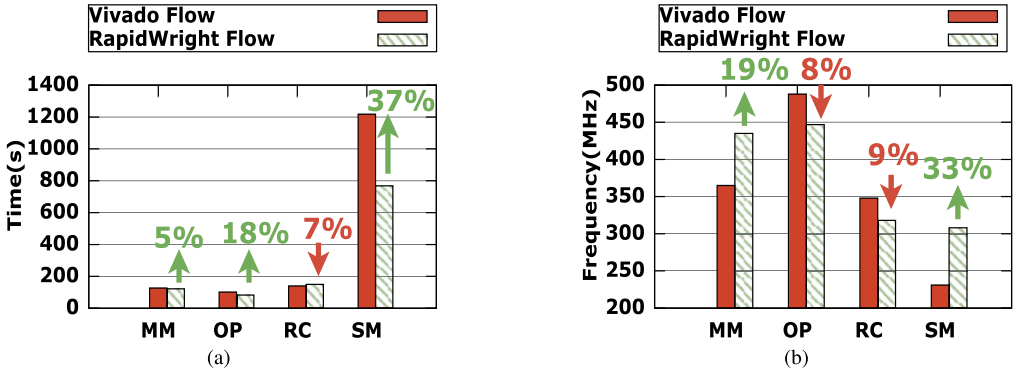


Fig. 1. Motivation example. (a) Compilation time comparison. (b) F_{max} comparison. The results from previous research show that the pre-implemented design flow with RapidWright can lead to improved productivity and QoR compared to the traditional design flow with Vivado [25] (MM=Matrix Multiplication; OP=Outer Product; RC=Robert Cross; SM=Smoothing).

ponent is synthesized and implemented. Finally, we generate the targeted CNN architecture by assembling pre-built components with minimal QoR loss to achieve various design goals such as decreased latency, reduced power consumption, optimized maximum frequency, and low hardware utilization. Recent research has demonstrated that such an approach provides improved QoR than that of the traditional Vivado flow in some instances [18,25,16]. By pre-implementing specific components of a design, higher performance is achieved locally and maintained with minimal loss when assembling the final circuit. This observation is supported by two major considerations: (1) vendor tools such as Vivado tend to deliver high-performance results on small modules in a design [18,17]. (2) Computing applications such as CNN architectures increase in size and complexity by replicating design modules. By carefully selecting principal design modules found in common CNNs, we leverage Vivado optimization to generate highly specialized implementations that can be combined into the desired CNN topology.

As motivation example illustrating the advantages of the pre-implemented flow, Fig. 1 summarizes a few results from the work of Mandebi et al. [25]. It studies the compilation time and maximum frequency achieved when implementing FPGA accelerators for matrix multiplication, outer product, Roberts Cross edge detection, and image smoothing using the Vivado flow and the pre-implemented flow with RapidWright. The results show that the pre-implemented designs flow 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 granularity for the pre-built components, the work demonstrated that pre-implementing modules could significantly improve the QoR when exploiting application and domain-specific features. However, the work only focused on micro kernels and small scale applications [25]. As a result, it remains necessary to evaluate the performance benefits of using the pre-implemented flow on more complex data flow architectures such as CNN accelerators.

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 CNNs. Specifically, our contribution includes:

(1) Reviewing the design of CNN architectures on FPGAs: we will explore the features of state-of-art CNNs that are suitable for FPGA-based acceleration.
(2) Reviewing the pre-implemented flow with RapidWright: we will discuss key steps to follow to efficiently leverage RapidWright in the design of crafted application and domain-specific FPGA accelerators.
(3) A complete component-based framework that maps CNN models to FPGA implementations without tedious HDL programming and verifications, while improving the QoR compared to the traditional design flow with Vivado.
(4) An effective and efficient algorithm is proposed for high-quality and scalable routability-drive placement of components on FPGA

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

2. Overview on CNN FPGA architectures

2.1. Architecture topology

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 [29,7]. They can be reviewed in two categories: *Single Instruction, Multiple Data-based (SIMD) accelerators* and *streaming-based accelerators*. In this section we highlight the potential benefits of designing FPGA-based CNN architectures with the pre-implemented flow, as well as the challenges that may arise. We do not discuss any architecture implementation detail.

In general, the SIMD accelerators start by fetching feature maps and weights from an external memory to on-chip buffers [43,42,49]. The data is then streamed into computing engines composed of several processing elements (PEs). The PEs typically implement general-purpose matrix multiplication circuits in which computations are scheduled to execute the CNN layers in sequence [1,9]. 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 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 [32]. The recurrence of components within CNN architectures makes them suitable candidates for RapidWright implementation as the CNN sub-modules can be optimized for performance, and the achieved performance can be preserved when replicating and relocating the modules across the FPGA. The main advantage of this approach is its flexible, as it fits the implementation of various CNN topologies. However, the resulting CNN architecture has a major limitation. It requires frequent memory transfer between FPGA on-chip scratchpad memories and external memory (DDR/HBM) to fetch the weights and feature maps. Furthermore, the layer-by-layer execution flow makes real-time inference difficult.

Accelerators with the streaming architecture always tailor the hardware with respect to the target network [26,29]. The topology of such CNN accelerators is transformed into a layer-by-layer execution schedule, following the structure of the DAG [23]. Shen et al. [35] note that FPGA-based acceleration used a Convolutional Layer Engine (CLE) to process consecutive CNN layers one at a time. The intermediate results between layers are stored in registers, on-chip memory (OCM), or directly pipelined into the next layer. However, since the dimension as well as filter parameters from consecutive layers might be different, using a fixed CLE for all layers leads to poor performance and inefficient utilization of resources. For an L-layer CNN, they propose using Q CLEs, where $Q < L$, to maximize to BRAM availability for each CLEs. With $Q < L$, some layers are replicated in the design, thus making this architecture suitable for the pre-implemented flow. In the same line of work, a streamed accelerator [26,10] consists on a sequential execution of all the layers of a given CNN. In the same line of work, fpgaConvNet [40,38] framework generates a specialized circuit to run a convolutional neural network given its high-level description. The framework then partitions the network's graph and generates distinct bitstreams for each part of the graph to dynamically configure the FPGA. The architecture analyzes inter-output and kernel parallelism. Given the area constraints associated with the design, fpgaConvNet shares MAC units to reduce required resources, which creates a trade-off between area and performance. The main advantage of this type of architecture 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 [33]. On the downside, this accelerator architecture cannot scale to arbitrarily large CNNs and can difficulty be applied for embedded devices. It is essentially restricted by available on-chip resources needed to implement compute units for each CNN layer, and critically, the size of OCM required to store the weights.

In this work, we propose a framework to generate an accelerator with a streaming architecture. Regarding such accelerators, several semi-automated design flows have been proposed in the literature. In the section below, we review some of them.

2.2. Component-based approaches

Previous research has shown that pre-implementing specific design components are a practical approach to reduce FPGA compilation time. [19,45]. The placement and routing of complex FPGA designs on large devices is generally time consuming. Therefore, component-based design flow is leveraged to reduce implementation time in FPGA architectures. Just In Time (JIT) [22] compilation supplies the users with Domain Specific Language (DSL) to develop FPGAs, linking the design patterns with precompiled bitstreams at runtime. On top of the FPGAs, the overlay connects all the reconfigurable tiles similar to the switch boxes, but it only routes word-wide data. In the same line of work, [45] presents a block-based compilation framework to reduce compilation time through partial reconfiguration (PR). The FPGA capacity is divided into a set of regions of predefined size. The physical PR regions are dedicated to the user logic. Finally, a packet-switched overlay network provides connectivity between the PR regions. In [44], the authors propose a placement model of pre-implemented components using set theory. The module placer is implemented as a constraint solver, which computes feasible placement positions for relocatable modules. Most of the works presented above use a Network On Chip (NOC) to interconnect the different components/modules, adding additional logic. Furthermore, the number and the size of PR regions are predefined, limiting the flexibility and might lead to the under-utilization of the PR regions by components.

2.3. Design flows

To improve the compute units resources and data movement, several optimizations [36,30] are employed on each layer of a given CNN. FlexCNN [36,30], uses a SW/HW co-design approach to compose an architecture for different types of convolution layers using techniques including dynamic tiling and data layout optimization across different layers. With OpenPose as an application driver, they drive up effective DSP utilization on $3 \times 3 - 1 \times 1$ kernel to reach a lowest latency. Nguyen et al. [30] focus on two layer-specific optimizations: layer-specific mixed data flow and layer-specific mixed precision. The mixed data flow aims to minimize the off-chip access while demanding a minimal OCM resource. The mixed precision quantization achieves both a lossless accuracy and model compression to reduce the off-chip accesses. Finally, a Bayesian optimization approach is used to select the best sparsity for each layer, achieving the best trade-off between accuracy and compression.

Numerous works [50,5,34] present a multi-layer processor approach, in which a dedicated hardware unit processes each layer to maximize the utilization of computing resources. Sine the OCM is not enough for multiple hardware units, the data must be stored in off-chip memory. Therefore, these works require a considerable amount of memory access for data. Even they work fine for shallow networks; it is challenging to scale up to deeper networks.

Streaming dataflow aims to tailor the accelerator in regard to the model topology. In this line of work, Sharma et al. [34] proposed DNNWEAVER, a framework that automatically generates a synthesizable accelerator for a given (DNN,FPGA) pair from a high-level specification in Caffe. To achieve large benefits while preserving automation, DNNWEAVER generates accelerators by using a virtual instruction set to describe a network. The model is then translated into an instruction sequence. The sequence is mapped as hardware FSM states. The design flow in [23] searches the optimized parameter for a handcrafted Verilog template with the input network description and platform constraint, which leads to a uniform mapping of PEs that reduces the accelerator architecture complexity. The acceleration strategy is further generalized for different CNN models with varying dimensions and topology. Ahmad et al. [2] analyze Winograd minimal filtering or fast convolution algorithms to reduce the arithmetic complexity of convolutional layers of CNNs. They propose a pipelined and parallel Winograd convolution engine that improves the throughput and power-efficiency while reducing the computational complexity of the overall system. The proposed techniques focus on automatically generate the HDL design based on the network parameters. The main contribution of this approach is the selection of an intermediate level description of the network to cover the gap between high-level network description and low-level hardware design.

3. Pre-implemented flow with Vivado and RapidWright

In this section, we present RapidWright and describe its integration in common design steps with Vivado. We also elaborate on background concepts such as the "out-of-context" design flow in Vivado. Finally, we provide necessary discussion on the pre-implementation of design components.

3.1. RapidWright

RapidWright [18] is an open source Java framework from Xilinx Research Labs that provides a bridge to Vivado backend at different compilation stages (synthesis, optimization, placement, routing, etc) using design checkpoint (DCP) files as illustrated in Fig. 2. Once a DCP is loaded within RapidWright, the logical/phys-

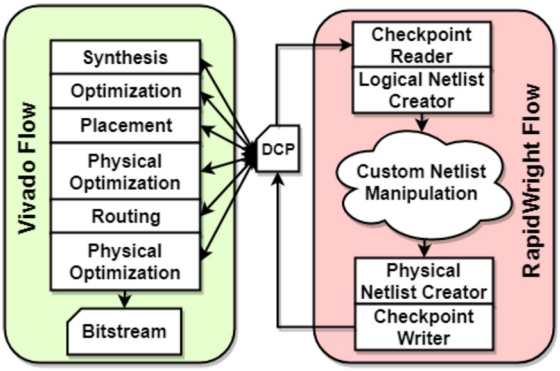


Fig. 2. Vivado and RapidWright interaction.

ical 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 logic and routing resources such as look-up tables (LUT), flip-flops (FF) and programmable interconnect points (PIPs) from a high-level programming language like Java. They also provide means to run some operations such as timing analysis, placement, and routing. Upon completing the netlist manipulation, RapidWright enables storing the changes in a netlist into a new DCP that can directly be loaded into Vivado.

3.2. Out-of-context design flow

The "out-of-context" (OOC) Flow [46] is a netlist generation mode ensuring that the placement of I/O buffers is disabled at compile time to facilitate the design of internal components of an architecture. It has several advantages among which: (1) it allows 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.

3.3. Pre-implementing design components

Vendor CAD tools such as Vivado use heuristics for the physical implementation (placement and routing). They consider the number of cells in a design, their connections, and the physical architecture of the target FPGA device 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 [28]. Focusing the optimization on smaller modules may therefore lead to overall QoR improvement in a design. Furthermore, several works in the literature have shown that pre-implementing components or macros can significantly decrease the overall FPGA compilation time with performance benefits [22,20,25]. The pre-implemented flow therefore aims to generate high-performance implementations by reusing in multiple contexts and chip locations, high-quality and customized pre-built circuits. Using an iterative process, the top-level design can then be constructed by assembling the pre-built circuits with minimal QoR loss.

To fully exploit the benefits of the pre-implemented flow with RapidWright, the design architect must first restructure the CNN HDL codes hierarchically. The reorganization of the HDL sources must consider three main design characteristics [18] that are:

1. **Modularity:** highlights the design structure so that it can be strategically mapped to architectural models.
2. **Module replication:** when modules are replicated, it allows the reuse of high-quality solutions in the design while increasing productivity.
3. **Latency tolerance:** if the modules in a design tolerate additional latency, inserting pipeline elements between them improves both synchronization performance and offshoring.

4. 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 Fig. 3. 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 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.

4.1. Function optimization

This section describes the major steps involved in the design of optimized sub-functions.

4.1.1. Granularity exploration

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

(1) **Convolution:** The convolution layer convolves the input image with a set of learnable filters, each producing one feature map in the output image.
(2) **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.
(3) **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.
(4) **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 4 layers onto the FPGA, the vast majority of forward processing networks can be implemented on the FPGA. The modules implementations should resolve around this minimum of granularity. Automated decomposition of user logic into leaf components is complementary future work.

4.1.2. Performance exploration

We start by manually building the CNN components OOC. 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 to be 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.

Optimization and implementation of components: the architecture of a convolution engine is depicted in Fig. 4a. The input

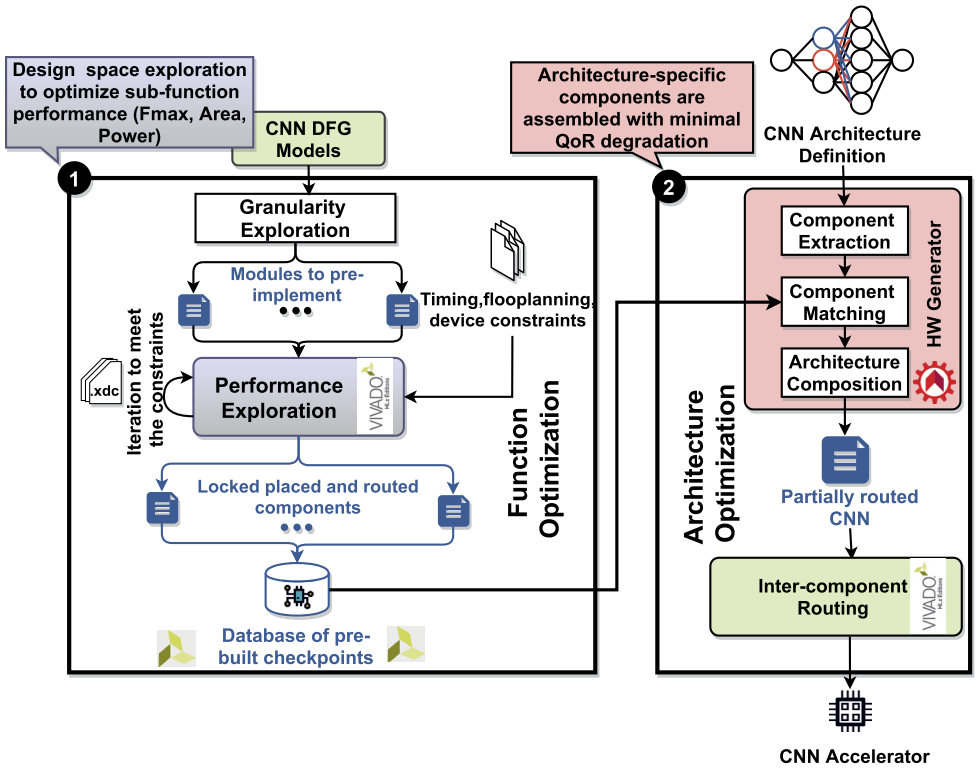
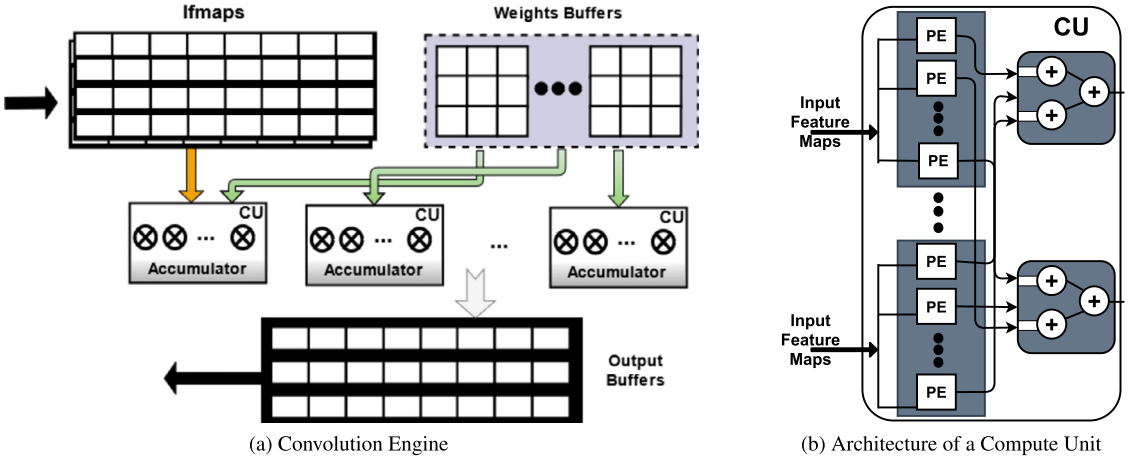


Fig. 3. General overview of the proposed design flow.



(a) Convolution Engine (b) Architecture of a Compute Unit

Fig. 4. Pre-implemented Design Components.

samples are read in a streaming fashion from the local buffer. For a convolution of size K, at least K-1 lines of data must be fetched before the circuit can process the first sample. All the computations performed before this are simply discarded through the use of conditionals. Each line buffer can store up to K+1 data. When a new sample is read, another sample is pushed out of the line buffer. Interestingly, the newest sample is used in the calculation, and then the sample is stored into the line buffer, and the old sample is ejected out. Therefore, it ensures that only K+1 lines must be cached rather than an unknown number of lines and minimize local storage use. Fig. 4b illustrates the Compute unit (CU) data path of a convolution engine. It starts by multiplying the input feature map data and the corresponding weights via the multiple parallel multiplication arrays; then, the final cumulative values are determined by the adder tree in a pipeline. Several slices of CUs constitute the convolution engine. The 27x18 multiplier in the DSP48E2 slice carries two parallel 8-bit multiplications that share one common operand. Precisely, the two INT8 operands are packed into the 27-bit port and then multiplied in parallel by the third, shared INT8 operand in the 18-bit port. The output of the DSP48E2 slice can generate two parallel products in a particular full-precision dual-product format, reducing by half the number of DSP48E2.

The design of the pooling layer is similar to [11] and is depicted in Fig. 5. The designs flow presents an implementation of the max-pooling in a stride of 2 samples. The main components of the architecture are a shift register with only L+2 stages, a comparator core, and a controller. The comparator core consists of three comparators for finding the maximum pixel value in the 2×2 window. This circuit is implemented independently from the convolution engine to accommodate most CNNs topologies. I/O interfaces of components implement a producer-consumer scheme with a glob-

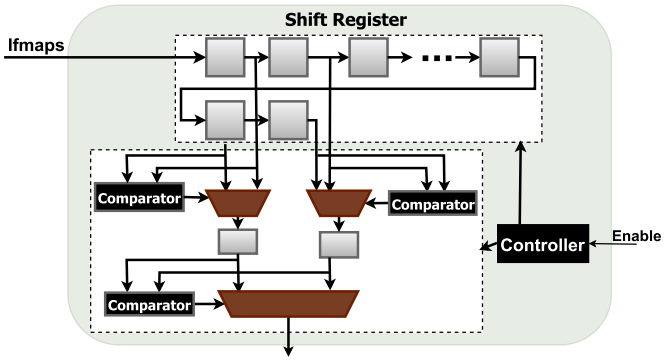


Fig. 5. Max-pooling Computing Engine.

ally asynchronous locally synchronous approach for weight storage, whereby memory resources operate faster than the compute logic. It then allows adjacent layers to start operation before the producer layers are completed. Each layer feeds its output to the next layer using similar datatype layouts and allows it to overlap in their operation once enough data has been accumulated in the previous layer. There is no need to store each layer's intermediate results in off-chip memory with a streaming architecture since they are directly piped down the stream.

To achieve high QoR in the performance exploration phase, the implementation of components follows the following design considerations:

(a) **Strategic floorplanning:** utilizing pblock constraints allows carefully selecting the FPGA resources that will be used by each design component. It helps improving the module-level performance and area. Hence, the designer has the possibility to only use necessary resources as opposed to letting the CAD tool utilize as many chip tiles as it wants. Given that Xilinx architectures generally replicate the resource structures (CLBs, DSPs, BRAM, URAM, etc) over an entire column of clock regions (see Fig. 8), 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.

(b) **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.

As example, let us consider a design in which we pre-implement the two sub-modules *Module1* and *Module2*. In order 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. In fact, the maximum frequency is proportional to the highest delay on the timing paths. We must therefore reduce the length of the net between the modules by ensuring that the cells at the interface of the pre-built components are placed near the edge of the pblocks of the 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) [46] to specify the interconnect tiles that will route the nets connecting to the other modules of a design. Fig. 6a presents a possible outcome of pre-implementing the two sub-modules without considering the port planning. The distance between the two FFs that are identified by the green and red marks may become a source high delay, resulting in reduced maximum frequency. On the other hand, Fig. 6b shows that partpins make it possible to shorten the length of the net, which decreases the F_{max} degradation when connecting the components in a design.

(c) **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.

(d) **Logic locking:** the main 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 the productivity.

(e) **Checkpoint file generation:** pre-implemented modules are stored in the form of DCPs. The top-level design will then implement synthesis black-boxes that will be filled by the optimized pre-built modules.

The implementation here is done using vendor tools and considers several constraints such as timing and floor planning. The pblock 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 not control on how the I/O ports are placed. With pblocks and timing constraints, I/O ports might be contained anywhere in the pblock resulting in routing congestion and timing issues around I/O interfaces when generating the whole design as described in Section 4.2.

4.2. Architecture optimization

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

4.3. Hardware generation

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.

4.3.1. 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 designed 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

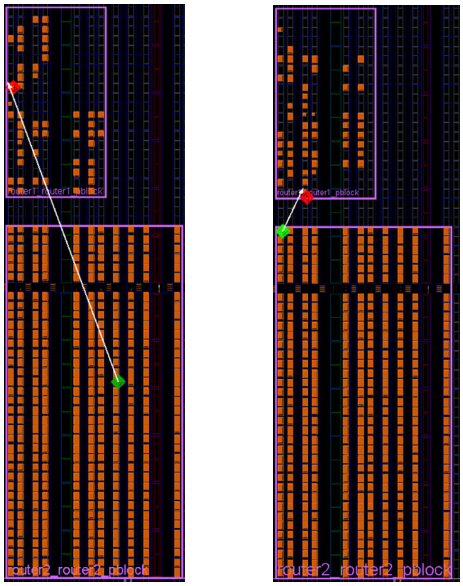


Fig. 6. Illustration of the importance of port planning. (a) Without PartPins, the FFs are placed randomly, resulting in a high distancing. (b) Defining PartPins provides context to Vivado when pre-implementing the modules, which results in shorter distance between the FFs.

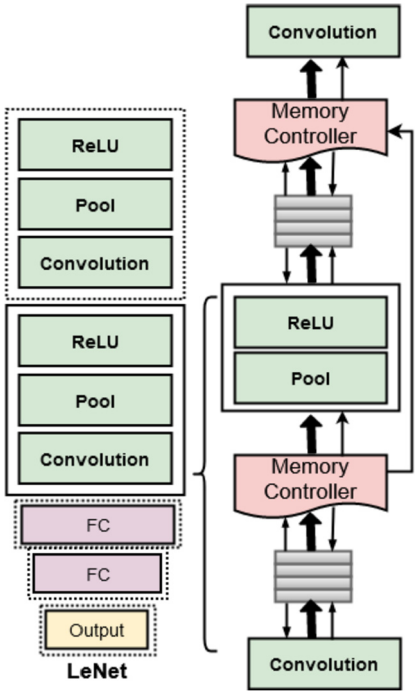


Fig. 7. Communication Interface between Components.

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 $\times$ 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 Fig. 7. That constraint is not required for the following ReLu, and the operation can be directly applied to intermediate results of the pooling layers.

4.3.2. 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 input width and height, the number of channels, the kernel size, the padding, and the strike. 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 database of pre-built checkpoints to compose the final architecture.

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. As shown in Fig. 7, components are pre-implemented with two interfaces. The first interface called “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 OCM. 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 netlist of the design 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.

4.3.3. 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. CLB slices are organized in a regular array. Each connects to a switch box to access general routing resources, which run vertically and horizontally between rows and columns. The device is surrounded by I/O Blocks allowing off-chip connections. DSP blocks and BRAMs are arranged in columnar-wise and spread across the device. We aim to find a congestion-aware timing-driven placement for components of the input graph.

Problem formulation Given an Utrascale FPGA with logic elements, its architecture, and a graph G of components, we need to map the component's netlist to the logic elements of the FPGA and determine their positions to minimize routed wirelength and congestion. In summary, (1) each component must be assigned to a valid position on the FPGA, and (2) the placement legalization rules of each tile are satisfied.

The algorithm works as follows: we recursively parse the input graph and place the first component. Since components are pre-implemented within pblocks, the number of resources used and allocated is reported. For each adjacent component, we assign a location on the FPGA grid, with minimal interconnect wire length, i.e., the estimated half-perimeter wire length (HPWL) from the placed cells locations. To fulfill that requirement, we define timing and congestion cost functions to evaluate the cost of the assigned location.

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

Algorithm 1: Hardware generation Algorithm.

```
   Input  : Design d, Graph G, Node root
   Output: DCP file
 1 let Q be a queue ;
 2 let t be the max number of iterations ;
 3 mark root as discovered ;
 4 Q.enqueue(root) ;
 5 while Q.size() != 0 do
 6     Node v = Q.dequeue() if v is the goal then
 7         return v ;
 8     end
 9     ParseGraph (v, G, Q, 0);
10 end

11 Function ParseGraph(G, v, Q, iter):
12     q = locate an optimal placement for v;
13     addNodeToDesign(q);
14     Nodes w = G.Next();
15     foreach edges from v to w do
16         if w is not marked then
17             p = locate an optimal placement for w;
18             addNodeToDesign(p);
19             Ports ports_v = selectPortOfInterest (v);
20             Ports ports_w = selectPortOfInterest (w);
21             nets = create_nets (ports_v, ports_w, v, w);
22             timing, cost = TimingEstimation (nets, v, w);
23             if (not (timing & cost) & (iter < t)) then
24                 iter ++;
25                 p = G.Previous();
26                 ParseGraph (G, p, Q);
27             end
28             mark w as discovered ;
29             w.parent = v ;
30             Q.enqueue(w) ;
31         end
32     end
33 End Function

34 Function TimingEstimation(nets, v, w):
35     foreach net ∈ nets do
36         if size(net) > 1 then
37             timing = getTileSize();
38         else
39             timing = timing_cost(nets);
40         end
41         cost = cgt_cost(v, w);
42     end
43     return (timing, cost)
44 End Function
```

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

Where $W_{i,j}$ is the wire between component i and j (distance from physical net's source pin to sink pin). A fan-out greater than one will in most cases, have some branching farther (reusing a path). In this case, the unit of length is the dimension size of a tile.

Congestion estimation for optimal routing, a placement algorithm must consider the number of resources used by each inter-component nets 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. Furthermore, the number of switch boxes to traverse factor into the total delay [48]. The algorithm tries to build a solution incrementally, one component at a time, removing those solutions that fail to satisfy the problem's constraints at any point in time. A placement is validated if the costs are lower than a defined threshold. Otherwise, for each previously placed component, we unplaced them, find another location, until the costs are satisfied. After a certain number of predefined iterations, if not placement satisfying the constraints

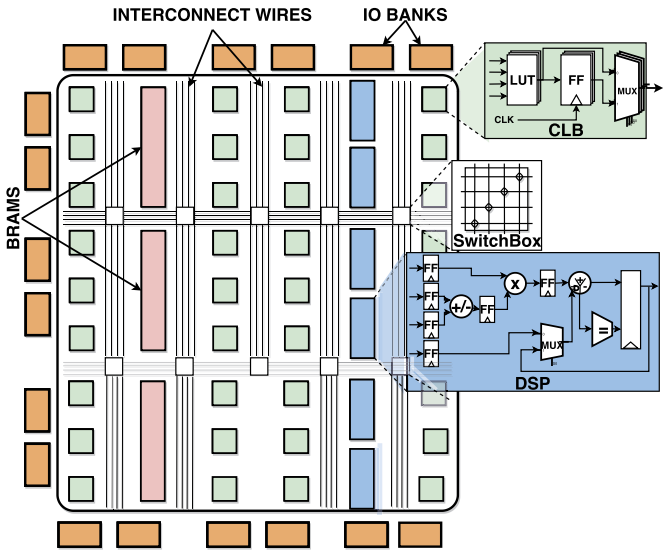


Fig. 8. Xilinx Ultrascale Architecture.

is found, we move to the next component. The proposed solution works as follows: we recursively parse the input graph and place the first component (Line 3-13). For each adjacent component, we assign a valid location on the FPGA grid (Line 14-18). We evaluate the cost of the current assignment (Line 22). If the placement satisfies the constraints (timing and congestion cost), we move to the next component (Line 28-30). Otherwise, we recursively unplace previous components and try to find another location (Line 23-27).

$$\begin{aligned} cgt_{coef} &= \sum_{i} C_i \\ cgt\_cost &= \sum_{W_{i,j}} \frac{\omega_{i,j} \times cgt_{coef_k}}{\#SwBox} \end{aligned} \tag{2}$$

Where C_i is the number of components overlapping within a $tile_k$, $\omega_{i,j}$ is a weight proportional to the number of pins of wire $W_{i,j}$, and $\#SwBox$ is the number of switch boxes traversed by the nets.

Although RapidWright provides a lightweight timing model [24], it works at the BEL level, which is more fine-grained than what is required here.

4.3.4. Datapath regularization

To reduce overall latency and data management overhead, datapaths must be regularized. Each component comes with its own latency in number of clock cycles. We must therefore ensure that operands arrive at the boundary of each component at the same time to expect correct results. This task is done by inserting FFs on the critical path. Inserting FFs do not increase overall latency as the number of FFs is the cumulative latency of operations on the datapath.

4.3.5. Inter-component routing

After the architecture composition, the design contains all the necessary CNN modules. Each design module still has the logic and the internal routing locked. However, the RapidWright hardware generator 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. We, therefore, utilize Vivado for the final routing, which essentially consists in finding FPGA interconnects to implement the logic routes created within RapidWright in a way that minimizes timing delays.

Table 1
FPGA Resource Utilization.

	CLB LUTs	CLB Registers	BRAMs	DSPs
Data Precision	8 bit			
LeNet	17005 (7.90%)	7591 (1.61%)	109 (18.44%)	121 (5.22%)
Pre-implemented LeNet	14533 (14.89%) ↓	6847 (9.57%) ↓	104 (5.43%) ↓	121.00 ⇔
VGG-16	100767 (41.35%)	151646 (38.80%)	462 (37.84%)	986 (50.83%)
Pre-implemented VGG-16	85204 (15.44%) ↓	121879 (11.25%)↓	437 (2.07%)↓	935 (1.12%) ↓

5. Experimental results

5.1. Evaluation platform and setup

For evaluation purposes, designs are implemented on a Xilinx FPGA XCKU060. The hardware is generated using Vivado v2019.2 and RapidWright v2019.1, and the components are implemented with vivado HLS. The hardware generation is conducted on a computer equipped with an Intel Corei7-9700K CPU@3.60GHz×4 processor and 32GB of RAM.

5.2. Benchmarks

We study two CNN architectures: LeNet [21] and VGG [37]. We run applications individually with the purpose of assessing achievable performances, in particular: (1) global latency, (2) Maximum Frequency (Fmax) and productivity and (3) resource utilization, when comparing pre-implemented to full implemented CNNs. For both networks, we use a 8 bit data precision. Table 2 presents the different the workload and the requirement in terms of memory of the two networks. We compare the pre-implemented circuits to the corresponding classic implementation. Here, a classic implementation refers to a circuit generated from a single top level file, following the vivado design flow [13].

Table 2
Computational hardware resources for state-of-art DNNs.

	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

5.2.1. LeNet architecture

It's 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 hard coded in ROM. This choice has been decided out of simplicity

5.2.2. VGG-16 architecture

VGG consists of 16 convolutional layers and is very appealing for the pre-implemented flow because of its very uniform architecture. Input images are passed through a stack of convolutional layers with the fixed filter size of 3×3 and the stride of 1. There are five max pooling filters built-in between convolutional layers. The stack of convolutional layers is followed by 3 fully connected layers. Each convolution layer is made of 2 or 3 convolutions with same parameters, followed by a pooling layer. The replicability of layers within VGG suits the pre-implemented flow. With 124 M of weights, there is not enough resources on-chip to store them. We use off-chip memory to store the coefficient data and data layout configuration files. The coefficient data files contain the parameters of each layer and the data layout configuration files include the size of the input feature map and the output feature map, as well as the shape of the tensor coefficients. 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 size of the block, 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.

5.3. Resource utilization

Pre-implementing basic components have the potentiality of reducing resource utilization as shown in Table 1. The classic implementation LeNet and VGG use respectively 7.90% and 41.35% of LUTs, 1.61% and 17.80% of registers. The pre-implemented version of LeNet uses less than 14.89% of LUTs and 9.57% of registers (resp. VGG uses less 9.09% of LUTs and 11.255% of registers) when compared to the classic implementation. Overall, the pre-implement networks use less resources than the baseline implementation. When the design is small, vivado can provide a better optimization of the resources. Furthermore, when pre-implementing components, we define pblocks, which limit the amount of resources that vivado can use and hence, forcing some area optimizations. When the design is bigger, vivado tends to maximize the capacity of adaptation and becomes difficult to capture all its specificities. A snapshot of Lenet of the FPGA fabric is shown in Fig. 10.

LeNet uses 18.44% of the BRAM available on the chip. This is simply because the weights and biases are hard coded in ROM and uses more resources. The pre-implemented LeNet (resp pre-implemented VGG) uses 5.43% less BRAM (resp. 37.84%). Vivado can optimize individual component IR without BRAM insertion while adding such resources when compiling bigger design, which translates into a higher power consumption. The amount of DPS is the same for LeNet implementation. However, we notice a slight decrease of 1.12% the pre-implemented VGG. By defining pblock 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.

5.4. Productivity

With the continuous growth of CNNs parameters and depth, improving the productivity is an important factor when it comes

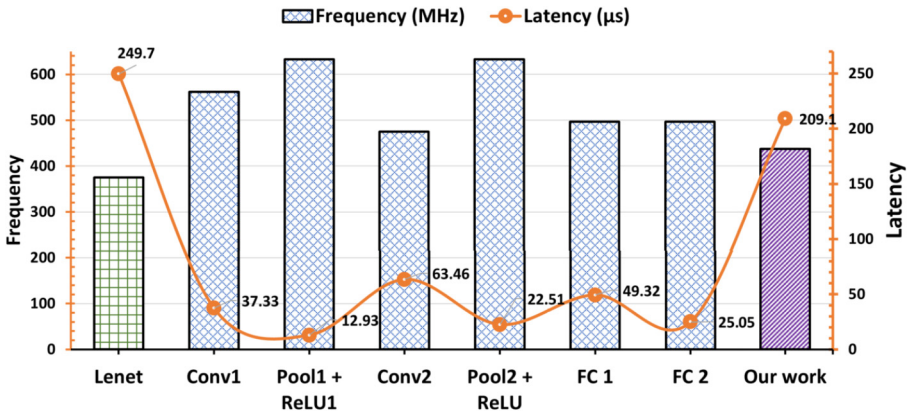
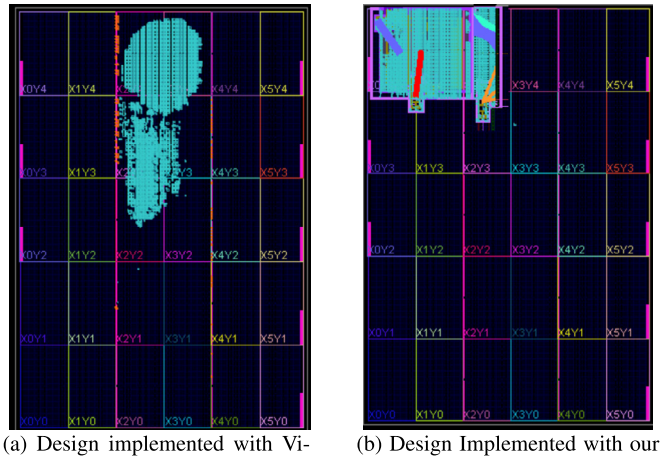


Fig. 9. Performance Exploration of LeNet.

Table 3
Energy Efficiency Comparison.

	CPU	Cloud-DNN [6]	Caffeine [51]	Biookaghazadeh et al. [4]	Zhu et al. [52]	Pang et al. [31]	Ours
FPGA Device	-	XC7Z045	KU060	Arria 10	ZCU102	VC707	KU060
Data Precision	float	16	16	8/16	16	16	8
Power (W)	140	49.25	26	23	23.6	8.152	25.2
Energy Efficiency (GOPS/W)	1.36	37.13	28.85	39.1	13.05	23.1	42



(a) Design implemented with Vivado

(b) Design Implemented with our approach

Fig. 10. Lenet circuit on FPGA fabric.

Table 4
Design Generation Time for implementation of LeNet and VGG with vivado and the pre-implemented flow in seconds.

	LeNet Pre-Implemented Flow		Classic LeNet	
	RapidWright	Inter-node Routing	Placement	Routing
Time (min)	0.84	12.7	18.32	25.67
Ratio	6.2%	96.54%	39.6%	60.4%
Total (min)	**13.54** (69% ↑)		**45**	
	VGG Pre-Implemented Flow		Classic VGG	
	RapidWright	Routing	Placement	Routing
Time (min)	5.27	35.67	8.77	128.23
Ratio	8.00%	92.00%	6.40%	93.60%
Total (min)	**41.94** (61% ↑)		**137.00**	

to hardware design. In this section, we show how the proposed flow can leverage component reuse to reduce both compile time and implementation cycles. Table 4 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. For the networks that are pre-implemented, since components have already implemented off-line, we only measure DCP generation with rapidwright and inter-component routing with vivado. With the pre-implemented flow, it takes 13.54 min (resp. 41.94 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 6.2% (resp. 8%) of the total time. RapidWright has minimal impact on the productivity. The biggest portion of the time is used to route the nets between components.

5.5. Performance

This section presents a comparison with FPGA designs that utilize a batch size of 1, and we report simultaneously latency and Frequency. In Fig. 9, we present the performance of each component as well as the pre-implemented LeNet. Overall, LeNet achieves up to 1.2× 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. Furthermore, the frequency of the pre-built design is upper bounded by the slowest component in the design. Fig. 9 also present the variation of the latency in micro seconds (µs) of each component. The pre-implemented LeNet reaches a 16.3% lower latency.

The pre-implementing VGG has 1.17× higher frequency than the baseline VGG implementation, with a 23.19% lower latency (Figs. 11, 12). Hence, a given design reflecting the properties of modularity, module replication, and latency tolerance, a circuit generated with our approach will have better performance than the classic implementation. In contrary to LeNet, VGG has more

Table 5
VGG-16 Performance Comparison with state-of-art approaches.

	Biookaghazadeh et al. [4]	Super-LIP [14]	ELB-NN [41]	Caffeine [51]	McDanel et al. [26]	fpgaConvNet [40]	Venieris et al. [39]	Cloud-DNN [6]	Ours
FPGA Device	Arria 10	ZCU102	ZC706	KU060	VC707	Zynq-7045	XC7Z045	XC7Z045	KU060
Architecture	SIMD	Dataflow	Dataflow	SIMD	SIMD	Dataflow	Dataflow	SIMD - Dataflow	Dataflow
Topology									
Fmax (MHz)	212	200	200	200	170	125	125	214	263
Data Precision (bit)	8 - 16	16	4	8	5	16	16	16	8
DSP	-	57.87%	33% (298)	38% (1058)	4% (112)	95%	100% (900)	78%	48%
BRAM	-	92.43%	93 % (509)	56% (782)	81% (834)	-	-	74.4%	36%
LUTs	-	-	52% (112992)	60% (200K)	78% (239K)	-	89% (216.60 K)	58.5%	40%
Latency (ms)	26.52 - 30.3	71.46	5.84	25.3	2.28	249.50	249.50	16.92	8.51
Throughput (GOPS)	990	-	3.3 TOPS	365	-	155.81	123.12	1828.61	1059

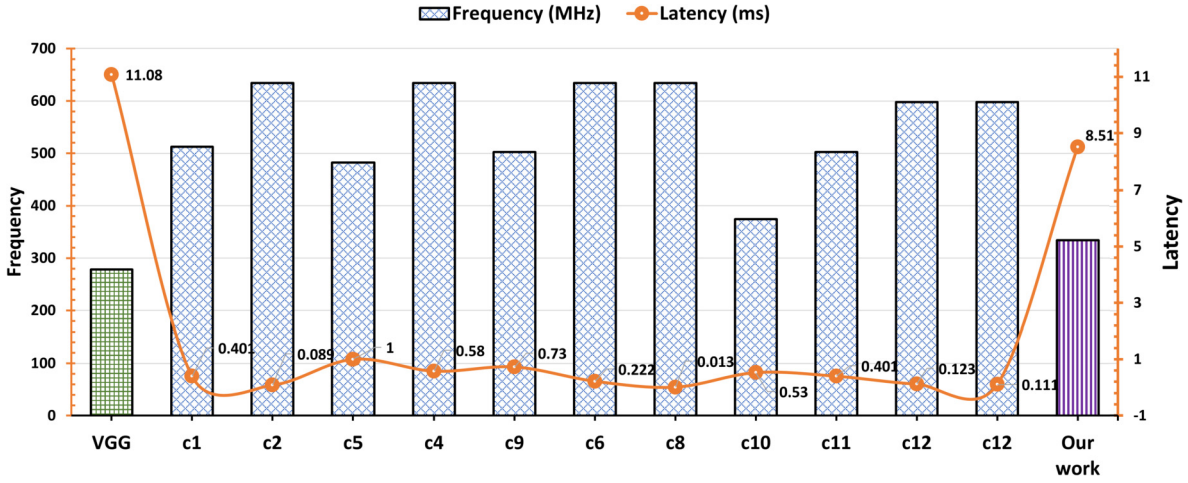


Fig. 11. Performance Exploration of VGG.

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. Even with a projecting higher latency, the proposing flow succeed on providing better performance.

To show the performance of our approach, we compare our implementation of VGG-16 with state-of-the art accelerators in Table 5. For each work, we report the architecture topology, data precision, resource utilization and throughput in GOPS. Due to differences in technology, hardware resources and system setup, it is hard to make an apple to apple comparison between different implementations. But we list some recent works for qualitative reference. The latency here represents the time it takes for a single frame inference. McDanel et al. [26] 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 highly that the SAC implementation can also be used to pre-implement the components to achieve competitive results. When it comes to the throughput, ELB-NN [41] has the highest performance of 3.3 TOPS with ultra-low data precision of 4 bit. Despite impressive throughput, the accuracy of the proposed circuit drops to 55.8%. Our work achieves a throughput of 1059 GOPS, which is lower than Cloud-DNN and ELB-NN. Nevertheless, it uses less than 50% of the FPGA fabric, with 2× lower latency than Cloud-DNN. Overall, our paper has the best ration performance/resources, with the highest frequency.

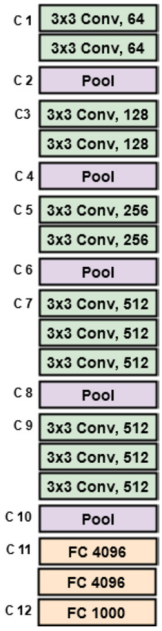


Fig. 12. VGG architecture with labelled components.

We also compare the FPGA energy efficiency to the existing designs on FPGAs and CPU (Table 3). For fair comparison, we use GOP/WS as the standard metric. Our implementation using 8-bit fixed-point has the highest energy efficient over a batch of 1. Despite having a higher frequency than most of the design, our im-

plementation has the smallest number of resources, which results in lower power consumption.

6. Conclusion

This paper proposes a pre-implemented flow based on a divide and conquers approach to accelerate model inference on 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 offers improvements in terms of latency and maximum frequency, with little to no impact on the number of resources used. Our workflow is designed in a modular fashion, allowing easy integration for new layer types.

However, there are still several aspects that we plan to investigate with the goal of improving the current work, such as supporting a more exhaustive range of DNNs. Particularly an optimized and automated floor planning to achieve higher performance. Furthermore, the maximum 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. Furthermore, the input to this framework is a “CNN architecture definition” we are working on extending our current flow to support other frameworks like ONNX and PyTorch. We also plan to expand our approach to utilize multiple FPGAs with larger models in the future.

CRediT authorship contribution statement

Danielle Tchuinkou Kwadjo: Conceptualization, Methodology, Software. **Erman Nghonda:** Software, Writing – Reviewing. **Joel Mandebi Mbongue:** Validation, Writing – Reviewing and Editing. **Christophe Bobda:** Supervision, Resources.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work was funded by the National Science Foundation (NSF) under Grant CNS 2007320.

References

[1] K. Abdelouahab, M. Pelcat, J. Serot, F. Berry, Accelerating cnn inference on fpgas: a survey, preprint, arXiv:1806.01683, 2018.
[2] A. Ahmad, M.A. Pasha, Towards design space exploration and optimization of fast algorithms for convolutional neural networks (cnns) on fpgas, in: 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), IEEE, 2019, pp. 1106–1111.
[3] P. Bhowmik, J.H. Pantho, J.M. Mbongue, C. Bobda, Esca: event-based split-cnn architecture with data-level parallelism on ultrascale+ fpga, in: 2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), IEEE, 2021, pp. 176–180.
[4] S. Biookaghazadeh, P.K. Ravi, M. Zhao, Toward multi-fpga acceleration of the neural networks, ACM J. Emerg. Technol. Comput. Syst. 17 (2) (2021) 1–23.
[5] M. Blott, T.B. Preußer, N.J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser, K. Vissers, Finn-r: an end-to-end deep-learning framework for fast exploration of quantized neural networks, ACM Trans. Reconfigurable Technol. Syst. 11 (3) (2018) 1–23.
[6] Y. Chen, J. He, X. Zhang, C. Hao, D. Chen, Cloud-dnn: an open framework for mapping dnn models to cloud fpgas, in: Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2019, pp. 73–82.
[7] Y. Chen, Y. Xie, L. Song, F. Chen, T. Tang, A survey of accelerator architectures for deep neural networks, Engineering 6 (3) (2020) 264–274.
[8] J. Fowers, G. Brown, P. Cooke, G. Stitt, A performance and energy comparison of fpgas, gpus, and multicores for sliding-window applications, in: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 2012, pp. 47–56.
[9] P. Haghi, T. Geng, A. Guo, T. Wang, M. Herbordt, Fp-amg: fpga-based acceleration framework for algebraic multigrid solvers, in: 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), IEEE, 2020, pp. 148–156.
[10] S. Hussain, M. Javaheripi, P. Neekhara, R. Kastner, F. Koushanfar, Fastwave: accelerating autoregressive convolutional neural networks on fpga, preprint, arXiv:2002.04971, 2020.
[11] W.-J. Hwang, Y.-J. Jhang, T.-M. Tai, An efficient fpga-based architecture for convolutional neural networks, in: 2017 40th International Conference on Telecommunications and Signal Processing (TSP), IEEE, 2017, pp. 582–588.
[12] 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.
[13] D.I.S.U. IP, Vivado design suite user guide, UG892 (v2020.2), 2021.
[14] W. Jiang, E.H.-M. Sha, X. Zhang, L. Yang, Q. Zhuge, Y. Shi, J. Hu, Achieving super-linear speedup across multi-fpga for real-time dnn inference, ACM Trans. Embed. Comput. Syst. 18 (5s) (2019) 1–23.
[15] H. Kung, B. McDanel, S.Q. Zhang, X. Dong, C.C. Chen, Maestro: a memory-on-logic architecture for coordinated parallel use of many systolic arrays, in: 2019 IEEE 30th International Conference on Application-Specific Systems, Architectures and Processors (ASAP), vol. 2160, IEEE, 2019, pp. 42–50.
[16] D.T. Kwadjo, J.M. Mbongue, C. Bobda, Performance exploration on pre-implemented cnn hardware accelerator on fpga, in: 2020 International Conference on Field-Programmable Technology (ICFPT), IEEE, 2020, pp. 298–299.
[17] D.T. Kwadjo, J.M. Mbongue, C. Bobda, Exploring a layer-based pre-implemented flow for mapping cnn on fpga, in: 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, 2021, pp. 116–123.
[18] C. Lavin, A. Kaviani, Rapidwright: enabling custom crafted implementations for fpgas, in: 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), IEEE, 2018, pp. 133–140.
[19] C. Lavin, M. Padilla, S. Ghosh, B. Nelson, B. Hutchings, M. Wirthlin, Using hard macros to reduce fpga compilation time, in: 2010 International Conference on Field Programmable Logic and Applications, IEEE, 2010, pp. 438–441.
[20] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, B. Hutchings, Hmflow: accelerating fpga compilation with hard macros for rapid prototyping, in: 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines, IEEE, 2011, pp. 117–124.
[21] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, Proc. IEEE 86 (11) (1998) 2278–2324.
[22] S. Ma, Z. Aklah, D. Andrews, Just in time assembly of accelerators, in: Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2016, pp. 173–178.
[23] Y. Ma, Y. Cao, S. Vrudhula, 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), IEEE, 2017, pp. 1–8.
[24] P. Maidee, C. Neely, A. Kaviani, C. Lavin, An open-source lightweight timing model for rapidwright, in: 2019 International Conference on Field-Programmable Technology (ICFPT), IEEE, 2019, pp. 171–178.
[25] J.M. Mbongue, D.T. Kwadjo, C. Bobda, Automatic generation of application-specific fpga overlays with rapidwright, in: 2019 International Conference on Field-Programmable Technology (ICFPT), IEEE, 2019, pp. 303–306.
[26] B. McDanel, S.Q. Zhang, H. Kung, 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, 2020, pp. 449–460.
[27] Microsoft, Project catapult, https://www.microsoft.com/en-us/research/project/project-catapult/, 2018.
[28] G. Miranda, H.P.L. Luna, G.R. Mateus, R.P.M. Ferreira, A performance guarantee heuristic for electronic components placement problems including thermal effects, Comput. Oper. Res. 32 (11) (2005) 2937–2957.
[29] S. Mittal, A survey of fpga-based accelerators for convolutional neural networks, Neural Comput. Appl. (2020) 1–31.
[30] D.T. Nguyen, H. Kim, H.-J. Lee, Layer-specific optimization for mixed data flow with mixed precision in fpga design for cnn-based object detectors, IEEE Trans. Circuits Syst. Video Technol. (2020).
[31] W. Pang, C. Wu, S. Lu, An energy-efficient implementation of group pruned cnns on fpga, IEEE Access 8 (2020) 217033–217044.
[32] M.J.H. Pantho, P. Bhowmik, C. Bobda, Towards an efficient cnn inference architecture enabling in-sensor processing, Sensors 21 (6) (2021) 1955.
[33] L. Petrica, T. Alonso, M. Kroes, N. Fraser, S. Cotofana, M. Blott, Memory-efficient dataflow inference for deep cnns on fpga, in: 2020 International Conference on Field-Programmable Technology (ICFPT), IEEE, 2020, pp. 48–55.
[34] H. Sharma, J. Park, D. Mahajan, E. Amaro, J.K. Kim, C. Shao, A. Mishra, H. Esmaeilzadeh, From high-level deep neural models to fpgas, in: The 49th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Press, 2016, p. 17.

[35] Y. Shen, M. Ferdman, P. Milder, Maximizing cnn accelerator efficiency through resource partitioning, in: 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), IEEE, 2017, pp. 535–547.
[36] A. Sohrabizadeh, J. Wang, J. Cong, End-to-end optimization of deep learning applications, in: Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2020, pp. 133–139.
[37] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015, pp. 1–9.
[38] S.I. Venieris, C.-S. Bouganis, fpgaConvNet: a framework for mapping convolutional neural networks on FPGAs, in: 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2016, pp. 40–47.
[39] S.I. Venieris, C.-S. Bouganis, Latency-driven design for fpga-based convolutional neural networks, in: 2017 27th International Conference on Field Programmable Logic and Applications (FPL), IEEE, 2017, pp. 1–8.
[40] S.I. Venieris, C.S. Bouganis, fpgaConvNet: mapping regular and irregular convolutional neural networks on FPGAs, IEEE Trans. Neural Netw. Learn. Syst. (2018) 1–17.
[41] J. Wang, Q. Lou, X. Zhang, C. Zhu, Y. Lin, D. Chen, Design flow of accelerating hybrid extremely low bit-width neural network in embedded fpga, in: 2018 28th International Conference on Field Programmable Logic and Applications (FPL), IEEE, 2018, pp. 163–1636.
[42] X. Wei, C.H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, J. Cong, Automated systolic array architecture synthesis for high throughput cnn inference on fpgas, in: Proceedings of the 54th Annual Design Automation Conference 2017, ACM, 2017, p. 29.
[43] X. Wei, Y. Liang, 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), IEEE, 2019, pp. 1–6.
[44] A. Wold, D. Koch, J. Torresen, Component based design using constraint programming for module placement on fpgas, in: 2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), IEEE, 2013, pp. 1–8.
[45] Y. Xiao, D. Park, A. Butt, H. Giesen, Z. Han, R. Ding, N. Magnezi, R. Rubin, A. DeHon, Reducing fpga compile time with separate compilation for fpga building blocks, in: 2019 International Conference on Field-Programmable Technology (ICFPT), IEEE, 2019, pp. 153–161.
[46] Xilinx, Hierarchical design, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug905-vivado-hierarchical-design.pdf, 2017.
[47] Xilinx, Alveo u250 data center accelerator card, https://www.xilinx.com/u250, 2019.
[48] Xilinx, Ultrascale architecture configurable logic block, https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf, 2018.
[49] Y. Xing, S. Liang, L. Sui, X. Jia, J. Qiu, X. Liu, Y. Wang, Y. Shan, Y. Wang, Dnnvm: end-to-end compiler leveraging heterogeneous optimizations on fpga-based cnn accelerators, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. (2019).
[50] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, 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, 2015, pp. 161–170.
[51] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, J. Cong, Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. (2018).
[52] C. Zhu, K. Huang, S. Yang, Z. Zhu, H. Zhang, H. Shen, An efficient hardware accelerator for structured sparse convolutional neural networks on fpgas, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 28 (9) (2020) 1953–1965.

Danielle Tchuinkou Kwadjo obtained a bachelor's and master's in Computer Science at respectively the University of Douala and the University of Yaoundé 1 in Cameroon. She started her Ph.D. at the University of Arkansas and transferred in summer 2019 to the University of Florida. Research Interest: Machine Learning, Computer Architecture, Reconfigurable Computing, Embedded system.

Erman Nghonda Tchinda is pursuing a Ph.D. in the Department of Electrical and Computer Engineering at the University of Florida, under the supervision of Dr. Christophe Bobda. I obtained a bachelor's in computer science at the University of Douala, and a master's at the University of Yaoundé 1 in Cameroon. I started a Ph.D. in Computer Engineering at the University of Arkansas before transferring to University Florida. I am currently working on developing a non-invasive Distributed Embedded Smart Camera system for immersion technologies.

Joel Mandebi is pursuing a Ph.D. in the Department of Electrical and Computer Engineering under the supervision of Dr. Christophe Bobda. I obtained a bachelor and master's in computer science at the University of Yaoundé 1 in Cameroon and started a Ph.D. in Computing Engineering at the University of Arkansas before transferring to Florida in 2019. Research Interest: Cloud Computing, Reconfigurable Computing, Computer Architecture, System-on-Chip, Embedded Systems, Hardware Security, High-performance Computing.

Professor Christophe Bobda received the bachelor's in mathematics from the University of Yaoundé, Cameroon, in 1992, the diploma of computer science and the Ph.D. degree (with honors) in computer science from the University of Paderborn in Germany in 1999 and 2003, followed by a post-doc at the University of Erlangen-Nuremberg. From 2010-2019 he was with the University of Arkansas in 2010 where he was appointed Professor. Since 2019 he is working at the University of Florida. Research interest are: Embedded Vision, Embedded Systems, Reconfigurable Computing, Computer Architecture, Cybersecurity, System-Level Design.