# A Unified FPGA Virtualization Framework for General-Purpose Deep Neural Networks in the Cloud

SHULIN ZENG, GUOHAO DAI, HANBO SUN, JUN LIU, SHIYAO LI, GUANGJUN GE, KAI ZHONG, KAIYUAN GUO, YU WANG, and HUAZHONG YANG, Tsinghua University

INFerence-as-a-Service (INFaaS) has become a primary workload in the cloud. However, existing FPGA-based Deep Neural Network (DNN) accelerators are mainly optimized for the fastest speed of a single task, while the multi-tenancy of INFaaS has not been explored yet. As the demand for INFaaS keeps growing, simply increasing the number of FPGA-based DNN accelerators is not cost-effective, while merely sharing these single-task optimized DNN accelerators in a time-division multiplexing way could lead to poor isolation and high-performance loss for INFaaS. On the other hand, current cloud-based DNN accelerators have excessive compilation overhead, especially when scaling out to multi-FPGA systems for multi-tenant sharing, leading to unacceptable compilation costs for both offline deployment and online reconfiguration. Therefore, it is far from providing efficient and flexible FPGA virtualization for public and private cloud scenarios.

Aiming to solve these problems, we propose a unified virtualization framework for general-purpose deep neural networks in the cloud, enabling multi-tenant sharing for both the Convolution Neural Network (CNN), and the Recurrent Neural Network (RNN) accelerators on a single FPGA. The isolation is enabled by introducing a two-level instruction dispatch module and a multi-core based hardware resources pool. Such designs provide isolated and runtime-programmable hardware resources, which further leads to performance isolation for multi-tenant sharing. On the other hand, to overcome the heavy re-compilation overheads, a tiling-based instruction frame package design and a two-stage static-dynamic compilation, are proposed. Only the lightweight runtime information is re-compiled with ~1 ms overhead, thus guaranteeing the private cloud's performance. Finally, the extensive experimental results show that the proposed virtualized solutions achieve up to 3.12× and 6.18× higher throughput in the private cloud compared with the static CNN and RNN baseline designs, respectively.

CCS Concepts: • **Computer systems organization → Reconfigurable computing**; **Neural networks**; **Cloud computing;**

Additional Key Words and Phrases: Virtualization, neural networks, cloud computing, FPGA



This work was supported by National Natural Science Foundation of China (U19B2019, 61832007, 61621091), National Key R&D Program of China (No. 2018YFB0105005, No. 2017YFA02077600), and China Postdoctoral Science Foundation (No. 2019M660641). This work was also supported by Beijing Innovation Center for Future Chips, Tsinghua EE Xilinx AI Research Fund, Beijing National Research Center for Information Science and Technology (BNRist).

Authors' address: S. Zeng, G. Dai, H. Sun, J. Liu, S. Li, G. Ge, K. Zhong, K. Guo, Y. Wang, and H. Yang, Tsinghua University, 30 Shuangqing Road, Haidian District, Beijing 100086, China; emails: {zengsl18, daiguohao, sun-hb17, liu-j20, lishiyao20}@mails.tsinghua.edu.cn, aaoceanaa@tsinghua.edu.cn, {zhongk19, gky15}@mails.tsinghua.edu.cn, yu-wang@tsinghua.edu.cn, yanghz@mail.tsinghua.edu.cn.



1936-7406/2021/12-ART24 $15.00
https://doi.org/10.1145/3480170

**ACM Reference format:**
Shulin Zeng, Guohao Dai, Hanbo Sun, Jun Liu, Shiyao Li, Guangjun Ge, Kai Zhong, Kaiyuan Guo, Yu Wang, and Huazhong Yang. 2021. A Unified FPGA Virtualization Framework for General-Purpose Deep Neural Networks in the Cloud. *ACM Trans. Reconfigurable Technol. Syst.* 15, 3, Article 24 (December 2021), 31 pages.
https://doi.org/10.1145/3480170

## 1 INTRODUCTION

**Deep Neural Network** (**DNN**) inference tasks take up the majority of the total deep learning workloads in the cloud, where INFaaS has become a primary enabler for the success of **Artificial Intelligence** (**AI**) in the data center. According to the report of Facebook, data analysis demands based on inference tasks is doubling each year in the data center [35]. Moreover, in the data center of Amazon, inference tasks make up almost 90% of the total deep learning tasks [24]. In recent years, due to the advantages of programmability, high performance, and energy efficiency, many cloud vendors have provided cloud services using FPGAs, such as Amazon [4], Alibaba [3], and Microsoft [15], for example. FPGA accelerators can provide high energy efficiency and performance solutions for DNN inference tasks [17, 20]. However, most FPGA based deep learning accelerators are optimized for single-task and static-workload scenarios [6, 11, 16, 42], that are hard to meet the requirement in the cloud.

In order to meet the growing demand for INFaaS, one intuitive method is to increase the number of FPGA-based DNN accelerators. However, this approach is not cost-effective for cloud vendors. Thus, multi-tenancy, i.e., multiple users sharing a single FPGA, is a necessary factor to promote the success of FPGA virtualization for deep learning inference applications in the cloud. However, current FPGA-based DNN accelerators are designed to achieve the highest inference speed. For instance, Microsoft Brainwave [16], which is specifically designed for INFaaS in the cloud, focuses on optimizing the latency of running a single DNN inference task. In addition, the DNN benchmark suite MLPerf [29] still utilizes the single DNN model for the server scenario. In fact, several complex DNN inference applications can benefit from the multi-tenancy deployment of a single task, not only the multiple users that share an FPGA-based DNN accelerator. For instance, recommendation systems [19] and speech recognition [5] can achieve a better system performance when co-locating multiple DNN models on a single GPU. However, last year, PREMA [12] began to develop a **Time-Division Multiplexing** (**TDM**) based scheduling method, in order to support multi-DNN inference workloads on the DNN accelerator, which is still far away from meeting the requirement of FPGA virtualization for INFaaS in the cloud. In this article, we enable the virtualization on a single FPGA of the node level, where a single-node multi-tenant DNN accelerator based on FPGA is also enabled for deep learning inference applications in the cloud.

It can be seen from Figure 1 that there are two typical scenarios in the cloud; the public and the private cloud. The key idea of virtualization in the first scenario is to ensure physical resources and performance isolation between the users. Physical resources isolation is an important requirement to guarantee the security of the users and the cloud vendors. The performance isolation means that the performance of each user should not be disturbed by the concurrent execution of multiple users. As for the second scenario, virtualization needs to provide computing power reconfiguration ability, in order to ensure that the overall system performance can be maximized under different situations with multiple users and dynamic workloads. However, to the best of our knowledge, there is no FPGA virtualization solution for general-purpose DNN applications, which can meet the requirements of both the public and private cloud, simultaneously.

Currently, two methods exist for sharing a single FPGA: TDM and **Space-Division Multiplexing** (**SDM**). The former hardly needs to re-program the FPGA, but schedules multiple tasks on the

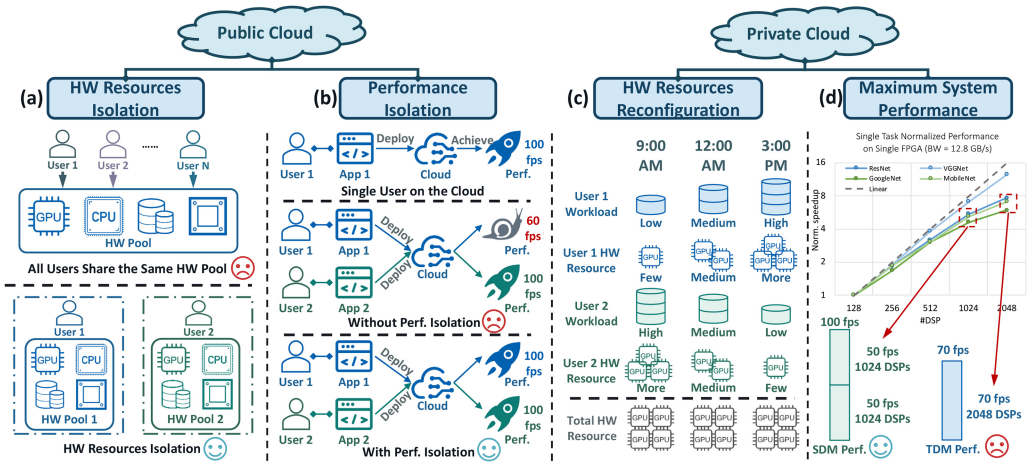


Fig. 1. From left to right: (a) Hardware resources isolation in public cloud. (b) Performance isolation in public cloud. (c) Resources reconfiguration in private cloud. (d) Maximum system performance in private cloud.

same physical resources in the form of time slices. However, it is not able to achieve the isolation of the physical resource, which results in poor security. On the contrary, SDM can easier achieve a better physical and performance isolation. In the sequel, we demonstrate that the proposed SDM-based multi-core hardware design can achieve the performance isolation of less than 1% deviation for multiple users.

In the private cloud, the computing resources allocated to a user or a task may change at any time by the central scheduler, due to the dynamic workload [12]. Current cloud-based DNN accelerators usually generate the configuration table during the offline deployment, which contains all the cases with different allocated hardware resources, so as to realize fast reconfiguration at the runtime. However, in the case of single-node multi-tenant sharing, template-based DNN accelerators take several days to generate all the bitstreams [26], while the DNN accelerators based on **Instruction-Set-Architecture** (**ISA**) require several hours to generate all the instruction files [12, 44]. The overhead of offline deployment will further deteriorate in the multi-node scaling-out scenarios. It then costs the cloud vendors thousands of dollars for the offline deployment of each single-node multi-tenancy [26].

Therefore, this article focuses on exploring a fast and general compilation framework for the virtualized ISA-based DNN accelerators, which can be deployed on both public and private cloud scenarios. The basic approach consists in decoupling the hardware-dependent and hardware-independent compilation processes. Consequently, we find that an optimization possibility that the overhead of the hardware-dependent compilation process is quite small. Thus, the lengthy hardware-independent compilation process can only be run once as a static compiler, in order to generate fine-grained instruction packages. A dynamic compiler can then generate lightweight hardware-related instructions for multi-tenancy support, by integrating and re-allocating the instruction packages according to the allocated hardware resources. The proposed two-stage compilation with negligible single-task performance loss can reduce the compilation cost of offline deployment to $\sim$1 minute, and can achieve $\sim$1 ms re-compilation overhead, so that the second stage can be deployed at the runtime. This is not feasible using a traditional one-stage compilation process. We believe that the proposed compilation framework will allow cloud vendors to optimize their maintenance costs, for both public and private cloud scenarios.

Another advantage of SDM-based sharing is that it can achieve a better system performance than the TDM-based sharing, in private cloud scenarios. More precisely, a non-linear relationship exists between the performance and the hardware resources of the ISA-based DNN accelerator, in the single-task scenario, due to the limited off-chip **bandwidth** (**BW**). To make a fair comparison, we refer TDM to the case where a single-large-core DNN accelerator is shared among multiple users on a single FPGA, while SDM denotes the DNN accelerator which consists of multiple small cores. It can be seen in Figure 1(d) that the single-task throughput is 50 fps and 70 fps, with a small core of 1,024 DSPs and 6.4 GB/s BW, and a large core of 2,048 DSPs and 12.8 GB/s BW, respectively. TDM will run with a single large core, resulting in an overall throughput of 70fps, while SDM will run with two small cores, achieving a better throughput of $50 \times 2 = 100$ fps using the same hardware resources in the multi-tenant scenario. In the sequel, we will show that the proposed SDM-based multi-core design can improve the system throughput compared to the TDM-based single-core design.

In this article, we propose a unified FPGA virtualization framework for general-purpose DNN in the public and private cloud scenarios, enabling multi-tenant sharing for both the ISA-based CNN and RNN accelerators on a single FPGA. The main contributions of this article are summarized as follows:

- We propose a multi-core based **Hardware Resources Pool** (**HRP**), in order to provide isolated and runtime-programmable DNN computing. We also design a two-level **Instruction Dispatch Module** (**IDM**) to provide low-overhead spatial multi-tenant sharing of FPGA resources.
- We propose a tiling-based instruction frame package design and a two-stage static-dynamic compilation, in order to overcome the heavy offline deployment and online reconfiguration overhead, while introducing a negligible single-task performance loss.
- We demonstrate the proposed unified FPGA virtualization framework on the CNN and RNN accelerators for both public and private cloud scenarios. Experimental results show that the proposed virtualized framework can achieve up to 3.12× and 6.18× higher system throughput compared with the static CNN and RNN baseline designs, respectively.

The following parts of this article are organized as follows. Section 2 presents related work and preliminaries. Section 3 introduces an overview of our ISA-based virtualization methodology. Sections 4 and 5 demonstrate the hardware architecture and the software compiler design on the two baseline DNN accelerator designs, respectively. We present our proposed virtualized FPGA system stack in Section 6. The experimental results are presented and analyzed in Section 7. Then, we conclude this article in Section 8.

## 2 RELATED WORK AND PRELIMINARIES

### 2.1 FPGA Virtualization in the Cloud

Hardware virtualization techniques are required to integrate FPGAs in the cloud. For instance, Vaishnav et al. [41] summarized the researches on FPGA virtualization and classified them into three categories; resource level, node level, and multi-node level. At the resource level, the hardware resources of the FPGAs are divided into reconfigurable resources (e.g., logic) and non-reconfigurable resources (e.g. I/Os). As for the virtualization of the reconfigurable resources on the FPGA chip, a commonly used method consists in implementing an intermediate overlay architecture [7, 13] between the high-level software framework and the low-level FPGA hardware. I/O virtualization enables the hardware resources sharing by different tasks, using the same I/O interface. Node level FPGA virtualization treats an FPGA chip as a computation node, while

multiple accelerators on the FPGA chip are used to simultaneously execute different tasks. Chen et al. proposed the FPGA virtualization architecture at both resource and node levels using a partial reconfiguration [9]. Knodel et al. also proposed a similar FPGA virtualization architecture [27]. Dai et al. further proposed the scheduling scheme over the partial reconfiguration architecture [14]. Multi-node level FPGA virtualization uses multiple FPGAs to provide an acceleration system for tasks. Microsoft proposed the Catapult system to accelerate its online Bing search application [36]. Baidu also proposed software-defined FPGA accelerators for deep learning and big data applications in their data center [34]. Baidu's XPU [33] was designed for diverse workloads using several tiny ISA-based cores. AmorphOS [26] proposed an FPGA virtualization framework using two different modes of fixed slots and globally co-locating, in order to support the multi-tenancy for general-purpose applications. However, this performs at the expense of lengthy and costly compilation overhead for offline deployment, which is impractical for multi-node scaling out in the real scenarios. In contrast to these FPGA virtualization frameworks that use full or partial reconfiguration for multi-tenancy support, this article proposes an ISA-based FPGA virtualization framework for deep learning inference applications, using a two-stage compilation framework with fully decoupled hardware-dependent and hardware-independent compilation processes. The proposed framework allows to achieve a low-overhead compilation for both offline deployment and online reconfiguration.

### 2.2 FPGA-based DNN Accelerators

Recent research on FPGA based DNN accelerators can be divided into two categories; (1) using templates based on **Register-Transfer Level** (**RTL**) or **High-Level Synthesis** (**HLS**) to map the target DNN model into several computation blocks [45, 46]. Such design flow requires to regenerate the bitstream file for each input DNN model, and to reprogram the FPGA. (2) introducing a customized ISA [2, 8]. The ISA-based DNN accelerators do not need to reprogram the FPGA, while it can reconfigure its DNN task by reloading the updated instruction files. However, this kind of accelerator requires a carefully optimized compiler [10, 44], in order to efficiently map the DNN models to the underlying hardware architecture.

A few existing researches focused on the FPGA-based DNN accelerators in the cloud. Chen et al. proposed an automated framework for mapping the DNN models to cloud FPGAs [11]. Han et al. proposed a sparse LSTM accelerator to deploy speech recognition applications in the cloud [20]. Large technology industries have also proposed DNN accelerator solutions in the cloud. For instance, Intel [6] used OpenCL to provide a DNN deployment framework and an accelerator in the cloud. Xilinx [42] designed an ISA-based CNN accelerator targeting the cloud FPGAs. However, most of the previously mentioned methods mainly focused on the performance optimization of running the DNN model in the single-task and static-workload scenarios in the cloud-based FPGA, without considering the characteristics of the multi-tenant and dynamic-workload scenarios in the cloud. Microsoft [16] proposed a multi-FPGA virtualization system framework to serve the DNN workloads in the cloud. It shows a high performance in terms of sharing multiple FPGAsHowever, it does not take into consideration the case of sharing a single FPGA, which is also crucial and requires more in-depth research. PREMA [12] first explored the multi-tenant sharing on the single node of the DNN accelerator using a TDM based method, which is also applicable to previous cloud FPGA-based DNN accelerators. In contrast to these TDM-based multi-tenancy supports on the single-large-core designs, this article first explores an SDM-based multi-small-core design for multi-tenant sharing of INFaaS workloads in the FPGA cloud.

It can be deduced from the literature review that few studies focused on both the FPGA virtualization and deep learning accelerators. Without considering the multi-tenant and

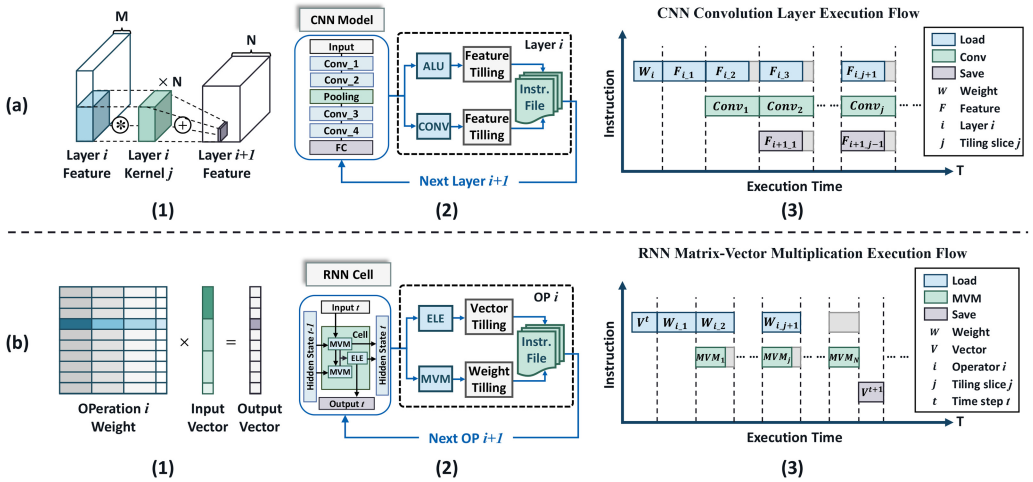


Fig. 2. Baseline designs of (a) CNN accelerator and (b) RNN accelerator. From left to right: (1) Key operators, (2) software compilation flow, and (3) execution data flow.

dynamic-workload applications in the cloud, such DNN accelerators are still far away from providing efficient and flexible FPGA virtualization solutions.

A general system stack of DNN accelerator mainly consists of the software compiler and the hardware architecture design. The execution data flow of the DNN accelerators is determined by the data dependence of the target neural network, instruction granularity of the software compiler, and module-level dependence of the hardware architecture. In this article, we demonstrate the proposed virtualization methodology on two different ISA-based baseline designs; CNN accelerator and RNN accelerator, which will be further introduced in the following subsections.

### 2.3 Baseline Design: CNN Accelerator

The operators of the CNNs contain a convolutional layer, pooling layer, activation layer, and so on. The convolutional layer is the key operator that accounts for more than 90% of the CNN's computation. For instance, in VGG-11, the convolutional layers contribution is 98.2% [18] of the operations. It can be observed in Figure 2(a-1) that the convolutional operation is three-dimensional, because it needs to consider the length and width of the feature map, and the different channels. The operation details are expressed as

$$f_j^{(i+1)}(x,y) = \sum_{c=1}^{M} \sum_{p=1}^{f} \sum_{q=1}^{f} \left[ f_c^{(i)}(x+p, y+q) * w_{jc}^{(i)}(p,q) \right], \tag{1}$$

where $i$ and $j$, respectively, represent the $i^{th}$ layer and $j^{th}$ convolution kernel (equivalent to the $j^{th}$ output channel), $x$ and $y$ represent the center coordinates of the input feature map and the pixel coordinate of the output feature map, $f$ and $c$, respectively, denote the patch size and the $c^{th}$ channel of the input feature map. It can be seen from in Equation (1) that convolutional operation takes the $i^{th}$ feature map patches and a series of convolution kernels as input, then outputs the $(i+1)^{th}$ feature map.

The baseline design of the ISA-based CNN accelerator is based on the Angel-Eye [17] and the Xilinx **Deep learning Processing Unit** (**DPU**) [44], as shown in Figure 4. The top-level hardware architecture, shown in Figure 4(a), contains an IDM and three instruction-related modules; the

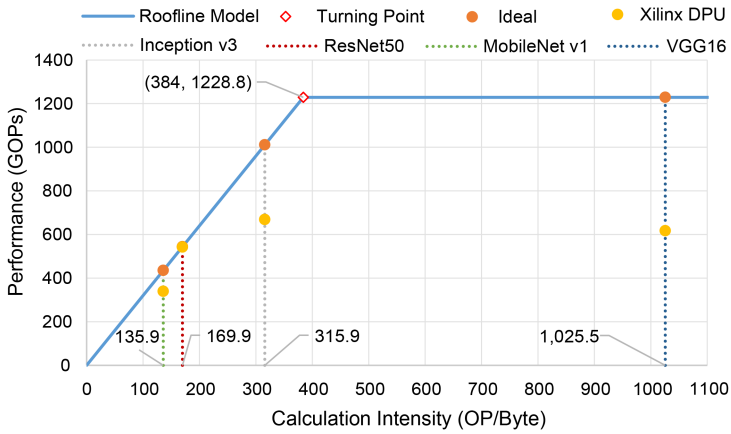


Fig. 3. The Roofline model of the baseline CNN accelerator using Xilinx DPU [44] with attainable performance (GOPs) versus calculating intensity (OPs/Byte). Performance is memory bound before the turning point and compute bound after the turning point.

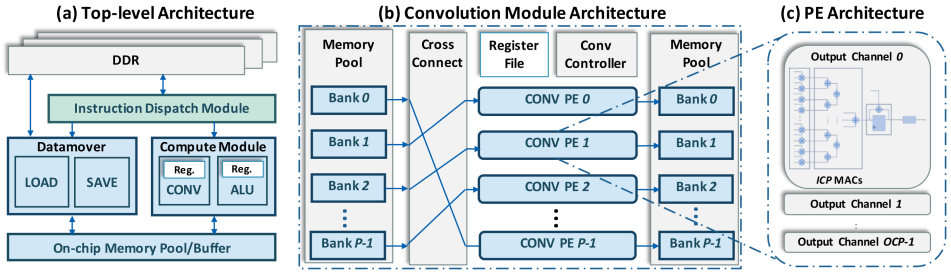


Fig. 4. Hardware architecture of the baseline CNN accelerator: (a) top-level architecture, (b) convolution module architecture, (b) PE architecture.

convolution computation module (CONV), the non-convolution computation module (MISC), and the datamover module (LOAD/SAVE). Figures 4(b) and (c) show the hardware architecture of the CONV module and the **Processing Element** (**PE**), respectively. The computation parallelism of the CONV module is given by

$$Parallelism = 2 * PP * ICP * OCP \quad (OPs/cycle), \tag{2}$$

where $PP$, $ICP$, and $OCP$ represent the parallelism along the pixel, the input channel, and the output channel dimensions of the feature map, respectively. Corresponding to the CONV module architecture, $PP$ is equal to the number of PE ($P$), which means that each PE completes the computation of one pixel of feature map per cycle. Inside each PE, there are a total of $OCP$ parallel computing channels, each corresponding to the computation of one output channel. Each computation channel can handle the **multiply-accumulate** (**MAC**) computation between the feature maps and weights of the $ICP$ input channels in each cycle. Therefore, the computation parallelism in Equation (2) needs to be multiplied by two.

Figure 3 shows the roofline model of attainable performance (GOP/s) versus the calculation intensity (OP/Byte). When the calculation intensity increases, the attainable performance increases before the turning point as the accelerator is memory bound. The attainable performance saturates after the turning point, as the accelerator becomes computation bound. Figure 3 marks the ideal

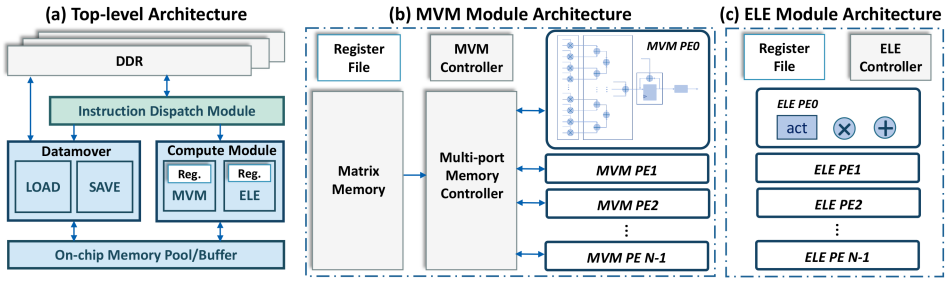


Fig. 5. Hardware architecture of the baseline RNN accelerator: (a) top-level architecture, (b) convolution module architecture, (b) PE architecture.

points of InceptionV3 [40], ResNet50 [21], MobileNetV1 [22], and VGG16 [39] for reference along with the actual points of Xilinx DPU.

As shown in Figure 2(a-2), the software compiler maps the CNN models into different instructions (e.g., *Conv*, *Misc*, *Load*, and *Save*) that can be run on the corresponding hardware functional modules. It takes the input CNN model's architecture information as a layer-wise **Data Flow Graph** (**DFG**), and generates instruction files in a layer-by-layer manner. Due to the limited on-chip memory and the large number of intermediate results, the software compiler needs to apply tiling methods on the feature map of each layer.

During the layer-by-layer execution flow shown in Figure 2(a-3), the CNN accelerator first loads all the weights of layer $i$ to the on-chip memory, then iteratively loads the slice $j$ of the feature map to the on-chip memory, then performs the convolution operation, and finally stores the feature map of the next layer back to the off-chip memory.

### 2.4 Baseline Design: RNN Accelerator

$$
\begin{aligned}
i_t &= \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i),\\
f_t &= \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f),\\
o_t &= \sigma(W_{ox}x_t + W_{oh}h_{t-1} + b_o),\\
g_t &= \tanh(W_{gx}x_t + W_{gh}h_{t-1} + b_u),\\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t,\\
h_t &= o_t \odot \tanh(c_t).
\end{aligned}
\tag{3}
$$

**Long Short-Term Memory** (**LSTM**) is an essential type of RNNs with a high ability to learn long-term data dependencies. The LSTM operators contain **Matrix-Vector Multiplication** (**MVM**) and three types of element-wise (ELE) operations, including addition, multiplication, and activation. Among them, MVM is the key operator which can take up more than 95% of the RNN's computation. It can be seen from Figure 2(b-1) that, unlike convolutional operation, MVM is a two-dimensional operation where each row of the weight matrix needs to perform a vector-by-vector multiplication with the input vector, in order to obtain one element of the output vector. The original LSTM network is shown in Equation (3), where the $\odot$ denotes the element-wise multiplication, $W$ denotes the weight matrices, $b$ represents the bias vectors, and $\sigma$ is the sigmoid activation function.

The baseline design of the ISA-based RNN accelerator is based on the row-wise block-striped decomposition method of MVM [30, 38] for dense LSTM networks, as shown in Figure 5. The computation modules of the RNN accelerator contain an MVM module and an ELE module, as

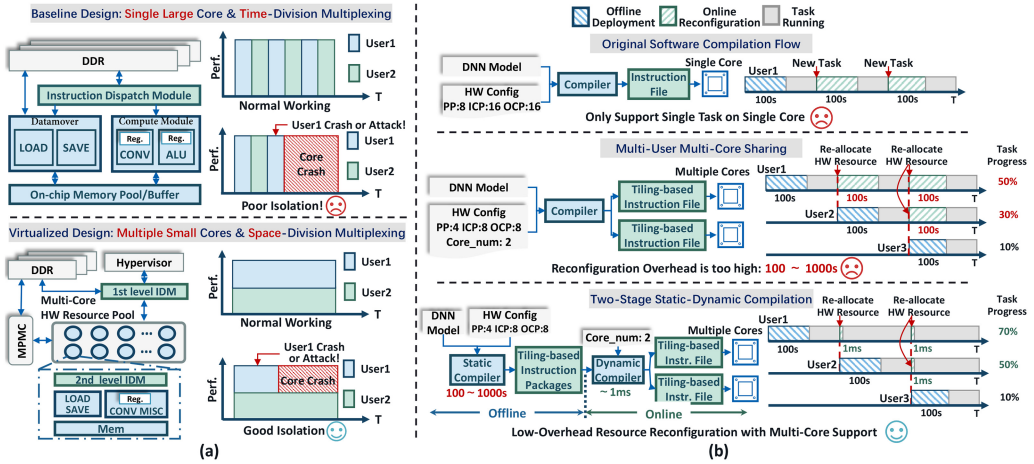


Fig. 6. Overall ISA-based virtualization methodology: (a) Virtualized ISA-based hardware architecture design targeted for the public cloud. (b) Low-overhead reconfiguration software compiler design for the private cloud. PP: pixel parallelism. ICP: input channel parallelism. OCP: output channel parallelism.

shown in Figure 5(a). Figure 5(b) presents the hardware architecture of the MVM module with its computation parallelism given by

$$Parallelism = 2 * HP * WP \quad (OPs/cycle), \tag{4}$$

where $HP$ and $WP$ denote the parallelism along the height and width dimension of weights, respectively. Corresponding to the MVM module architecture, $HP$ is equal to the number of PE $M$, which means that each PE completes the computation of one row of weights per cycle. Inside each PE, a total of $N$ MAC units exists, where $N$ is equal to $WP$.

It can be seen from Figure 2(b-2) that the software compiler takes the architecture information of the input RNN cell as an operator-wise DFG, and generates the instruction files in an operator-by-operator manner. Since the weight matrices of the MVM operator are much larger than the intermediate results, the software compiler of the RNN accelerators performs tiling methods on the weights for the MVM. As for the element-wise operators, a tiling method is applied to the input vectors.

Figure 2(b-3) shows the operator-by-operator execution flow of the RNN accelerator. For each time step $t$, it first loads the input vector to the on-chip memory, then iteratively loads the weight slice $j$ of operator $i$ to the on-chip memory, and finally performs the MVM calculation. Unlike the CNN accelerator, which needs to save the intermediate results of each layer back to the off-chip memory quite frequently, the RNN accelerator only needs to save the final results of each time step $t$ back to the off-chip memory. This is due to the fact that the intermediate results of the RNN are all one-dimensional vectors, while those of the CNN are all three-dimensional matrices. Thus, the intermediate results of the RNN can be stored completely stored in the on-chip memory.

## 3 ISA-BASED VIRTUALIZATION METHODOLOGY

This section gives a brief introduction to the proposed ISA-based virtualization methodology, which is applicable to any ISA-based DNN accelerator. As shown in Figure 6, in order to support both public and private cloud scenarios, simultaneous isolation and high performance are required. Thus, we propose virtualized ISA-based hardware architecture and a low-overhead

reconfiguration software compiler for the two baseline DNN accelerators, which will be discussed in Sections 4 and 5, respectively.

### 3.1 Isolation: ISA-Based Hardware Architecture with Space-Division Multiplexing

As shown in the upper part of Figure 6(a), current single-large-core DNN accelerator designs running in a TDM manner are neither reliable nor secure enough to support the multi-tenant scenarios of the cloud [28]. Concerning the reliability, if a user's application crashes or a part of the hardware fails, the running tasks of all the other users will be forced to stop, due to the crash of the same physical computing core that is shared by the multiple users. This potential reliability issue is not coherent with the requirements of the physical resources and performance isolation in the public cloud. On the contrary, SDM can easily satisfy the isolation requirements in the public cloud by allowing different users to monopolize different physical resources. In terms of security, TDM-based and SDM-based encapsulations are usually used together, in order to provide the best security among multiple users [25]. However, the existing single-large-core design cannot employ SDM-based security encapsulation on a single FPGA node, which leads to a failure in meeting the security requirements of multi-tenant sharing in the public cloud. To address these challenges, we propose a **multi-core HRP** technique to support the SDM based multi-tenant scenario, by dividing a single large core into multiple small cores. Thus, each user can monopolize a given number of small cores in the HRP to obtain a reliable, safe, and undisturbed operating environment.

A problem brought by the SDM consists in how to make multiple small cores simultaneously perform a single task. In fact, this is non-trivial because simply tiling the DNN model along the same dimension for multi-core sharing, could lead to severe performance degradation. This is due to the fact that different layers of the DNN models have different preferences for the three dimensions, as discussed in Section 2.3. Figure 6(b) presents the introduced **tiling-based instruction package design** to enable multi-core sharing with a negligible performance loss at the software level. The basic idea consists tiling the output feature map of each DNN layer using two orthogonal dimensions, without introducing an additional hardware modification. The proposed tiling-based **Instruction Frame Packages** (**IFPs**) can provide enough information for multi-core sharing, in order to minimize the single-task performance loss. In addition, we introduce a **two-level IDM** to handle multi-tenant task scheduling and single-task multi-core sharing synchronization control at the hardware level. The proposed two-stage IDM is designed to support multi-tenant sharing on ISA-based DNN accelerator with a minimal overhead of hardware resource. Therefore, it can be applied to any ISA-based DNN accelerator design for enabling virtualization in the cloud.

### 3.2 High Performance: Low-Overhead Reconfiguration Software Compiler Design

When multiple users share the virtualized multi-core resources, the instruction files of each user need to be re-compiled and sent to the corresponding cores each time the resources are re-allocated. An intuitive method consists in generating all the instruction files that contain all the resource allocation situations during the offline deployment, thus avoiding the lengthy online re-compilation overhead, at the expense of an intolerable compilation overhead and very high maintenance costs. On the other hand, the current compilation process of the ISA-based DNN accelerator is also too slow to be deployed during the runtime, which could lead to severe tail latency of serving multi-tenant INFaaS in the private cloud scenario. To tackle these challenges, we propose a **two-stage static-dynamic compilation**, where the original compilation flow is divided into two parts; static compiler and dynamic compiler, as shown in Figure 6(b). The former can be considered as a hardware-independent compilation process, that generates the tiling-based IFPs during offline deployment, based on the input DNN model and the hardware configuration of the basic shareable unit. It is a pre-processing stage that only needs to run once, as decoupled with the operations

related to the hardware resource allocation. The latter is a hardware-dependent compilation process able to generate the final instruction files in a very short time, since it only needs to re-allocate the pre-generated tiling-based IFPs to each core, based on the re-allocated hardware resources for each user. We demonstrate that the proposed two-stage compilation process can reduce the overhead of the offline deployment to almost 1 minute, by carefully decoupling the hardware-dependent and hardware-independent compilation process. Moreover, it can achieve about 1ms overhead of online reconfiguration, by only recompiling the light-weight runtime information. This ensures that the task progress of each user is hardly affected even in the case of frequent dynamic reconfiguration.

## 4 HARDWARE ARCHITECTURE FOR VIRTUALIZATION

At the hardware level, a two-level IDM is introduced to handle the multi-tenant task scheduling and the single-task multi-core sharing synchronization control. A multi-core HRP is then proposed to provide isolated and runtime-programmable hardware resources in an SDM-based sharing manner, for multi-tenant scenarios in the cloud.

### 4.1 Two-Level IDM

The main function of the original IDM is to implement the instruction distribution and dependency management in a single core. As for the hardware design for virtualization, the shareable units in the multi-core HRP also need to be scheduled for the multi-tenant support. Thus, a two-level IDM is required to schedule and manage the hardware resources of two different dimensions, where the first level IDM is a task-level scheduler and the second level IDM is a module-level scheduler.

*4.1.1 First Level IDM.* The first level IDM can be considered as a task-level scheduler. Its basic components are shown in Figure 7(b). The on-chip instruction memory fetches the instructions from DDR, and caches them until the next reconfiguration. The instruction decoder sends the instructions to the second level IDM of the corresponding core, according to the core index of each instruction.

The context-switch controller records the context information after receiving the reconfiguration signal from the hypervisor. It currently supports two context-switch modes; task-level and layer-level switching. The first mode only needs to wait for the implementation of the current inference task, and then loads new instructions into each core. In the second mode, the CNN layer index of each user is the context information to be recorded. Since the baseline design works in a layer-by-layer manner, intermediate data such as feature map will be written back to the DDR when a layer calculation is finished. Therefore, there is no need to record it as context information. Afterwards, the context-switch controller loads the updated instructions and layer index to all the cores of each user, so that the computing cores can continue to calculate from the next layer.

The multi-core synchronization controller manages the layer-wise multi-core synchronization. The hypervisor configures this module to define which cores need to be synchronized. It will then read in the *sync_local* signals of all the cores belonging to each user. Only when all the *sync_local* signals are valid, it will send a valid *sync_global* signal to each core, so that these cores can start the calculation for the next layer.

*4.1.2 Second Level IDM.* The second level IDM can be considered as a module-level scheduler inside each core. As shown in Figure 7(b), the lower two sub-modules are specifically designed for virtualization. The context-switch module restarts the computation, based on the context information recorded by the first level IDM in the online reconfiguration stage. The system synchronization controller generates the *sync_local* signal when the *System* instruction for synchronization of the current layer satisfies the dependency constraints. This means that the

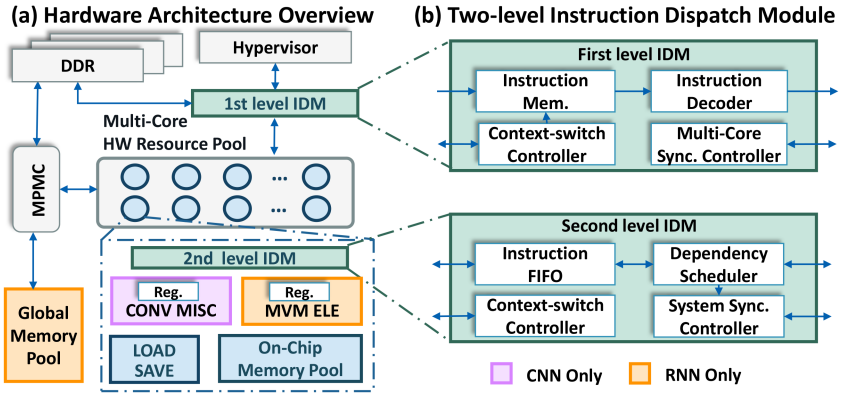


Fig. 7. The hardware architecture design of DNN accelerator for virtualization: (a) hardware architecture overview, (b) two-level IDM. The differences between CNN and RNN virtualized accelerator designs are marked by purple and orange, respectively.

computation of the current layer has been finished. Afterwards, it suspends the scheduler to stop dispatching instructions until it receives a valid *sync_global* signal for the next layer to start.

*4.1.3 RNN Accelerator Case Study.* The two-level IDM of the virtualized RNN accelerator has the same functional modules as the virtualized CNN accelerator. As discussed in Section 2.4, the network architecture of the RNN consists of multiple network layers, each having an LSTM unit. Each LSTM unit can be further divided into various operators with complex data flows. Thus, the context-switch controller, in the two-level IDM, supports a new context-switch mode, which is the operator-level switching. This latter is consistent with the software compiler working in an operator-by-operator manner. At the operator-level switching mode, intermediate data of vectors and operator index will be recorded as context information. Note that such context-switch overhead is small, since the intermediate data of the RNN consists of only one-dimensional vectors, while a three-dimensional feature map is used in the CNN. The multi-core synchronization controller is used to manage the operator-wise multi-core synchronization, using the *System* instruction. The *sync_local* and *sync_global* signals are processed between the first and second level IDM, in order to control the operator-wise data dependency flow.

## 4.2 Multi-Core HRP

The virtualized HRP mainly consists in dividing the single large core in the baseline design into multiple small cores, to provide isolated and runtime-programmable hardware resources. In this section, the implementation techniques are discussed to ensure hardware resources isolation and performance isolation, while meeting the requirements of the public cloud.

*4.2.1 Hardware Resources Isolation.* The shareable on-chip physical resources include on-chip memory, on-chip BW, and PE arrays consisting of DSPs. The multi-core design fundamentally isolates these physical resources. Each core cannot access the memory or the computing resources of others. In addition, each core can only perform basic synchronous interaction with other cores, through the first level IDM. Note that the off-chip DDR memory and the BW are other crucial physical resources. Since the cloud FPGAs, such as Xilinx VU9P and U200, usually have up to four independent DDR banks, the safest approach is that each user can monopolize single or multiple DDR chips. However, this limits the maximum number of users that a single FPGA can serve.

Sharing a single DDR by multiple users requires a proper isolation at the operating system level. Note that this is not discussed in this article.

*4.2.2 Performance Isolation.* The performance loss, caused by multiple users, is mainly a result of the competition for the same physical resources. Accordingly, the competition can only occur when multiple users share a single DDR. Since each DDR has only a single 512-bit data port, if the total demand for the memory BW of multiple users exceeds 512 bits, the performance will inevitably be deteriorated due to the competition among multiple data ports. Thus, the total bit width of the multi-tenant data ports cannot exceed the data bit width of a single DDR data port. This consists of a basic hardware restriction. In addition, a well-designed arbiter is needed to ensure that the performance crosstalk between the multi-tenant data ports is minimized. This has been widely studied and tested in recent years [43]. By using the discussed techniques, we can ensure that the FPGA virtualization design, based on the multi-core HRP, achieves a good isolation performance in the multi-tenant public cloud scenarios.

*4.2.3 RNN Accelerator Case Study.* As for the hardware resource isolation, each core owns a PE array of MVM computation module, a vector-wise operator PE array of ELE computation module, and a separate on-chip memory bank for weight matrix. Since the vector's intermediate data need to be synchronized and updated among the multiple cores, a vector memory pool is shared by all the cores. The vector memory is responsible for storing and updating the intermediate results between the operators of each time step $t$. Vector data is written back to the off-chip DDR memory, only when the final result calculation is completed or a context switch occurs.

As previously mentioned in Sections 2.3 and 2.4, the intermediate results of CNN are three-dimensional data that can be up to tens of megabytes, while intermediate results of RNN are few kilobytes of one-dimensional data. This difference in intermediate data volume makes the multi-core data synchronization mechanism different between virtualized CNN and RNN accelerators. More precisely, CNN accelerators use the off-chip DDR memory, while RNN accelerators use the on-chip vector memory pool, for data synchronization.

As for the performance isolation, the competition for the same physical resources mainly happens on the vector memory pool. To minimize the performance crosstalk of multi-tenant execution, a vector memory pool and a **multi-port memory controller** (**MPMC**), with a well-designed arbiter, are required. The former sets up multiple groups of the vector memory pool so that each user can monopolize a single vector memory resource. Since the resource utilization of the vector memory pool is relatively small compared with that of the weight matrix memory, this exchanging hardware resources method for performance isolation is acceptable for RNN virtualization.

The latter is similar to the virtualized CNN accelerator techniques. It needs a well-designed arbiter [43] to ensure that each core and each user can obtain stable and crosstalk-free off-chip DDR BW and on-chip vector memory BW. Using these techniques, the multi-core HRP based virtualized RNN accelerator can ensure a good resource and a strong performance isolation, in the public cloud scenarios.

### 4.3 Multi-Core BW Sharing

The MPMC, shown in Figure 8(a), is a fair bus arbiter used to share a single DDR with multiple virtualized cores. The MPMC, such as HyperConnect [37], guarantees an equal BW distribution among multiple memory ports and provides hypervisor-level performance isolation. Each port of the MPMC exposes a 512-bit AXI-4 memory bus with five channels; **Address Read** (**AR**), **Read Data** (**R**), **Address Write** (**AW**), **Write Data** (**w**), **Write Acknowledgement** (**B**), as shown in Figure 8(b). Each channel implements a ready-valid handshake protocol.

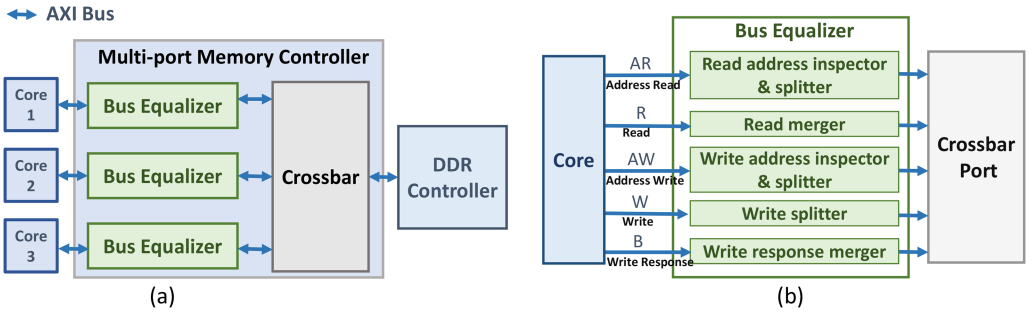


Fig. 8. MPMC hardware architecture. (a) Each core connects to MPMC's crossbar via AXI bus through a Bus Equalizer. (b) The architecture of Bus Equalizer. Bus Equalizer manages five channels of AXI ports, splitting and merging transaction requests to ensure fair BW distribution. Bus Equalizer also monitors read and write address channels to block illegal access, to guarantee core isolation.

*4.3.1 Fair BW Distribution.* To perform an equal BW distribution among the ports, the MPMC implements bus equalizers for reading and write channels. A bus equalizer has two functionalities; (1) limiting the burst size of each memory port to a nominal burst size. When the burst size is larger than the nominal size, the read or write requests will be split into multiple sub-requests and the responses will be merged. (2) limiting outstanding transaction numbers. The equalizer has an internal counter for ongoing sub-requests. Once the maximum unfinished transaction number is reached, further transaction requests are suspended. When the number of ongoing sub-requests is less than the threshold, new transaction requests will start being processed. The bus equalizers are connected to a round-robin crossbar which processes one sub-request at a time. The bus equalizer with a round-robin arbiter guarantees a fair BW distribution among the cores.

*4.3.2 Core Isolation.* The MPMC achieves isolation using two techniques; (1) controlling the ready/valid handshaking signals. Each HA port's handshaking signals are separately controlled by the MPMC, which allows the system to enable/disable any single core at the hardware level. (2) restricting the accessible physical address. MPMC inspects the read and writes addresses of each HA port's AR and AW channels. The legal memory address range can be set by the server daemon, while illegal transaction requests will not be processed by the bus equalizer.

## 5 SOFTWARE COMPILER FOR VIRTUALIZATION

At the software level, a tiling-based instruction package design is proposed. It uses two orthogonal tiling dimensions without introducing an additional hardware modification, so as to enable multi-core sharing with a negligible single-task performance loss. A two-stage static-dynamic compilation is also proposed to minimize online reconfiguration overhead, where the original compilation flow is divided into a static compiler and a dynamic compiler.

### 5.1 Static Compilation

It can be seen in Figure 9(a) that the static compiler generates tiling-based IFPs in a layer-by-layer manner, and applies a latency simulator to obtain a latency **Look-Up-Table** (**LUT**). Both the tiling-based IFPs and latency LUT are cached for the dynamic compiler during the online reconfiguration.

*5.1.1 Tiling-Based Instruction Frame Packages.* In order to support the multi-core sharing of a single task, the output feature map of each layer should be tiled into several independent sub-tiles for parallel computing. Due to the fact that the compiler generates *Conv* instructions along the

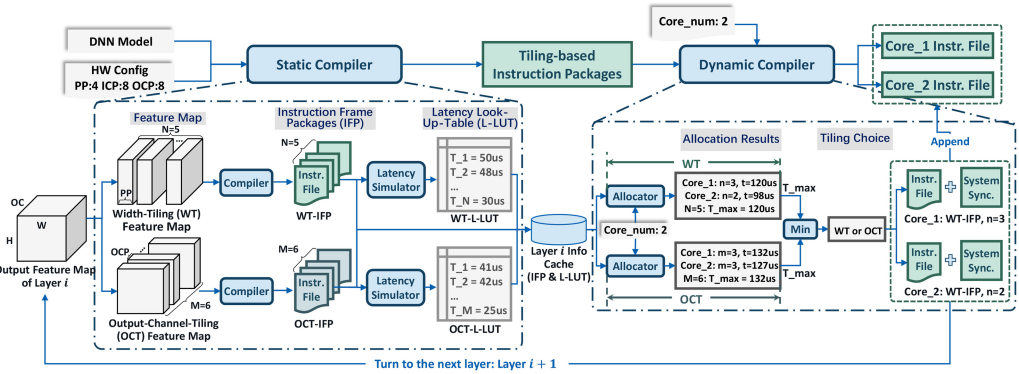


Fig. 9. The software compiler design of CNN accelerator for virtualization. The left part is the static compiler, where the output feature map of each layer is tiled along two different orthogonal dimensions, then the corresponding IFPs and latency LUTs are generated and cached for later use during runtime. The right part is the dynamic compiler, where a workload-balanced instruction allocator is proposed to find the optimal tiling dimension for each layer and map the IFPs to different cores. Finally, the synchronized *System* instructions are added to support layer-wise multi-core synchronization control for the final instructions.

height dimension, tiling along the height dimension results in a complex dependency relationship between the IFPs. Thus, the output channel and width dimensions are chosen as the target tiling dimensions, in order to generate independent IFPs.

The additional overhead introduced by multi-core sharing is encountered because each core needs to load the same data from the DDR to the on-chip memory, thus reducing the overall data reuse efficiency. On the other hand, the tiling along the output channel can be considered as weight parallelization. In other words, each core will load a different part of weights with the same input feature map. On the contrary, tiling along the width loads a different part of the input feature maps but the same weights. Since the input feature map and the weights of each layer are different, different layers will have different preferences for the two tiling methods, in order to obtain a better latency. Therefore, the performance loss of multi-core sharing can be minimized by choosing the proper tiling method for different layers.

*5.1.2 Latency Simulator.* A fast latency simulator is designed, in order to obtain a cycle-accurate latency evaluation of each tiling-based IFP. The latency of the *Conv* instruction is estimate based on the computation amount and the computation parallelism:

$$t = \frac{Ch_{in} * Ch_{out}}{ICP * OCP} * Width_{out} * Kernel_w * Kernel_h * T, \tag{5}$$

where $T$ denotes the clock cycle. The latency of the instructions used for data movement (e.g., *Load* and *Save*) can be estimated using the total length of the data, and the BW:

$$t = \frac{Length_{data}}{BandWidth * eff}. \tag{6}$$

Where $eff$ is a BW efficiency parameter. Afterwards, a directed acyclic graph $G(V, E)$ is set up according to the data and hardware dependencies of the instructions, where $V$ represents each instruction in the IFP and $E$ is used to store the dependencies between instructions. By traversing the entire graph $G$, we can get the latency estimation of the IFP and store it into a latency LUT.

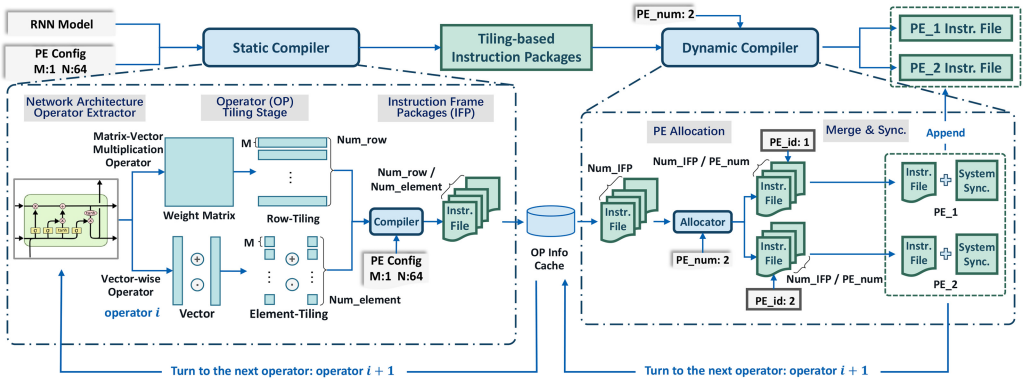


Fig. 10. The software compiler design of RNN accelerator for virtualization. The static compiler on the left part employs row-tiling and element-tiling for MVM and vector-wise operations, respectively. Since there is only one tiling dimension, no latency simulator is required and only the tiling-based IFPs are generated and cached for later use during runtime. The dynamic compiler on the right part maps the tiling-based IFPs to different cores and appends synchronized *System* instructions to generate the final instructions.

*5.1.3 RNN Accelerator Case Study.* Concerning the tiling-based instruction frame packages, different tiling dimensions are applied for MVM operations and vector-wise operations, as shown in Figure 10. Since our baseline RNN accelerator utilizes a row-wise block-striped decomposition method for MVM operations, it is trivial to apply the row-wise tiling on the weight matrices, thus providing several independent sub-tiles of weight matrices with $M$ rows for multi-core sharing of a single task. As for the vector-wise operations, each input vector is tiled into several sub-vectors of size $M$, where $M$ is equal to the computation parallelism of the ELE module in each core, as shown in Figure 5.

Due to the multi-dimensional data reuse in the convolution operation of the CNN accelerators, tiling-based IFPs along the width or output channel dimension will inevitably introduce a performance loss when compared with the instruction files optimized with the whole layer. As for the RNN accelerator, the input vector of the MVM operation should be multiplied by each row of the weight matrix. This means that data reuse only exists in the column dimension of the weight matrix. Therefore, row-tiling IFPs of MVM operations will introduce no performance overhead, for the virtualized RNN accelerator. Since there is no need to find a better tiling method for each layer, the latency simulator can be removed, and the static compiler only needs to generate tiling-based IFPs, in an operator-by-operator manner for the dynamic compiler.

## 5.2 Dynamic Compilation

Figure 9(b) shows the workflow of the dynamic compiler. Firstly, the dynamic compiler fetches the latency LUTs of the two tiling methods from the cache for each layer. The allocator will then find the optimal allocation scheme for multi-core sharing, in order to minimize the latency of the current layer, according to the number of re-allocated cores. Afterwards, the dynamic compiler chooses the tiling method with minimal latency as the target strategy for the current layer. The dynamic compiler takes the corresponding IFPs from the cache, combines them into multiple instruction sequences according to the optimal allocation scheme, and finally adds a synchronization *System* instruction at the end of each instruction sequence. The dynamic compiler repeats this process until the instructions of all the layers are generated.

*5.2.1 Workload-Balanced Instruction Allocator.* For each layer, given the number of tiling-based IFPs ($N$) and the number of allocated cores ($Num_{kernel}$), a workload-balanced IFP allocation method ($Alloc$) is needed. This problem can be modeled as an optimization problem:

$$\arg\min_{Alloc} \max_{k=1}^{M} \sum_{i=1}^{N} Alloc(i,k)T(i), \tag{7}$$

$$\sum_{k=1}^{Num_{kernel}} Alloc(i,k) = 1, \forall i \in \{1,\ldots,N\}, \tag{8}$$

$$Alloc(i,k) \in \{0,1\}, \forall i \in \{1,\ldots,N\}, \forall k \in \{1,\ldots,M\}, \tag{9}$$

where $Alloc(i,k) = 1$ denotes the $i^{th}$ IFP is allocated to the $k^{th}$ core. $T(i)$ denotes the latency of the $i^{th}$ IFP. This optimization problem can be solved using dynamic programming methods.

*5.2.2 Layer-Wise Multi-Core Synchronization.* Since the computing workload of each core in the same layer is not exactly the same, a synchronization mechanism should be introduced to ensure correct data dependencies. A synchronization bit is added to the function field of the *System* instruction. When each core runs to the synchronization *System* instruction, it stops running and enters the waiting state. The calculation of the next layer can start only when all the cores of a user have run to the *System* instruction of the current layer.

*5.2.3 Context Switching Analysis.* The context switching cost is mainly composed of two parts; (1) $T_{recompile}$, i.e., the compiling time to regenerate the instruction files. (2) $T_{transfer}$, i.e., the time to send the new instruction files to the CNN accelerator. The overall context switching cost ($T_{context}$) can be estimated as

$$T_{context} = T_{recompile} + T_{transfer}, \tag{10}$$

*5.2.4 RNN Accelerator Case Study.* It can be observed in Figure 10 that the instruction allocator in the dynamic compiler only needs to allocate and pack the tiling-based IFPs of each operator, based on the allocated resources. In order to realize the workload-balanced multi-core sharing of a single task, the instruction allocator will evenly distribute tiling-based IFPs to each core. An operator-wise multi-core synchronization mechanism is then applied after the instruction allocator, to add a synchronized *System* instruction at the end of each operator's instruction file for each core.

The context switching cost is mainly composed of three parts, as shown in Equation (11). The first two terms are similar to those of the virtualized CNN accelerator, as shown in Equation (10). The last term ($T_{transfer\_vec}$) denotes the time needed to send the context information of the intermediate vectors back to the RNN accelerator.

$$T_{context} = T_{recompile} + T_{transfer\_inst} + T_{transfer\_vec}. \tag{11}$$

## 6 SYSTEM STACK

### 6.1 Distributed Functional Components

Figure 11(a) presents a distributed FPGA virtualization system cluster. The cluster has multiple client servers and multiple accelerator servers. Client servers have virtual machine containers and client runtime to deploy DNN applications, while accelerator servers are equipped with virtualized FPGA cards to run the applications and support the dynamic reconfiguration. All the servers are jointly controlled by a Control Node, which is responsible for task scheduling and data synchronization. Note that users can choose to run one application on multiple servers, using a batch mode.

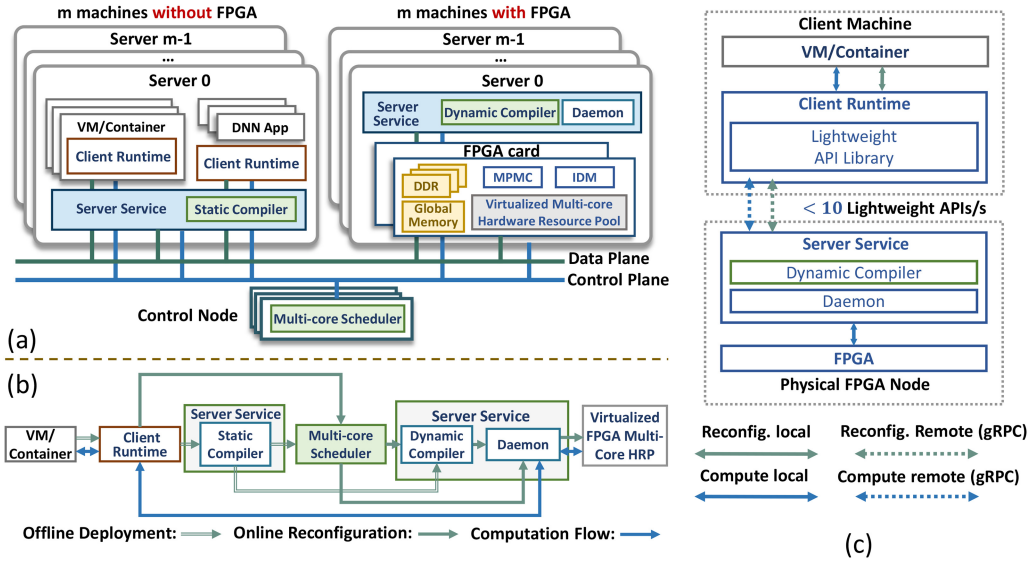


Fig. 11. The FPGA virtualization system stack and remote access flow. (a) The FPGA virtualization server cluster and system services. (b) Offline deployment, online reconfiguration and computation dataflow. (c) Low-overhead remote access dataflow with minimum remote API calls.

*6.1.1 Client Runtime.* The container or virtual machine of the client servers provides the runtime environment to deploy DNN models. More precisely, the client runtime provides a runtime SDK with APIs to compile DNN models, input/output data, request/release virtualized FPGA resources, and launch/close tasks. The APIs are designed to be lightweight and do not require the user's modification of the DNN inference code.

*6.1.2 Server Daemon.* The server daemon manages the virtualized multi-core FPGA resources. It uses the **Xilinx runtime** (**XRT**) library for low-level control and management of Xilinx PCIe-based FPGA cards. The server daemon sets the allocated resources scheduled by the control node, and sends the instruction files generated by the dynamic compiler to the target virtualized cores on FPGA. It also controls the data I/O to the accelerator, and initiates data movement between the client and accelerator servers. Moreover, it is responsible for controlling context switching, and monitoring the status of virtualized accelerators on the FPGA.

*6.1.3 Data Plane.* During the running applications process, the data plane is used to transmit data between the client and accelerator nodes. It is designed to support various high-BW and low-latency back-end data transmission protocols, including TCP/IP Ethernet, RoCE RDMA, Infiniband RDMA, and so on.

*6.1.4 Control Plane.* The control plane based on TCP/IP sends control signals to manage the whole cluster. The functional components, distributed in each node, remain synchronized with the central control node through the private transmission protocols of the control plane.

## 6.2 Remote Access Dataflow

Figure 11(b) shows the deployment and computation dataflow, and illustrates how functional components interact with each other. As for the offline deployment, the user sends the DNN model through client runtime to the static compiler, then the tiling-based IFPs generated by the

**ALGORITHM 1:** Token-Based Scheduling Algorithm Based on PREMA [12]

1: **initialization**
2: **for** each task $k \in$ task queue $K$ **do**
3: $k.latency = Latency(k, cores = 1)$
4: $slack = k.QoS - k.latency$
5: $k.token \leftarrow k.priority \times slack$
6: **end for**
7: **for** each task $k \in$ task queue $K$ **do**
8: $k.token+ = k.priority \times Slowdown_{normalized}$
9: **end for**
10: $Candidates = [k\ for\ k \in K\ if\ k.token > Threshold]$
11: **for** each task $k \in Candidates$ **do**
12: **if** $Environment = PrivateCloud$ **then**
13: $k.cores = AllocateCores(k, idle\_cores)$ // For spatial multi-core sharing in private cloud
14: **else**
15: $k.cores = user.cores$ // For spatial multi-core sharing in public cloud
16: **end if**
17: **end for**
18: **return** *Candidates*

static compiler are sent to the scheduler and dynamic compiler. When the online reconfiguration is initiated from the client runtime, the scheduler sends allocated virtualized core resources to the dynamic compiler, which then *locally* generates instruction files and passes them to the server daemon. During inference, the user launches a task from the virtual machine or container with the APIs provided by the client runtime, while the server daemon offloads the data and instruction files to target physical cores and sends back results. The decoupling of static-dynamic compilers, and the use of server daemon for low-level controls in the physical FPGA node, removes most of the API remoting overhead for INFaaS in the cloud.

### 6.3 Multi-Core Scheduler

Inference tasks have different priorities. The scheduler needs to ensure that the latency-critical tasks meet their timing requirements, and to avoid starving low-priority tasks. The scheduler adapts the priority-aware "token"-based task scheduling policy in PREMA [12], as detailed in Algorithm 1. The scheduler has two steps; (1) allocating each task with a number of tokens and choosing the candidate tasks to be executed next, (2) allocating each candidate task with a number of cores. In the first step, the number of tokens for each task is dynamically adjusted. Higher-priority tasks get more tokens and are more likely to become candidates, while the tokens of the low-priority tasks proportionally increase to their slow-down during the waiting period. As for the second step, since PREMA is only designed for TDM-based sharing of a single large core, the scheduling algorithm is modified to support the SDM-based sharing of multiple small cores, as shown in lines 11–17. The scheduler allocates cores to each candidate task by solving an optimization problem (for private cloud, line 13), or using a predetermined value provided by the users (for public cloud, line 15).

It is important to mention that the scheduling algorithm is mainly used in the private cloud scenarios. Since public cloud users have a fixed number of cores during the whole processing lifetime, the objective is to ensure that each user's performance is stable and undisturbed. The multi-core scheduler will only assign the predetermined core number, as requested by each user

to the corresponding task. As for private cloud users, the core number for each candidate task is dynamically determined during the runtime, in order to maximize the performance and resource utilization of the whole system.

*6.3.1 Resource Allocation.* The virtualized core allocation aims at maximizing the average system performance of all the candidate tasks in the private cloud. The latency $Latency(k, n)$ of a given task $k$, executing on $n$ cores, can be estimated as described in Section 5.1.2.

$$Perf(k, n) = \begin{cases} 0 & n = 0 \\ \frac{1}{Latency(k,n)} & n > 0 \end{cases}, \tag{12}$$

$$\sum_{k=1}^{K} N_a^k = N_{idle}, N_a^k \geq 0. \tag{13}$$

The resource allocation problem can be formulated as an optimization problem with the objective $Perf(k, n)$ expressed in Equation (12), and with the constraints given in Equation (13). The optimization problem can be solved using the **Integer Linear Programming** (**ILP**) during the runtime with negligible overhead.

## 7 EXPERIMENTS

### 7.1 Experiment Setup

*7.1.1 Hardware Platform.* We evaluate our virtualized design on two different hardware platforms. For the private cloud scenario, we set up a local machine with Xilinx Alveo U200 FPGA and an Intel Xeon 4210 CPU running at 2.2 GHz. For the public cloud scenario, we choose the f3 instance in Aliyun, which contains a Xilinx VU9P FPGA and an Intel Xeon Platinum 8163 CPU running at 2.5 GHz. The GPU platform for isolation evaluation consists of an Nvidia Tesla V100 GPU and an Intel Xeon Gold 6132 CPU running at 2.60 GHz.

*7.1.2 Software Environment.* We use Xilinx SDAccel 2018.3 for hardware synthesis and software deployment in both the local machine and the Aliyun cloud environment. The software compiler design for virtualization is implemented in Python, and the host application is developed using C++ with OpenCL APIs provided by SDAccel. To enable performance isolation on GPU, we use Nvidia CUDA **Multi-Process Service** (**MPS**) [31] by setting the environment variable `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE`. The latest NVIDIA A100 GPU [32] introduces a new feature of **Multi-Instance GPU** (**MIG**) for multi-tenant sharing. However, it does not support dynamic reconfiguration at runtime, and each online reconfiguration needs to restart the virtual machine. Thus, we do not use NVIDIA A100 GPU for the performance isolation evaluation.

*7.1.3 CNN Models and Accelerator Configurations.* We evaluate four CNN models with the image size of 224 × 224: VGG16 [39], ResNet50 [21], Inception v3 [40], and MobileNet [22]. We use TensorFlow [1] for both FPGA and GPU. CNN models are quantized to INT8 format for FPGA deployment. For the baseline CNN accelerator design, we set up two different configurations: a static single large core design with the parallelism of 8,192 and a static multi-core design with 16 small cores, each of which has a parallelism of 512. We employ the same hardware configuration as the static multi-core design for our virtualized multi-core CNN accelerator. All the CNN accelerators run at 300 MHz. The memory BW of each small core is 128 bits, and all the four DDR banks on FPGA are utilized. We employ one DSP for two INT8 MAC operations.

Table 1. Hardware Resources Utilization of the Static Designs and Virtualized DNN Accelerators on Xilinx U200 and VU9P FPGA

| FPGA | Implementation | LUT | FF | BRAM | URAM | DSP |
|---|---|---|---|---|---|---|
| U200 (Local) | User Budget | 891k | 1,960k | 1,153 | 960 | 6,824 |
| | Static CNN design (1 × 8,192) | 242k | 233k | 235 | 168 | 2,048 |
| | Static CNN design (16 × 512) | 418k | 390k | 395 | 307 | 2,048 |
| | Virtualized CNN design (16 ×512) | 436k | 402k | 416 | 320 | 2,048 |
| | Static RNN design (1 × 2,048) | 124k | 196k | 265 | 256 | 1,024 |
| | Static RNN design (16 × 256) | 136k | 365k | 300 | 256 | 1,024 |
| | Virtualized RNN design (16 × 256) | 141k | 370k | 316 | 256 | 1,024 |
| VU9P (Aliyun f3) | User Budget | 891k | 1,961k | 1,153 | 960 | 6,824 |
| | Static (1 × 8,192) | 242k | 232k | 235 | 168 | 2,048 |
| | Static (16 × 512) | 419k | 390k | 395 | 307 | 2,048 |
| | Virtualized CNN design (16 × 512) | 436k | 401k | 416 | 320 | 2,048 |

The total parallelism is 8,192 and 2,048 for CNN and RNN accelerators, respectively.

*7.1.4 RNN Models and Accelerator Configuration.* We evaluate the single-layer dense LSTM model presented in Section 2.4, with the hidden dimension of 100, 400, 800, and 1500. LSTM models are quantized to INT16 format for FPGA deployment. For the baseline RNN accelerator design, we evaluate the static single large core designs with the parallelism of 32, 512, and 2,048, and the static multi-core designs with 16 small cores, each with 1, 32, and 256 multipliers. The configurations of the virtualized RNN accelerator designs are the same as the static multi-core design. All the RNN accelerators run at 200 MHz and share four DDR banks. We employ one DSP for one INT16 MAC operation in each multiplier. The parallelism of the RNN accelerator is less than that of the CNN accelerator, because the bottleneck of RNN computation is the data transmission of weights between the on-chip and off-chip memory, instead of the MVM and ELE computation. We will further analyze the differences between the computation and memory access of the RNN accelerators under different hardware configurations. Besides, we employ the batch size of one for both CNN and RNN inference tasks.

### 7.2 Resources Utilization and Context Switch

*7.2.1 Hardware Resources Utilization.* As shown in Table 1, the static multi-core CNN design utilizes nearly twice more logic and memory resources than the static single-core CNN design with the same DSP resources on Xilinx U200 and VU9P FPGA. This is because there are multiple copies of each module in the multi-core design that can be reused in the single-core design. As for our virtualized multi-core CNN design, it introduces about 1% logic and memory resources overhead compared to the static multi-core design, which mainly comes from the two-level IDM design. The primary resource utilization of the current virtualized CNN accelerator design is logical resources, which occupy nearly 50% of LUTs and FFs. This means that we can reduce resource utilization by optimizing the logic and data paths of the single-core, thereby further scaling the virtualization design up to a higher degree of parallelism in our future work.

Table 1 shows that the logic resources overhead of the virtualized RNN design is about 3.7% and 1.4% for LUTs and FFs, respectively. The memory resources overhead is 5.3% for BRAMs since there is an additional vector memory pool for the virtualized RNN accelerator. Besides, our evaluation results show that both the resource overhead of virtualized CNN and RNN accelerators scales linearly with the computation parallelism, meaning that our proposed virtualized DNN designs

Table 2. Compilation and Context Switching Cost (ms) with the Number of Re-allocated cores as 1, 2, 4, 8, and 16 on CNN and RNN Models

| DNN Models | Static Compilation | Dynamic Compilation | $T\_transfer$ | Context Switch Cost |
|---|---|---|---|---|
| VGG16 | 47,528.1 | 0.40–0.65 | 0.05–0.18 | 0.45–0.83 |
| ResNet50 | 50,458.9 | 0.86–1.06 | 0.03–0.15 | 0.89–1.21 |
| Inception v3 | 36,731.7 | 1.06–1.50 | 0.06–0.20 | 1.12–1.70 |
| MobileNet | 16,006.1 | 0.53–0.67 | 0.03–0.15 | 0.56–0.82 |
| LSTM_100 | 1,202.6 | 0.16–0.64 | 0.01–0.02 | 0.17–0.66 |
| LSTM_800 | 3,909.8 | 0.16–0.60 | 0.02–0.04 | 0.18–0.64 |
| LSTM_1500 | 10,181.4 | 0.17–0.57 | 0.03–0.06 | 0.20–0.63 |

can easily scale up to a larger design to maximize the resource utilization on different FPGAs. Besides, the resource bottleneck of the virtualized RNN accelerator mainly lies in memory for dense LSTM networks. However, we can utilize compression techniques [20] to prune the dense LSTM network into the sparse one, so as to achieve a balance between memory and computing resources. Moreover, our proposed virtualization methodology is applicable to both the dense and sparse RNN accelerators. The virtualized RNN accelerator design for dense LSTM networks is a more general case, and we can easily migrate it to sparse LSTM networks.

*7.2.2 Context Switching Cost Analysis.* As shown in Table 2, it takes the static compiler 16.0–50.5 s to generate the tiling-based IFP during the offline deployment for the four CNN models, while the dynamic compilation cost is only 0.4–1.5 ms during online reconfiguration. Considering the overhead of transferring the instruction files to the CNN accelerator, the online reconfiguration overhead is limited to 0.45–1.70 ms for the virtualized CNN accelerators. As for RNN models, the context switching cost of the virtualized RNN accelerator is 0.162–0.643 ms, which is much lower than that of the virtualized CNN accelerator. This is because the instruction allocator of the dynamic compiler for RNN is more straightforward than the CNN virtualized one. On the other hand, the number of tiling-based IFPs for CNN models is much fewer than RNN models, which can be deduced from the reduction of static compilation time for RNN models. These two factors together reduce the dynamic compilation time for RNN models. Besides, the size of final instruction files and context information for RNN models are both smaller than that of CNN models, leading to less transmission time for context switching. What's more, since our software compiler design is implemented in Python, it is possible to further reduce the dynamic compilation cost by implementing it using C++ and multi-thread optimization methods. In all, the experimental results demonstrate that our proposed two-stage compilation process can significantly reduce the compilation overhead for offline deployment to less than 1 minute, even with the dynamic compiler employed at the offline stage. Since our proposed method can fully decouple the hardware-dependent and -independent compilation processes, the dynamic compilation related to the lightweight resource allocation can be deployed during runtime with ~1 ms overhead, which is impossible for a traditional one-stage compilation process due to the excessive overhead.

### 7.3 Performance Evaluation

*7.3.1 Evaluation on Isolation.* We compare our virtualized DNN accelerator design on VU9P FPGA with virtualized Tesla V100 GPU for the performance isolation evaluation. We set the

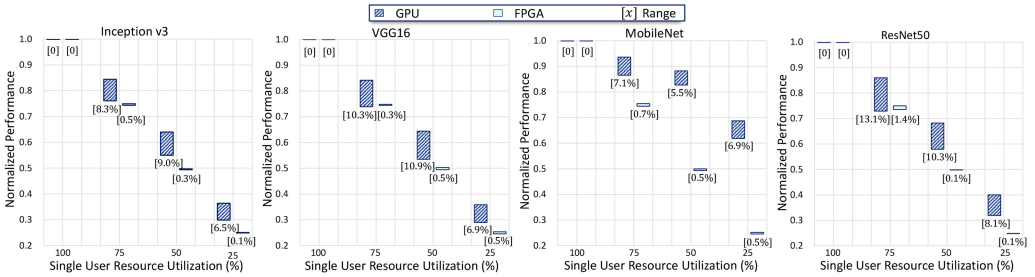


Fig. 12. Performance isolation evaluation on CNN models of our FPGA virtualization design and GPU virtualization design with different hardware resources under multi-tenant scenarios.

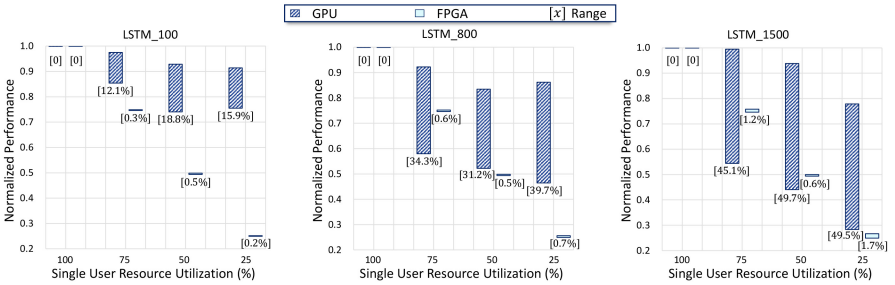


Fig. 13. Performance isolation evaluation on RNN models of our FPGA virtualization design and GPU virtualization design with different hardware resources under multi-tenant scenarios.

maximum number of users to four and give one user fixed resources $x$ (100%, 75%, 50%, and 25%), and then adjust the remaining users to occupy $1 - x$ resources in different proportions. We can get the maximum and minimum performance of a user with different fixed resources $x$.

Figure 12 shows the performance isolation evaluation on CNN models. When a user monopolizes all resources, there is no performance deviation. When the resources occupied by a single user are 75%, 50%, and 25%, GPU virtualization solution has performance deviations of 7.1%–13.1%, 5.5%–10.9%, and 6.5%–8.1%, while our FPGA virtualization design limits the performance deviation within 1%. This shows that our FPGA virtualization solution can achieve better isolation than GPU, and it can well meet the requirements for isolation in the public cloud. A single user running MobileNet on GPU will get a non-linear performance due to the fact that a small DNN model such as MobileNet cannot fully utilize the computing resources of Tesla V100 GPU.

Although NVIDIA's MPS supports computing resources partitioning, it does not provide any capability to resolve inter-application conflicts within the shared memory hierarchy such as shared cache and DRAM [23]. Resource partitioning alone is not sufficient for performance isolation, applications running on the same GPU still experience DRAM bus conflicts and **Miss Status Holding Registers** (**MSHR**) contention in the shared cache, leading to performance deviation in the multi-tenant scenario, as shown in Figure 12.

Figure 13 shows the performance isolation evaluation on RNN models. The GPU virtualization solution's performance deviation on the LSTM models is further deteriorated due to multi-tenant sharing. When the hidden size of LSTM models is 100, 800, and 1,500, the performance deviation of GPU virtualization solution under 25%–75% single-user resource utilization is 12.1%–18.8%, 31.2%–39.7%, and 45.1%–49.7%, respectively. This performance deviation deteriorates with the increase of LSTM model size because the existing GPU virtualization solution cannot achieve the physical

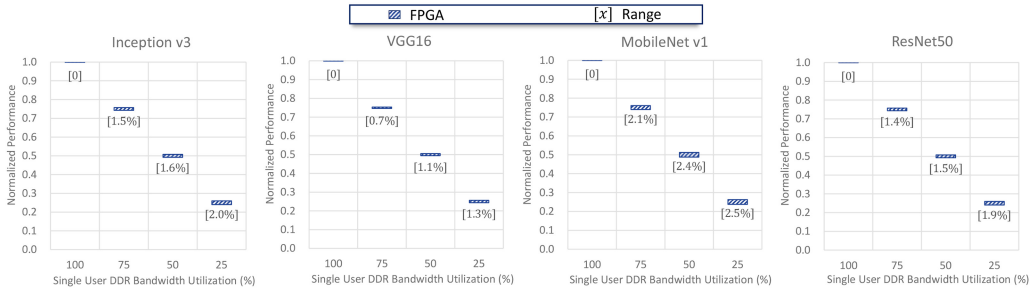


Fig. 14. Performance isolation evaluation for multiple tenants sharing one DDR memory's off-chip BW.

and performance isolation of off-chip memory access. LSTM is a typical memory-intensive DNN model with a high demand for on-chip memory and off-chip BW. When the LSTM model is small (100), GPU's on-chip memory is enough to cache most of the weights, and the competition for off-chip BW is relatively small. When the LSTM model is large (1,500), GPU's on-chip memory is insufficient, leading to the situation that every task needs to access the off-chip memory frequently. Due to the heavy competition of off-chip BW, the performance of multi-tenant scenarios deviates greatly. While our FPGA virtualization solution provides isolated hardware resources of both the computation cores and off-chip BW, achieving a negligible performance deviation of 0.2%–1.7% for RNN models.

We evaluate our virtualization system's performance isolation where multiple users share one DDR memory's off-chip BW. Figure 14 shows the performance deviation when a single user occupies 25%–75% of the DDR BW for four CNN tasks. For four tasks under different BW utilization, the performance deviation ranges from 0.7%–2.5%. The low-performance deviation of our FPGA virtualization system under off-chip BW sharing is attributed to the fair bus arbiter in MPMC.

*7.3.2 Evaluation on Single-Task Throughput of Virtualized CNN Accelerator.* Figure 15 and Table 3 show the virtualized CNN accelerator with width-only tiling has better throughput when there are fewer cores, while the output-channel-only tiling is better with more cores. These results indicate that simply applying one tiling dimension for multi-core sharing is hard to meet the performance requirement under different multi-tenant workloads, as different CNN layers and number of cores have different preferences on the tiling methods to support multi-core sharing. Both of these two tiling methods result in a large performance loss of 19.95% and 30.26% on average compared to the single-core baseline design on ResNet50. By considering the impact of both tiling methods, we can obtain optimized multi-core instructions with only 1.12% performance loss on average. We get similar results on inception v3 and VGG16 with an average performance loss of 0.95% and 3.93%, respectively. It proves that our proposed tiling-based IFPs and workload-balanced allocator are effective for the multi-core sharing of the virtualized CNN accelerator. The dynamic compiler is able to find the optimal tiling choice of each layer during runtime reconfiguration, and generates instruction files with minimal performance loss.

It should be noted that the average performance loss of the optimized multi-core design on MobileNet is up to 31.64%, as shown in Figure 15. The reason behind this is that MobileNet is a typical small CNN model, and the ratio of its parameter amount to the computation amount is significantly larger than the other three CNN models, so its demand for memory BW is much greater. Figure 16(3) shows that *Load* operations take up 21% of the total execution time and are mainly dependent on the *Conv* operations in MobileNet, meaning that MobileNet is memory BW limited under current hardware configurations. We verify the correctness of our hypothesis

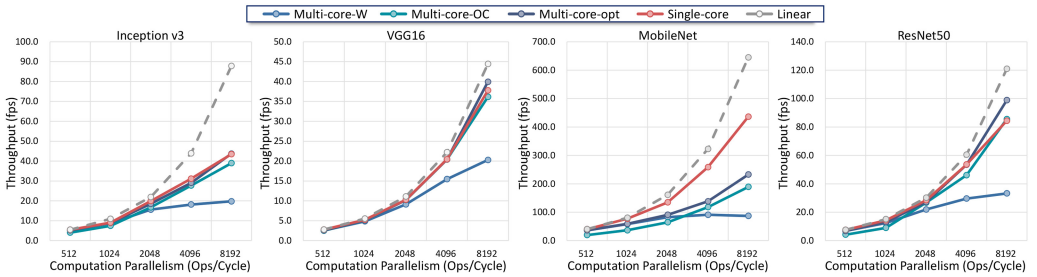


Fig. 15. The single-task throughput of the virtualized CNN accelerator with three different tiling strategies (W: width-only, OC: output-channel-only, opt: optimized) and the single-core baseline design.

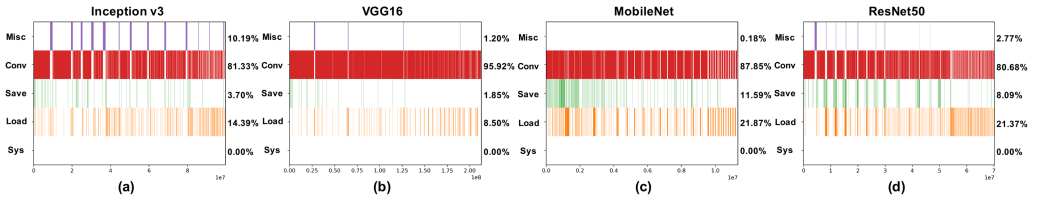


Fig. 16. The proportion of execution time of different types of instructions when running four CNN models on a single core with the computation parallelism of 512. These results are profiled on Xilinx U200 FPGA.

through simulation experiments by doubling the memory BW of multi-core and single-core designs. The simulation results show that the average performance loss of the optimized multi-core design on MobileNet is reduced to 5.33%, which further demonstrates that current hardware configuration has redundant computing resources and insufficient memory BW for MobileNet.

Figure 16 illustrates the proportion of execution time of different types of instructions when running four CNN networks on the virtualized CNN accelerator, which can give us some insights into the non-linearity of CNN's single-task performance. We take VGG16 and Inceptionv3 as examples, which are the two CNN models with the smallest non-linearity and the largest non-linearity, respectively. As shown in Figure 16(1), *Conv* and *Misc* operations take up 81% and 10% of the total execution time, respectively. Besides, the computation flow of *Conv* operations in Inceptionv3 mainly depends on *Misc* operations, since the structure of Inceptionv3 is the most complex and has the most branches among the four CNN models. Thus, even though increasing the computing parallelism can reduce the *Conv* operations' execution time, the stalling caused by the dependence of *Conv* operations on the non-convolution operations *Misc* will be magnified, and the non-linearity will increase with the parallelism. This is because *Misc* operations have only two dimensions of parallelism to accelerate, while *Conv* operations have three dimensions, as discussed in Section 2.3. As shown in c of VGG16, the execution time of *Conv* operations is dominant (96%) and is almost unaffected by other instructions. When the parallelism increases, the speedup ratio obtained by the *Conv* operations can be better reflected in the whole computing process, resulting in a nearly linear performance improvement.

*7.3.3 Evaluation on Single-Task Throughput of Virtualized RNN Accelerator.* Figure 17(a) shows that virtualized RNN accelerator introduces no single-task performance loss compared to the static single-core baseline design. It proves that our previous analysis of row-tiling IFPs of MVM operations is correct. That is, the data reuse and tiling dimension of MVM operations correspond to each other. As shown in Figures 17(a-2) and (a-3), there is almost no non-linearity when the hidden size

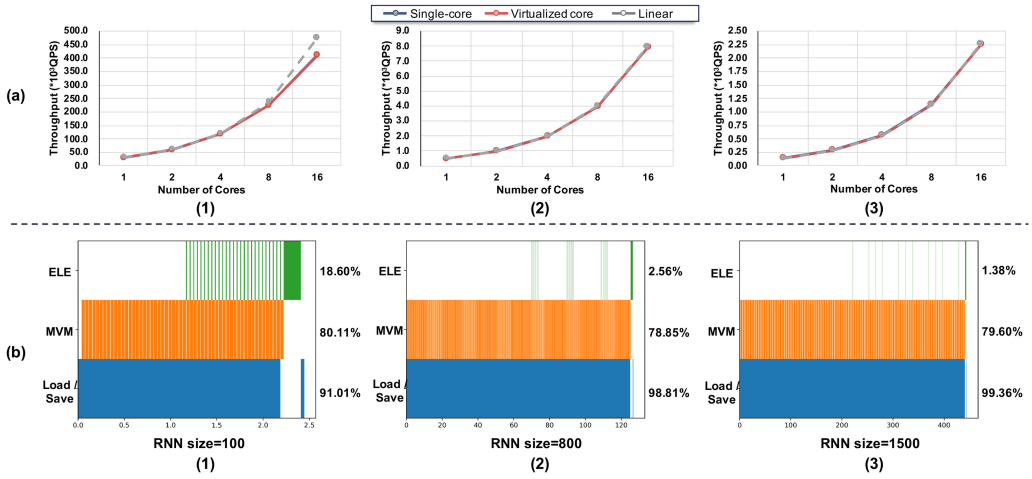


Fig. 17. (a) Single-task performance (**Queries Per Seconds** (**QPS**)) of the virtualized RNN accelerator and the single-core baseline design on the single-layer LSTM model with the hidden size of 100, 800, and 1500. (b) The execution time of different instructions when running on the virtualized multi-core design ($16 \times 16$).

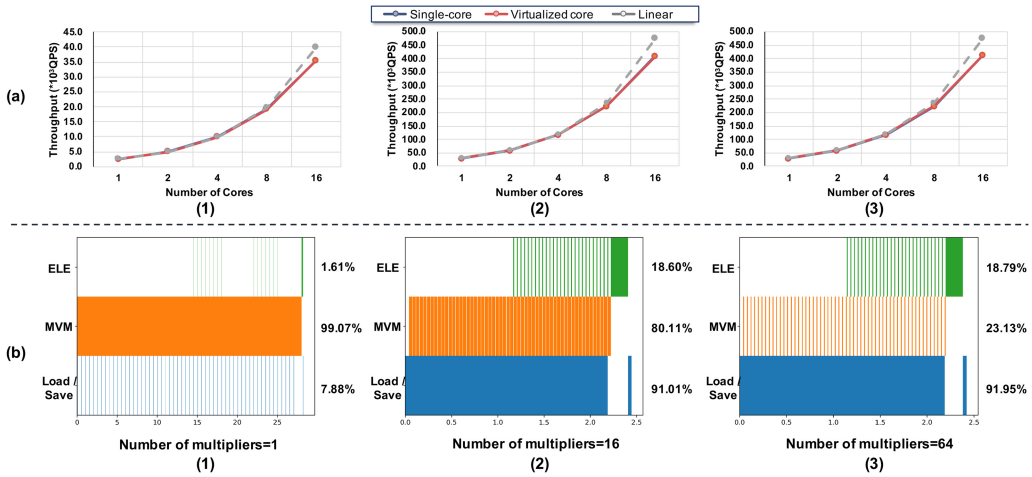


Fig. 18. (a) Single-task performance (QPS) of the virtualized RNN accelerator and the single-core baseline design on the single-layer LSTM model with the hidden size of 100 (b) and the execution time of different instructions when running on the virtualized multi-core design, with the number of multipliers in each core as 1, 16, and 64.

is 800 and 1,500. It can be seen from Figures 17(b-2) and (b-3) that both cases are entirely dominated by data movements (98.8% and 99.4%). The non-linearity in the case with the hidden size as 100 comes from the fact that ELE operations take up 18.6% of the total execution time, resulting in the time of data movements down to 91.0%, as shown in Figure 17(1).

To evaluate the differences between the computation and memory access of the virtualized RNN accelerator under different hardware configurations, we set the number of multipliers of each core as 1, 16, and 64. The three examples illustrated in Figure 18 correspond to three cases: computation dominated, computation and memory access balanced, and memory access dominated. The

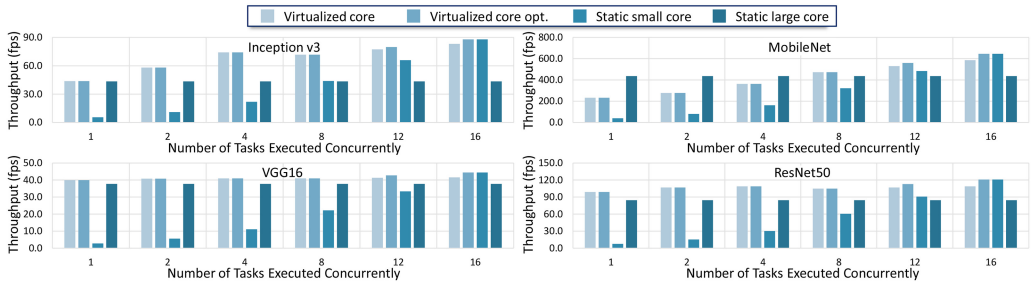


Fig. 19. The multi-task throughput of the virtualized multi-core CNN designs, the static multi-core design, and the single-core baseline design under different workload. (opt: optimization for the single task.)

Table 3. Single-Task Performance on ResNet50

| | Throughput (fps) | | | | | Average |
|---|---|---|---|---|---|---|
| Parallelism | 512 | 2 × 512 | 4 × 512 | 8 × 512 | 16 × 512 | Loss |
| multi-core-W | 6.8 | 12.4 | 21.9 | 29.6 | 33.3 | 30.26% |
| multi-core-OC | 4.2 | 9.0 | 26.8 | 46.1 | 85.5 | 19.95% |
| multi-core-opt | 6.8 | 13.1 | 27.2 | 53.5 | 98.9 | 1.12% |
| single-core | 7.6 | 14.3 | 28.5 | 53.6 | 84.4 | 0 |
| linear | 7.6 | 15.1 | 30.2 | 60.5 | 120.9 | \ |

(W: width-only, OC:output-channel-only, opt: optimized).

non-linearity of the first case is small, while that of the second two cases is large. Besides, the proportion of ELE operations in the three cases are consistent with the magnitude of non-linearity. It means that the non-linearity is related to the ratio of different operations and associated with the data dependence between operations.

*7.3.4 Evaluation on Multi-Task Throughput.* We evaluate the throughput of the virtualized multi-core design, static multi-core design, and static single-core design in the private cloud scenario by changing the number of concurrent tasks. As shown in Figure 19, the left half of each CNN model is a low-workload situation, where static multi-core design cannot fully utilize FPGA resources, resulting in poor performance. The right half is a high-workload situation, where a static single-core design has poor performance due to non-linearity, as shown in Figure 15. While our virtualized multi-core CNN design can achieve optimal performance in any situation.

For the case of 16 concurrent tasks, each small core will run a task independently. Since our proposed compilation process introduces additional overhead for the single-core performance, the throughput of virtualized multi-core design is lower than that of the static multi-core design, as shown in Table 3. To solve this problem, we can use the original compiler to generate the single-core instruction files during the offline deployment, while the dynamic compiler only generates tiling-based instructions for the tasks with more than one core allocated. Through this optimization, we can ensure the optimal performance in the high-workload situation. The experimental results show that our virtualized multi-core CNN accelerator can achieve 1.07–1.69× and 1.88–3.12× throughput improvement over the static single-core design and static multi-core design, respectively.

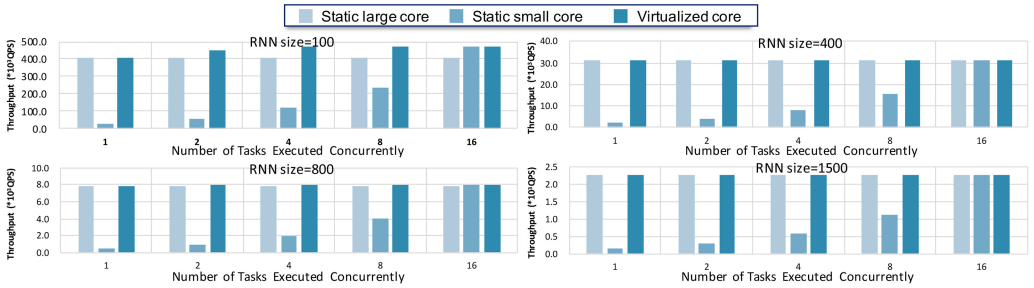


Fig. 20. The multi-task throughput (QPS) of the virtualized multi-core RNN designs, the static multi-core design, and the single-core baseline design under different workload.

As for RNN accelerators, Figure 20 shows that our virtualized RNN design can achieve 5.67–6.18× multi-task performance improvement over the static multi-core design. This performance improvement is mainly due to our virtualized designs' adaptability to the dynamic workload in multi-tenant scenarios, where our virtualized design can make full use of the computing resources. In contrast, the static multi-core design will lead to the waste of computing resources when the number of tasks is not sufficient. Compared with the static single-core design, our virtualized RNN design only improves multi-task performance by 1.01–1.11×. This is because when the size of the hidden dimension exceeds 400, the total off-chip BW of the FPGA board can not meet the demand, and the execution time of the RNN accelerator is almost completely occupied by memory access. As previously analyzed, this leads to near-linear single-task performance. Thus, in this case, the static single large core with TDM sharing can also achieve an excellent multi-task performance. Even so, our virtualized multi-core RNN accelerator design can maintain a good hardware resource isolation property for multi-tenant security, which is not available for the static single-core design.

## 8 CONCLUSION AND FUTURE WORK

In this article, we propose a unified virtualization framework for general-purpose DNN applications in the cloud, enabling multi-tenant sharing for both the CNN and RNN accelerators on a single FPGA. The ISA-based virtualized multi-core design enables that hardware resources can be dynamically reconfigured with negligible performance loss for the single-task applications. The proposed two-stage static-dynamic compilation and tiling-based instruction frame package techniques make it possible for fast dynamic reconfiguration at the software level, with the context switching cost limited to ~1 ms. According to the experimental results, our virtualization designs achieve up to 3.12× and 6.18× higher throughput over the static CNN and RNN baseline design, respectively.

The exploration of our proposed methods on the system multi-node level will be exciting, which we leave as future work. Besides, some commonly used optimization techniques, such as layer fusion and sparse models, are not discussed in this article since they are fully compatible with our proposed virtualization methods with lower compilation complexity. We will explore an ISA-based multi-FPGA virtualization system with a more general DNN inference case in our future work.

## REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX, 265–283.

[2] Mohamed S. Abdelfattah, David Han, Andrew Bitar, Roberto DiCecco, Shane O'Connell, Nitika Shanker, Joseph Chu, Ian Prins, Joshua Fender, Andrew C. Ling, and Gordon R. Chiu 2018. DLA: Compiler and FPGA overlay for neural network inference acceleration. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. IEEE, 411–418.

[3] Alibaba. 2019. Alibaba F1. Retrieved December 20, 2020 from https://www.aliyun.com/product/ecs/fpga?spm=5176.224200.100.29.813f6ed6OuUlZ2&aly_as=x0_o5Br.

[4] Amazon. 2019. AWS F1. Retrieved December 20, 2020 from https://aws.amazon.com/ec2/instance-types/f1/.

[5] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, Jie Chen, Jingdong Chen, Zhijie Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Ke Ding, Niandong Du, Erich Elsen, Jesse Engel, Weiwei Fang, Linxi Fan, Christopher Fougner, Liang Gao, Caixia Gong, Awni Hannun, Tony Han, Lappi Johannes, Bing Jiang, Cai Ju, Billy Jun, Patrick LeGresley, Libby Lin, Junjie Liu, Yang Liu, Weigao Li, Xiangang Li, Dongpeng Ma, Sharan Narang, Andrew Ng, Sherjil Ozair, Yiping Peng, Ryan Prenger, Sheng Qian, Zongfeng Quan, Jonathan Raiman, Vinay Rao, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Kavya Srinet, Anuroop Sriram, Haiyuan Tang, Liliang Tang, Chong Wang, Jidong Wang, Kaifu Wang, Yi Wang, Zhijian Wang, Zhiqian Wang, Shuang Wu, Likai Wei, Bo Xiao, Wen Xie, Yan Xie, Dani Yogatama, Bin Yuan, Jun Zhan, Zhenyao Zhu. 2016. Deep speech 2: End-to-end speech recognition in english and mandarin. In *Proceedings of the International Conference on Machine Learning*. 173–182.

[6] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. 2017. An openCL™ deep learning accelerator on arria 10. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 55–64.

[7] Alexander Brant and Guy G. F. Lemieux. 2012. ZUMA: An open FPGA overlay architecture. In *Proceedings of the Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 93–96.

[8] Vinayak Gokhale, Aliasger Zaidy, Andre Xian Ming Chang, and Eugenio Culurciello. 2017. Snowflake: An efficient hardware accelerator for convolutional neural networks. In *IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–4.

[9] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. 2014. Enabling FPGAs in the cloud. In *Proceedings of the ACM Conference on Computing Frontiers*. ACM, 3.

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. USENIX, 578–594.

[11] Yao Chen, Jiong He, Xiaofan Zhang, Cong Hao, and Deming Chen. 2019. Cloud-DNN: An open framework for mapping DNN models to cloud FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 73–82.

[12] Yujeong Choi and Minsoo Rhu. 2020. Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture*. IEEE, 220–233.

[13] Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou. 2014. A fully pipelined and dynamically composable architecture of CGRA. In *Proceedings of the Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 9–16.

[14] Guohao Dai, Yi Shan, Fei Chen, Yu Wang, Kun Wang, and Huazhong Yang. 2014. Online scheduling for fpga computation in the cloud. In *Proceedings of the International Conference on Field-Programmable Technology*. IEEE, 330–333.

[15] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure accelerated networking: SmartNICs in the public cloud. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*. USENIX, 51–66.

[16] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A configurable cloud-scale DNN processor for real-time AI. In *Proceedings of the Annual International Symposium on Computer Architecture*. IEEE, 1–14.

[17] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. 2017. Angel-eye: A complete design flow for mapping CNN onto embedded FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 1 (2017), 35–47.

[18] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. 2019. A survey of FPGA-based neural network inference accelerators. *ACM Transactions on Reconfigurable Technology and Systems* 12, 1 (2019), 2.

[19] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. 2020. Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. In *Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*. IEEE, 982–995.

[20] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William J. Dally. 2017. ESE: Efficient speech recognition engine with sparse LSTM on fpga. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 75–84.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 770–778.

[22] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv:1704.04861. Retrieved from https://arxiv.org/abs/1704.04861.

[23] Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan Rajkumar. 2019. Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs. In *Proceedings of the 2019 IEEE Real-Time and Embedded Technology and Applications Symposium*. 29–41. DOI: https://doi.org/10.1109/RTAS.2019.00011

[24] Andy Jassy. 2018. AWS re:Invent 2018. Retrieved December 20, 2020 from https://www.youtube.com/watch?v=ZOIkOnW640A.

[25] Zhe Jiang, Neil Audsley, Pan Dong, Nan Guan, Xiaotian Dai, and Lifeng Wei. 2019. MCS-IOV: Real-time I/O virtualization for mixed-criticality systems. In *Proceedings of the 2019 IEEE Real-Time Systems Symposium*. 326–338. DOI: https://doi.org/10.1109/RTSS46320.2019.00037

[26] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. 2018. Sharing, protection, and compatibility for reconfigurable fabric with amorphos. In *Proceedings of the 13th {USENIX} Symposium on Operating Systems Design and Implementation*. 107–127.

[27] Oliver Knodel, Paul R. Genssler, and Rainer G. Spallek. 2017. Virtualizing reconfigurable hardware to provide scalability in cloud architectures. In *International Conference on Advances in Circuits, Electronics and Micro-electronics (CENICS)*. 33–38.

[28] Ondrej Kotaba, Jan Nowotsch, Michael Paulitsch, Stefan M. Petters, and Henrik Theiling. 2013. Multicore in real-time systems–temporal isolation challenges due to shared resources. In *Proceedings of the 16th Design, Automation & Test in Europe Conference and Exhibition*.

[29] Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, Gu-Yeon Wei, and Carole-Jean Wu. 2020. MLPerf: An industry standard benchmark suite for machine learning performance. *IEEE Micro* 40, 2 (2020), 8–16.

[30] Eriko Nurvitadhi, Dongup Kwon, Ali Jafari, Andrew Boutros, Jaewoong Sim, Phillip Tomson, Huseyin Sumbul, Gregory Chen, Phil Knag, Raghavan Kumar, Ram Krishnamurthy, Sergey Gribok, Bogdan Pasca, Martin Langhammer, Debbie Marr, and Aravind Dasu. 2019. Why compete when you can work together: Fpga-asic integration for persistent rnns. In *Proceedings of the 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 199–207.

[31] NVIDIA. 2019. CUDA multi process service overview. Retrieved December 20, 2020 from https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[32] Nvidia. 2020. Nvidia A100 tensor core GPU architecture. Retrieved December 20, 2020 from https://www.nvidia.cn/data-center/a100/.

[33] Jian Ouyang. 2017. XPU: A programmable FPGA accelerator for diverse workloads. In *Proceedings of the IEEE Hot Chips Symposium*.

[34] Jian Ouyang, Wei Qi, Wang Yong, Yichen Tu, Jing Wang, and Bowen Jia. 2016. SDA: Software-defined accelerator for general-purpose distributed big data analysis system. In *Proceedings of the IEEE Hot Chips Symposium*. IEEE.

[35] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, Juan Pino, Martin Schatz, Alexander Sidorov, Viswanath Sivakumar, Andrew Tulloch, Xiaodong Wang, Yiming Wu, Hector Yuen, Utku Diril, Dmytro Dzhulgakov, Kim Hazelwood, Bill Jia, Yangqing Jia, Lin Qiao, Vijay Rao, Nadav Rotem, Sungjoo Yoo, and Mikhail Smelyanskiy. 2018. Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications. arXiv:1811.09886. Retrieved from https://arxiv.org/abs/1811.09886.

[36] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. *SIGARCH Computer Architecture News* 42, 3 (2014), 13–24.

[37] Francesco Restuccia, Alessandro Biondi, Mauro Marinoni, Giorgiomaria Cicero, and Giorgio Buttazzo. 2020. AXI hyperconnect: A predictable, hypervisor-level interconnect for hardware accelerators in FPGA SoC. In *Proceedings of the 57th ACM/IEEE Design Automation Conference*. 1–6. DOI: https://doi.org/10.1109/DAC18072.2020.9218652

[38] Vladimir Rybalkin, Alessandro Pappalardo, Muhammad Mohsin Ghaffar, Giulio Gambardella, Norbert Wehn, and Michaela Blott. 2018. FINN-L: Library extensions and design tradeoff analysis for variable precision LSTM networks on FPGAs. In *Proceedings of the 28th International Conference on Field Programmable Logic and Applications*. IEEE, 89–897.

[39] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. arXiv:1409.1556. Retrieved from https://arxiv.org/abs/1409.1556.

[40] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2818–2826.

[41] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. 2018. A survey on FPGA virtualization. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. IEEE, 131–138.

[42] Xilinx. 2018. Accelerating DNNs with Xilinx Alveo Accelerator Cards. Retrieved December 20, 2020 from https://www.xilinx.com/applications/megatrends/machine-learning.html.

[43] Xilinx. 2019. AXI Interconnect IP. Retrieved from https://www.xilinx.com/products/intellectual-property/axi_interconnect.html.

[44] Yu Xing, Shuang Liang, Lingzhi Sui, Xijie Jia, Jiantao Qiu, Xin Liu, Yushun Wang, Yi Shan, and Yu Wang. 2019. DNNVM: End-to-end compiler leveraging heterogeneous optimizations on FPGA-based CNN accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2019), 2668–2681.

[45] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2018. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 11 (2018), 2072–2085.

[46] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2018. DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs. In *Proceedings of the International Conference on Computer-Aided Design*. ACM, 56.

Received January 2021; revised May 2021; accepted August 2021