Accelerating Hybrid Quantized Neural Networks on Multi-tenant Cloud FPGA

Danielle Tchuinkou Kwadjo
ECE Department
University of Florida
Gainesville FL, USA
dtchuinkoukwadjo@ufl.edu

Erman Nghonda Tchinda
ECE Department
University of Florida
Gainesville FL, USA
enghonda@ufl.edu

Joel Mandebi Mbongue
ECE Department
University of Florida
Gainesville FL, USA
jmandebimbongue@ufl.edu

Christophe Bobda
ECE Department
University of Florida
Gainesville FL, USA
cbobda@ece.ufl.edu

2022 IEEE 40th International Conference on Computer Design (ICCD) | 978-1-6654-6186-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICCD56317.2022.00079

***Abstract*—The increasing adoption of Field-Programmable Gate Arrays (FPGA) into cloud and data center systems opens the way to the unprecedented acceleration of Machine Learning applications. Convolutional Neural Networks (CNN) have largely been adopted as algorithms for image classification and object detection. As we head towards FPGA multi-tenancy in the cloud, it becomes necessary to investigate architectures and mechanisms for the efficient deployment of CNN into multi-tenant FPGAs cloud Infrastructure. In this work, we propose an FPGA architecture and a design flow that support efficient integration of CNN applications into a cloud infrastructure that exposes multi-tenancy to cloud developers. We prototype the proposed approach on randomly allocated virtual regions to tenants. We study how space-sharing of a single device between multiple cloud tenants influence the design flow, the allocation of resources, and the performance in term of resource utilization and overall latency compared to single-tenant deployments. Prototyping results show a latency at most 8% lower than that of single-tenant deployment while achieving higher resource utilization. We also record a maximum frequency of up to 12% higher in multi-tenant implementations.**

***Index Terms*—FPGAs, Multi-tenancy, CNN Acceleration, Distributed Inference**

I. Introduction

Driven by the increasing demand for performance and efficiency in computation, Field-Programmable Gate Arrays (FPGAs) are increasingly adopted as part of the pool of resources integrated into cloud and data center systems. Cloud providers can now offload compute-intensive algorithms running in the background of the infrastructure unto FPGA devices to achieve lower latency, reduced power consumption, and higher throughput. For example, OVHcloud uses FPGA-based network processing to defend customer workspaces against distributed denial-of-service attacks [1]. In addition, cloud developers can now design custom hardware accelerators without incurring maintenance expenses. For instance, Amazon EC2 F1 instances provide development, debugging, and deployment infrastructure for heterogeneous applications that exploit communication between general-purpose processors and FPGA accelerators [2].

The rising integration of FPGAs in the cloud offers a unique opportunity to accelerate applications in Machine Learning. In recent years, Convolutional Neural Networks (CNN) gained much attention due to their high accuracy and performance in image classification and object detection. However, higher accuracy is typically obtained using deeper and wider CNN architectures that feature a larger number of layers and channels. This dramatic increase in CNN complexity means that advanced FPGAs are needed for efficient CNN inference, typically available in a cloud deployment. It is not surprising to see an increased deployment of CNN accelerators on cloud FPGAs to accelerate computer vision pipelines [[3], [4]]. Current cloud infrastructures provision single-tenant FPGAs that are entirely allocated to a single user at a time [amazon, baidu, etc.]. However, FPGA multi-tenancy is a rising trend among researchers [5]–[7]. Cloud architectures that expose multi-tenant FPGAs to developers allow running multiple hardware workloads concurrently on a single device independently of whether the hardware accelerators belong to different users.

This work investigates the design and inference of CNNs on multi-tenant cloud FPGAs. Since the FPGA is space-shared between concurrent hardware accelerators, a cloud developer shares FPGA resources with the co-tenant and the shell that implements controls from the cloud provider. Therefore, in the context of multi-tenant cloud FPGAs, we propose a design flow and an architecture that improves hardware utilization and productivity, to ensure minimal latency increase for CNNs inference. We use the FINN framework [8] as our baseline and extend it to support the pre-implemented flow, which is a divide-and-conquer approach that enables application and domain-specific optimization on the design of CNN architectures. Our proposed framework provides an efficient streaming implementation for multi-tenant FPGAs by benefiting from the customizability of FINN. Specifically, the contribution of this paper include:

- Defining the constraints of cloud deployments that expose multi-tenant FPGAs to developers.
- Propose an FPGA architecture as part of the shell to support co-hosting hardware accelerators on a single cloud FPGA.
- Discuss the design flow that relies on graph partitioning to achieve efficient acceleration of CNN inference without tedious HDL programming and verifications, while improving the Quality Of Result (QoR) compared to the traditional design flow with Vivado.

The rest of the paper is organized as follows: section II

2576-6996/22/$31.00 ©2022 IEEE
DOI 10.1109/ICCD56317.2022.00079

presents some background information discussing the acceleration of CNN on single and multi-FPGA platforms. Then, section III elaborates on the different steps that enable the deployment of CNN inference in multi-tenancy. Afterwards, experimental results are presented in section IV and section V concludes the paper.

II. Background

Accelerators with the streaming architecture always tailor the hardware with respect to the target network [9]. The topology of such CNN accelerators is transformed into a layer-by-layer execution schedule, following the structure of the DAG [10].

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 [11], [12]. On the downside, this accelerator architecture cannot scale to arbitrarily large CNNs. 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. Most of the research works in cloud FPGA platforms revolve around multi-FPGA cloud infrastructures. Shan et al. [13] proposed an optimized power flow to map CNNs on multi-FPGA configuration bitstreams that satisfy different application requirements. The platform consists of a host CPU that controls eight FPGAs over a PCI-express (PCIe) bus. It can quickly reconfigure them with several configurations generated offline, adapting them to the actual application performance requirements. However, they only consider CNNs applications that can be modeled as multi-kernel task-level pipelines. Cloud-DNN [3] proposes a Framework for Mapping DNN Models to Cloud FPGAs by partitioning the DNN into three sub-nets. The sub-nets are mapped to different dies in an SSI-based FPGA.

Several works [4], [12], [14], [15] in the literature employ FINN to generate NN accelerators on FPGAs. Nevertheless, FINN accelerators' area consumption and parallelism parameters cannot be arbitrarily deduced. Since the performance of an accelerator is bounded by the slowest component within the design, finding the parameters to generate a balanced design can be a bottleneck. In this work, we propose an accurate model to find the optimal parameters for the configuration to assess the resource consumption and timing for FINN accelerators. We also propose a pre-implemented flow to compose the final accelerator considering the platform restriction.

A. FINN Architecture

FINN is a framework from Xilinx Research Lab, enabling the design of heterogeneous custom streaming architecture for a given topology. Separate compute engines are dedicated to each layer, communicating via on-chip data streams. Each engine starts to compute as soon as the previous engine produces output. It currently supports fully connected, convolutional, ReLU, and pooling layers.

The computational core of the compute engines is the matrix-vector unit (MVU), as the vast majority of computing operations in neural networks can be expressed as matrix-vector operations. The sliding window unit (SWU) supplies the convolution engine with the image matrix from the incoming feature map by applying interleaving and implementing the im2col algorithm. An MVU computes the matrix-matrix product using a different column vector from the image matrix stream. The MVU consists of an input and output buffer and an array of Processing Elements (PEs), each with a number of SIMD lanes. The number of PEs (P) and SIMD lanes (S) is configurable to regulate the throughput. A PE performs a number of parallel multiplications equal to the SIMD value. It then reduces them in an adder tree for their subsequent accumulation towards the computed dot product. Finally, threshold comparisons derive the output values from the accumulation results.

III. Proposed Framework

This section discusses the different steps to generate a CNN accelerator, the constraints that need to be implemented to maximize the performance, and a design flow to generate the architecture underneath. The proposed framework is depicted in Figure 1. Table I summarizes the notations used in the problem formulation.

TABLE I
Notations

Name	Description
$G = (V, E, \omega, \phi)$	Graph G with a set of Vertices V, edge set E, vertices weights ω, edges weight ϕ
i, N	Index of a vertice, $\|V\|$
LUT_i	LUT capacity of the VR_i.
FF_i	Flip-flop requirement of the VR_i.
$BRAM_i$	BRAM capacity of the VR_i.
DSP_i	DSP capacity of the VR_i.
IFM_{DIM_i}	Dimension of the Input feature maps.
K_i	kernel size
IFM_{CH_i}	Number of channels of the input layer.
OFM_{CH_i}	Number of channels of the output layer.

A. Multi-Tenant FPGA platform

The FPGA fabric is divided into disjoint Virtual Regions (VRs) purposed to host Virtual machine workloads, enabling fast IO access to VR registers (Figure 1-(2)). Each FPGA of the platform consists of a shell layer, which is a set of static components on the FPGA that cloud users cannot modify (Figure 4). The shell is made of two major components: **(1) IO Controllers:** to manage the communication with off-chip resources such as memory, CPU, etc. In this work, we do not elaborate on the interfacing logic of the shell as we rely on vendor IPs to design high-performance IO controllers. **(2) On-chip Interconnect:** it implements a soft-NoC topology[1] that enables efficient on-chip communication between VRs. We do not discuss the internal architecture of the shell.

[1]The NoC reaches a near spec maximum frequency of 872 MHz and a bandwidth of 28Gbps

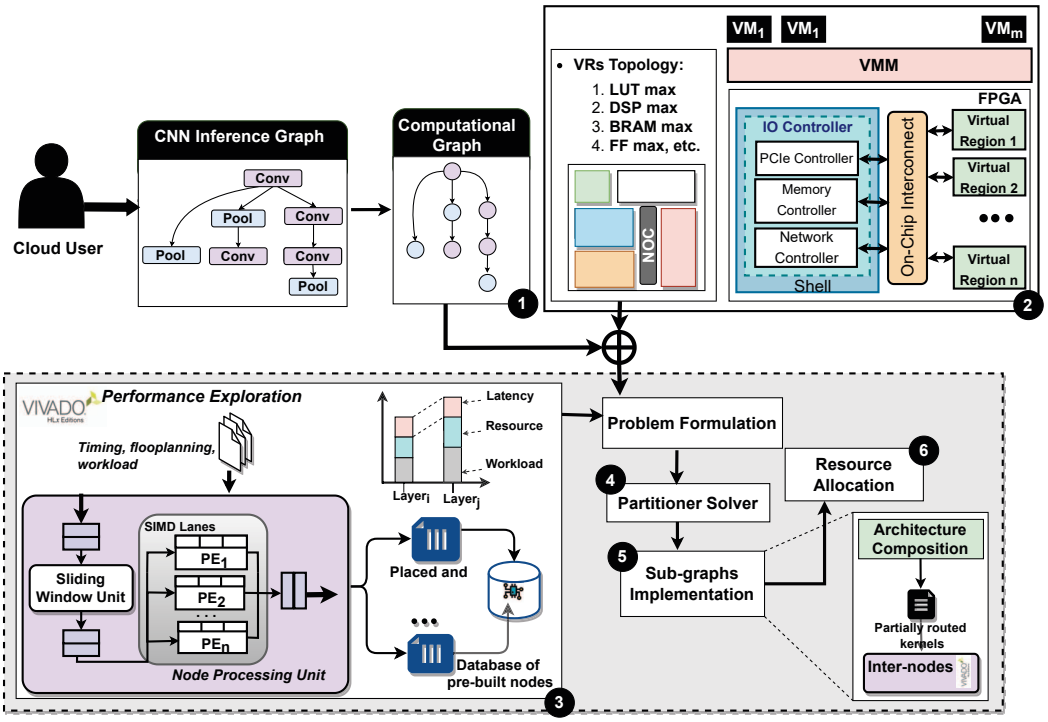

Fig. 1. Framework Overview

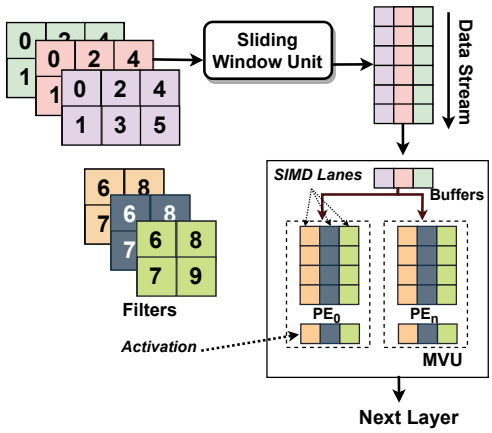

Fig. 2. FINN architecture. SWU interleaves the input by applying the image-to-column algorithm and feeds MVTU.

B. Framework Overview

The deployment of an application in a multi-tenant cloud infrastructure is depicted in Figure 1 as follow:

(1) **Computational Graph**: First, it takes as input an inference model trained with Tensorflow or ONNX Deep learning framework. Then, it generates the computational graph: $G = (V, E, \omega, \phi)$ with a set of Vertices V, edges set E, vertices weights ω, edges weight ϕ. The vertices weight represents the computational workload of each layer, and the edge weight is the local memory ratio, which is the amount of data (in Kb) that is moved between two nodes.

(2) **Platform Description**: Given the physical layout of FPGA chips (array of logic components and interconnect), each "FPGA unit of virtualization" will represent a designated area on the device that we call "virtual region" or VR. The VRs are then advertised in the cloud as opposed to entire FPGAs. To support resource elasticity, the VR is interfaced to an NoC that establishes on-chip communication between VRs in a user domain. The FPGA is accessed through a set of *"IO Controllers"*. In this work, we only use a Peripheral Component Interconnect Express (PCIe) connection, but the architecture can also accommodate network interfaces. To deploy an accelerator within the proposed platform, each request is associated with the VRs topology description, including the resources allocated to each VR and their interconnect in the form of a dataflow graph as presented in section III-F.

(3) **Performance Exploration**: Given the platform description resources and the inference graph, the framework explore the parameters that will minimize the latency given the resources budget of the VRs. Additionally, developing high-performance hardware accelerators on FPGA often demands skills in hardware design and long development cycles. Besides, the depth of CNN architectures increases by reusing and replicating several layers. We take advantage of the replication of CNN layers to improve design performance and productivity by individually pre-implementing (Synthesis, placement, and routing) CNN's components. Furthermore, the pre-implemented designs can be reused in adjacent layers, improving the engineering time. We employ the FINN-

HLS [8] framework to design accelerators (Figure 2).

(4) With the implementations and performance details (timing, floorplanning, workload), we define several constraints for the solver to partition the computational graph G into a set of sub-graphs $G = (G_1, G_2, ..., G_M)$.

(5) Sub-graphs architectures are generated by stitching the corresponding pre-built components through a fully automated process.

(6) Finally, the sub-graphs are allocated to the VR.

C. Performance Exploration

This section essentially consists in performing a design space exploration of the performances achievable by CNN sub-functions such as Convolution, pooling, and fully connected layers (FC) under the FINN architecture. It takes into consideration some design constraints, such as the FPGA's resources and timing. If the design space exploration results in satisfactory performance, the produced netlists are saved into a database as Design Checkpoint (DCPs).

*1) **Problem Definition**:* We use the following notation to describe a convolution. For each layer i in a given CNN, there are IFM_{DIM_i} IFMs, K_i kernel size, $(IFM_{CH_i}\ and\ OFM_{CH_i})$ are the number of channels of the input and output layer. For FC, a layer i can be represented by the height H_i, which is the number of neurons of the layer, and W_i, which is the number of synapses per neuron.

To highlight the effect of the folding on latency, let us consider the results presented in Figure 3. A higher level of parallelism implies a higher number of resources used. Each layer has a set of parameters (S, P) that control the degree of parallelism, which must be chosen so that the final accelerator results in a balanced streaming pipeline, with resources fitting within the given budget. Finding the right configuration can greatly impact the final results. Previous work has demonstrated that extensive automated search in the design space can identify accelerator configurations better than human designers. Regarding heterogeneous streaming architecture, the slowest layer will determine the overall throughput. The guiding principle is to implement rate-balancing [8] between the layers. So, each layer should use roughly an equal number of clock cycles (CC) to process an image.

*a) **Latency Constraints**:* For an inference model with N nodes and a platform with M VRs, we seek to maximize $\{(S_i, P_i)\ \forall i = 1, ..., N\}$ such that:

$$\begin{aligned} throughput &= \frac{\#batch}{max(Latency_1, Latency_2, ..., Latency_N)} \\ CC_i &= (1+\epsilon) \times CC_{i+1} \quad \forall i = 1, ..., N \\ with\ CC_i &= \frac{OFM_{H_i} \times OFM_{W_i} \times K_i^2 \times IFM_{Ch_i} \times OFM_{Ch_i}}{S_i \times P_i} \end{aligned} \tag{1}$$

Assuming:

$$\begin{cases} OFM_{Ch_i} == IFM_{Ch_{i+1}} \\ \alpha_i = OFM_{Dim_i}^2 \times K_i^2 \times IFM_{Ch_i} \\ \beta_{i+1} = OFM_{Dim_{i+1}}^2 \times K_{i+1}^2 \times OFM_{Ch_{i+1}} \end{cases}$$

Equation 1 can be reduced to:

$$\alpha_i \times S_{i+1} \times P_{i+1} = (1+\epsilon) \times \beta_{i+1} \times S_i \times P_i$$

ϵ is the imbalance factor, allowing a margin between different layers.

*b) **Variables Constraints**:* For a layer i, we fix the maximum value of P_i and S_i to 64. As observed in [8], a $P_i > 64$ results in low BRAM usage, forcing Vivado HLS to implement the weight and threshold memories using LUTs, which causes the LUTs per operation to increase, resulting in low resource efficiency. We denote by $\sigma_{i,p}$, a binary decision variable such that $\sigma_{i,p} = 1$ iff $P_i = x_{i,p}$, with $\forall x_p = 1, ..., 64$.

$$P_i = \sum_{p=1}^{64} \sigma_{i,p} \times x_{i,p} \quad and \quad \sum_{p=1}^{64} \sigma_{i,p} = 1$$

$$S_i = \sum_{p=1}^{64} \gamma_{i,p} \times y_{i,p} \quad and \quad \sum_{p=1}^{64} \gamma_{i,p} = 1$$

$$\epsilon_i = \sum_{p=1}^{2\times\epsilon*100} \delta_{i,p} \times z_{i,p} \quad and \quad \sum_{p=1}^{2\times\epsilon*100} \delta i, p = 1$$

$z_{i,p}$ is the set of relaxing values. For example, if $\epsilon = 0.25\ ns$, then a maximum difference latency of $+/-0.25\ ns$ is permitted between the latency of the layers. Hence $z_{i,p} \in [-\epsilon, \epsilon]$. With a maximum of two decimal numbers per relaxing factor, the search range is equals to $2 \times \epsilon * 100$.

*2) **Resources Constraints**:* The framework has to quickly estimate an accelerator's LUT, DSP, and OCM requirements from a given set of values of the parallelism variables (P and S). Design congestion can negatively impact the achievable frequency for any FPGA design. Hence, it is recommended to balance resource utilization between layers. A balance resource utilization should not exceed the maximum utilization of 70 % LUTs, 50 % FF, and 80% DSPs Block of total available resources. We express as $F_{t_i}(P_i, S_i)$, a linear function that estimates the the amount of resources of type t demanded by the ith layer for a given (P_i, S_i) configuration.

$$\begin{cases} \sum_{i=1}^{N} F_{lut_i}(P_i, S_i) \leq LUT_{VRs}, \quad \forall i = 1, ..., M \\ \sum_{i=1}^{N} F_{dsp_i}(P_i, S_i) \leq DSP_{VRs}, \quad \forall i = 1, ..., M \\ \sum_{i=1}^{N} F_{bram_i}(P_i, S_i) \leq BRAM_{VRs}, \quad \forall i = 1, ..., M \end{cases}$$

The values of the $F_{t_i}(P_i, S_i)$ are computed using the layer cost model as in [16]. The optimization problem is expressed as a Mixed Integer Quadratic Program (MIQP).

D. Graph Partitioning

The role of the partitioner is to segment the computational graph into sub-graphs and assign those to VRs. As the sub-functions have been configured to fit the resource budget of VRs, we only focus on having the minimum number of partitions. There can be two graph partitioning scenarios: (1) A CNN accelerator can fit in a single VR; In this case, no graph partitioning is required. (2) A CNN accelerator requires more

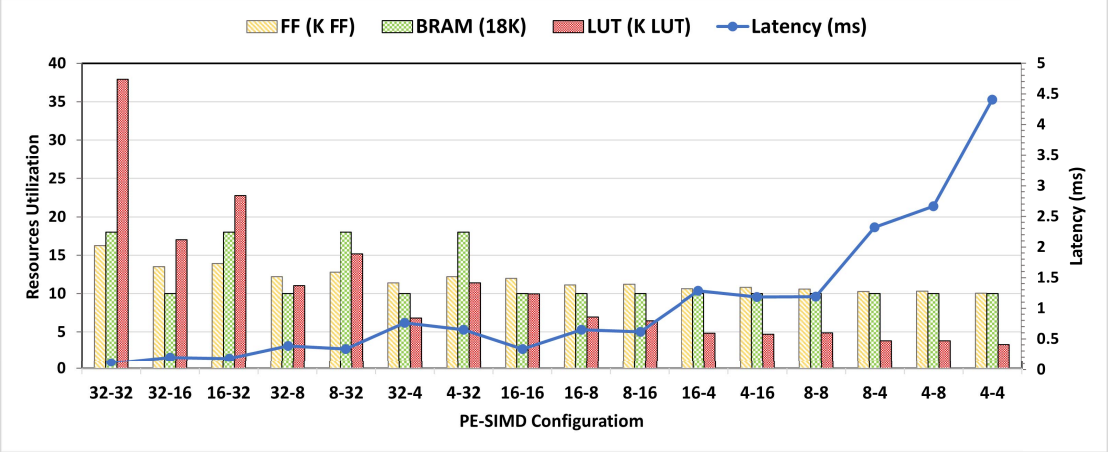

Fig. 3. Folding Factor Design Space Exploration

than one VR. In that case, we proceed with multi-way graph partitioning, which consists of finding a k-balanced partitions of a graph $G = (V, E, \omega, \phi)$ that minimizes objective function over the cut nets for some value of ϵ. In this work, the FPGA resources we consider are the number LUTs, BRAMs, and DSPs.

a) ***Multi-level partitioning****:* we implement a recursive balanced bi-partitioning to generate the different partitions of the computational graph. More precisely, whenever the partition P_i does not violate the constraints: **(1)** the partitions does not satisfy the VRs requirement in terms of resources, **(2)** The number of partitions is smaller than the number of VRs. We recursively bi-partition each block of P_i until we have k blocks in total. In Algorithm 1, this process is implemented in Lines 2–5. If one of the conditions mentioned is violated, we proceed to the refinement step. The weight of the heaviest partition i is restricted by a fixed upper bound $U = \epsilon \times \frac{\omega(V)}{k}$, with ϵ represents the unbalanced factor, since all partition cannot have exactly the same weight, and $k \leq \#VRs$.

b) ***Refinement step****:* : For n iteration, a bi-partitioning will produce 2^n partitions, resulting in unbalanced partitions, or too many partitions. The refinement step allows us to merge smaller partitions or further split heavier partitions (with $k \leq \#VRs$) to accommodate VRs resources.

Algorithm 1: Automated graph partitioning algorithm

Input : Graph $G = (V, E, \omega, \phi)$, k, $\epsilon > 0$
Output: k-balanced Partitions

1 **Function** `partition` (G, k, ϵ) **:**
2 **if** $(k \leq \#VRs)$ *and* $(RES_{P_i} >= RES_{VRs_j}, \forall i \in k, j \in \#VRs)$ **then**
3 $G_i = bi\_partition(G, k, \epsilon)$;
4 $II_i := partition(G_i, k, \epsilon)$;
5 **else**
6 $II_p = V$
7 **end**
8 $II = refine(G, balance(G, \{II_p\}))$;
9 **Return** $\{II\}$

E. Sub-graphs Implementation

Its function is to generate accelerators for the different partitions. We start by synthesizing the CNN components OOC. The OOC flow ensures that I/O buffers and global clock 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. The component's granularity is discussed in section IV-B. To achieve high QoR, the implementation of components follows the following design considerations: **(1) Floorplanning:** Pblock boundaries allow you to leverage clock region or SLICEs boundaries to determine the size of the pblock. This can help limit clock skew and help with the overall clock placement of the design. It also help minimizing the resources utilization, instead of 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, the smaller the area of a pblock is, the more the component can be relocated across the chip, which increases the reusability. **(2) Strategic port planning:** the placement of the ports when pre-implementing modules is one of the most important steps to ensure high performance and productivity improvement. 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. **(3) 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. textbf(4) 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. It decreases compilation times and improves productivity. **(5) Checkpoint file generation:** the pre-implemented components are stored on disk in the form of DCPs.

The next step is combining the designs of the pre-built components into a sub-graphs architecture as defined after the graph partitioning phase. We employ a custom API designed with the RapidWright [17], to compose the sub-graph hardware accelerators. The final design still has the logic and the internal routing locked, and the nets created to connect sub-graph components are not routed yet. 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 to minimize timing delays.

F. Resources Allocation

From the cloud provider's perspective, the FPGA resources are represented as a dataflow graph (DFG) in which each node represents a VR, and the edges denote the communication latency between the VRs regardless of the physical FPGA from which they are provisioned. Assuming the number of sub-graphs $k \leq \#VRs$, we seek to decrease the communication latency between designs implemented in different VRs, but belonging to the same VM. The optimization problem can be presented as a set of equations in the form of a Mixed Integer Quadratic Program (MIQP) and is expressed in Equation 2

$$Min \sum_{i=1}^{n} \sum_{q=1}^{n} \sum_{j=1}^{m} \sum_{k=1}^{m} c_{jk} \times link_{iq} \times x_{ij} \times x_{qk} \qquad (2)$$

With x_{ij} binary decision variable that represents whether VR_j is assigned to the $accelerator_i$ ($x_{ij} = 1$) or not ($x_{ij} = 0$), c_{jk} communication latency from VR_j to VR_k, $link_{iq}$ binary constant defining whether data flows from $accelerator_i$ to $accelerator_q$.

IV. Experimental Results

A. Evaluation Platform and Setup

For evaluation purposes, we split the FPGA into Three different VRs of different sizes. We rely on the architecture defined in [21]. Figure 4 shows the FPGA layout with the resource utilization of each dedicated area. The user designs can only be hosted in the VRs. We use Vendor IPs to implement the PCIe interface in the "*PCI Block*". The "*UNASSIGNED*" region is used to place and route the soft-NoC. Designs are implemented on a Xilinx Kintex UltraScale+ FPGA (xcku5p). The hardware is generated using Vivado v2021.1 and RapidWright v2020.1, and the components are implemented with Vitis HLS.

The hardware generation is conducted on a computer equipped with an Intel Corei7-9700K CPU@3.60GHz×4 processor and 32GB of RAM. The performance exploration stage is solved with LocalSolver [22] as it has demonstrated

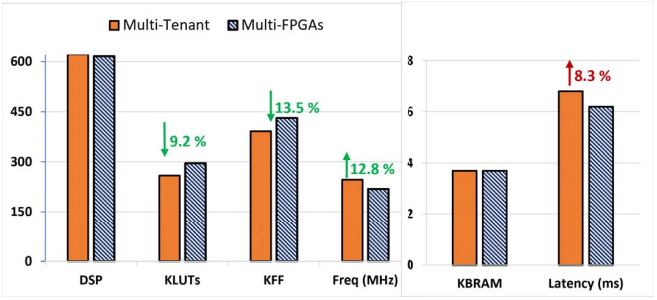

Fig. 5. Performance Comparison of the multi-tenant vs the multi-FPGA implementation

obtaining efficient results (optimality gap $< 10\%$) within seconds regardless of the size of the problem when compared to other Mixed-Integer Programming (MIP) solvers on NP-hard problems such as the quadratic assignment problem.

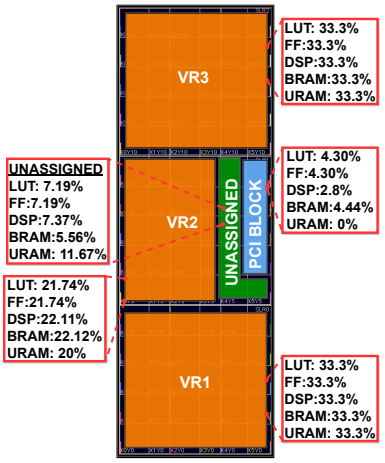

Fig. 4. FPGA Layout

Experiments are conducted on ResNet-50 to verify our algorithm's correctness and performance. Prior works [23] show that the last layer is highly-sensitive to low-precision quantization. Therefore, that layer's weights and activations are quantized to an 8-bit and 16-bit fixed-point, respectively, to minimize the loss in accuracy. In other layers (including the first layer), the weights and output activations are quantized to 1 bit and 4 bits, respectively. The component granularity is defined by the block topology of ResNet as illustrated in Figure 6.

B. Performance

In this section, we aim to compare the performance of the ResNet baseline on a multi-FPGAs platform to a multi-tenant. The baseline implementation of ResNet requires the allocation of two xcku5p FPGAS. To generate the corresponding design, we manually partition the graph and assign the resulting sub-graphs to two FPGAs. For performance comparison, we create four instances of the same FPGA layout (Figure 4) and randomly allocate VRs to a single tenant such that the total amount of resources is close to the number of resources of 2 FPGAs. The performance and the resources are depicted in Figure 5. Both platforms use approximately the same number of DSP and BRAMs, with resp. 9.2% and 13.5 % less LUTs and FFs. We also note a 12.8% higher frequency and an 8.3% higher latency. The higher latency is justified by the delay

TABLE II
RESNET PERFORMANCE COMPARISON WITH STATE-OF-ART APPROACHES

	Ma et al. [10]	Cloud-DNN [3]	Biookaghazadeh et al. [18]	Elastic-DF [4]	Zhang et al. [19]	CNN-on-AWS [20]	Our Approach
FPGA/Platform	Intel Arria 10	AWS	Intel Arria 10	2*U250	4* Virtex Ultrascale	5*AWS F1	2*xcku5p
Platform	Single FPGA			Multiple FPGAs			Mult-tenant FPGAs
Model	ResNet-50	ResNet-50	ResNet-50	ResNet-50	ResNet-152	ResNet-18	ResNet-50
FMax	200 MHz	125 MHz	212 MHz	217 MHz	150 MHx		246 MHz
(Precision (fixed) w=weight, a=activation)	w16a16	w16a16	w8a8	w1a2	w16a16	w16a16	w1a4
DSP Blocks	1,518 (100%)	80.25%	33%	-	(19%, 18.8%, 9.3%, 30.3%)	-	((2.96%, 5.12% 1.05%, 3.82%)
LUTs	218.6K (51%)	64%	34%	-	(77.4%, 74.5%, 83.4%, 88.6%)	-	(3.57%, 6.36% 4.69%, 27.3%)
BRAM (M20K)	1,927 (71%)	83%	48%	-	(81.4%, 81.3%, 82.1%, 86%)	-	(41.2%, 50.6%, 23.42%, 60.3%)
Latency/Image (ms)	12.51	13.9	20.9	2.3	-	2.1	6.8

TABLE III
COMPARISON OF COMMUNICATION PERFORMANCES TO THE BASELINE

# of VR	Best scenario		Worst scenario	
	Comm. time (ns)	Comparison to host baseline	Comm. time (ns)	Comparison to host baseline
3	1.678	× 6987↑	11.024	× 907 ↑
10	9.59	× 1046 ↑	99.736	× 100 ↑
20	22.04	× 453 ↑	246.848	× 40 ↑

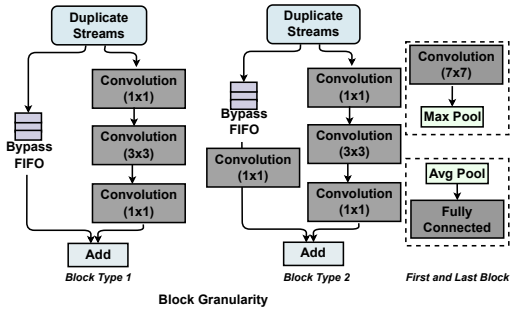

Fig. 6. Different modules granularity

added by the NoC to move the data between VRs. ResNet built by pre-implementing components uses 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 pblocks, limiting 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, making it difficult to capture all its specificities.

Table II compares our work with results from prior work on FPGA inference. Works are grouped into single and multi-FPGA implementations. Among the single-FPGA, Ma et al. [10] report the smallest latency of 12.51 ms by integrating optimized RTL components within an automated CNN compiler for various inference tasks. Elastic-DF is the closest work to ours regarding multi-FPGA implementations and achieves a latency of 2.1 ms. However, they employ a data quantization of w1a2, resulting in an accuracy drop of 67.3%, while w1a4 presents a better performance and memory cost-accuracy (78.1%) trade-off. Furthermore, FINN uses hls::stream for data transfer. Since the data width is limited to 4096, with a w1a4 quantization, the data width exceeds the 4096 from the third block, yielding to use of a *Stream Data Width Converter*, to upscale or downscale a stream, with a slight additional latency.

*a) **Benefits of On-chip Communication**:* one may question the necessity of implementing a soft-NoC to support multi-tenancy. Designing FPGA accelerators is a time-consuming, depending on the design's complexity to implement. Considering a context in which a user has already programmed specific functions in a cloud FPGA, leveraging the deployed accelerators instead of redesigning an entire hardware stack is beneficial in terms of productivity. It could also be more cost-effective as a hardware accelerator could be reused by several hardware workloads. Another advantage is the low communication overhead compared to relying on software functions to initiate data movement between hardware accelerators. Considering that host round trips to the FPGA could take an average $\sim 10\mu$s, we compare hardware-level data copy of 32 bits with that of software. We scale the number of virtual regions on an FPGA from 2 to 20 and record the time required for transferring a packet between the two most distant VRs in terms of routing hops. We consider two operating conditions: **(1) The best scenario:** there is no congestion. **(2) The worst scenario:** The on-chip interconnect is congested. Each router on the way is overloaded, which introduces routing delays. Table III summarizes the experimental observations. In the best scenario and with the FPGA divided into 20 regions, transferring a packet takes 22.04 ns, which is 453× faster than an equivalent operation by the host. In the worst scenario, with the same quantity of VRs in an FPGA, the communication uses $\sim 2\mu$s, which is 40× faster than an equivalent operation by the host. Overall, this teaches two major lessons: **(1) implementing on-chip communication** support between the VRs drastically improves the throughput compared to letting a VM or the host copy the data between the accelerators on a chip. **(2) Achieving higher throughput** is tightly associated with decreasing the number of regions provisioned on a single FPGA.

TABLE IV
DESIGN GENERATION TIME FOR IMPLEMENTATION OF RESNET WITH VIVADO AND THE PROPOSED FRAMEWORK IN MINUTES

	Multi-Tenant ResNet			Baseline ResNet	
Tasks	Perfor. Exploration	Graph Partitioning	Component Implem	Synthesis	P&R
Time	12.6 sec	3.6 sec	4.63 h	3.2 h	6.9h
Ratio	∼0%	∼0%	∼99.9%	31.6%	68.31%
Total (hours)	4.63 h ↑ (2.18 ×)			10.1h	

C. Productivity

With the continuous growth of CNNs parameters and depth, improving productivity is an important factor in hardware design. This section shows how the proposed flow can leverage component reuse to reduce compile-time and implementation cycles. Table IV presents the time in hours to generate the design checkpoint with both rapidwright and vivado. ResNet topology reuse 72% of its layers. The proposed framework takes advantage of that properties to achieve a $2.18\times$ productivity.

V. CONCLUSION

This paper proposes a framework to accelerate model inference on a multi-tenant FPGA Cloud Platform. The cloud architecture provides an FPGA abstraction to the users, which consists in dividing the FPGA into "Virtual Regions." The architecture also features a shell layer that enables fast access to FPGA resources and inter-VRs communication. The framework takes the computational graph of the CNN model inference as input. Then, it performs an intensive search in the form of a quadratic optimization problem to determine each layer's highest degree of parallelism considering the platform constraints. The graph is then partitioned, and the resulting sub-graphs are allocated to the VRs such that the communication latency is minimized. Experiments and results show that our approach improves 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. In future works, we intend to expand to a wider variety of neural networks and report power and energy consumption.

ACKNOWLEDGEMENT

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

REFERENCES

[1] Bittware, "How ovhcloud uses fpgas to mitigate ddos attacks," Nov 2021 [Online] https://www.bittware.com/resources/case-study-ovh/, 2020.

[2] D. Pellerin, "Amazon ec2 f1 instances," Nov 2021 [Online] https://aws.amazon.com/ec2/instance-types/f1/, 2016.

[3] Y. Chen, J. He, X. Zhang, C. Hao, and 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.

[4] T. Alonso, L. Petrica, M. Ruiz, J. Petri-Koenig, Y. Umuroglu, I. Stamelos, E. Koromilas, M. Blott, and K. Vissers, "Elastic-df: Scaling performance of dnn inference in fpga clouds through automatic partitioning," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 2, pp. 1–34, 2021.

[5] N. Tarafdar, T. Lin, D. Ly-Ma, D. Rozhko, A. Leon-Garcia, and P. Chow, "Building the infrastructure for deploying fpgas in the cloud," in *Hardware Accelerators in Data Centers*. Springer, 2019, pp. 9–33.

[6] G. Dai, Y. Shan, F. Chen, Y. Wang, K. Wang, and H. Yang, "Online scheduling for fpga computation in the cloud," in *2014 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2014, pp. 330–333.

[7] K. Zhang, Y. Chang, M. Chen, Y. Bao, and Z. Xu, "Computer organization and design course with fpga cloud," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. ACM, 2019, pp. 927–933.

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

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

[10] 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)*. IEEE, 2017, pp. 1–8.

[11] 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)*. IEEE, 2017, pp. 535–547.

[12] L. Petrica, T. Alonso, M. Kroes, N. Fraser, S. Cotofana, and 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.

[13] J. Shan, M. T. Lazarescu, J. Cortadella, L. Lavagno, and M. R. Casu, "Power-optimal mapping of cnn applications to cloud-based multi-fpga platforms," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 12, pp. 3073–3077, 2020.

[14] A. Khodamoradi, K. Denolf, and R. Kastner, "S2n2: A fpga accelerator for streaming spiking neural networks," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 194–205.

[15] F. K. Mohammad Ghasemzadeh, Mohammad Samragh, "Rebnet: Residual binarized neural network," in *Proceedings of the 26th IEEE International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '18, 2018.

[16] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser, and K. Vissers, "Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, pp. 1–23, 2018.

[17] 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)*. IEEE, 2018, pp. 133–140.

[18] S. Biookaghazadeh, P. K. Ravi, and M. Zhao, "Toward multi-fpga acceleration of the neural networks," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 17, no. 2, pp. 1–23, 2021.

[19] W. Zhang, J. Zhang, M. Shen, G. Luo, and N. Xiao, "An efficient mapping approach to large-scale dnns on multi-fpga architectures," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 1241–1244.

[20] J. Shan, M. T. Lazarescu, J. Cortadella, L. Lavagno, and M. R. Casu, "Cnn-on-aws: Efficient allocation of multikernel applications on multi-fpga platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 2, pp. 301–314, 2020.

[21] J. M. Mbongue, D. T. Kwadjo, A. Shuping, and C. Bobda, "Deploying multi-tenant fpgas within linux-based cloud infrastructure," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 2, pp. 1–31, 2021.

[22] T. Benoist, B. Estellon, F. Gardi, R. Megel, and K. Nouioua, "Localsolver 1. x: a black-box local-search solver for 0-1 programming," *4or*, vol. 9, no. 3, p. 299, 2011.

[23] H. Nakahara, H. Yonekawa, T. Fujii, and S. Sato, "A lightweight yolov2: A binarized cnn with a parallel support vector regression for an fpga," in *Proceedings of the 2018 ACM/SIGDA International Symposium on field-programmable gate arrays*, 2018, pp. 31–40.