Virtualizing FPGAs in the Cloud

Yue Zha
University of Pennsylvania
Philadelphia, PA
zhayue@seas.upenn.edu

Jing Li
University of Pennsylvania
Philadelphia, PA
janeli@seas.upenn.edu

Abstract

Field-Programmable Gate Arrays (FPGAs) have been integrated into the cloud infrastructure to enhance its computing performance by supporting on-demand acceleration. However, system support for FPGAs in the context of the cloud environment is still in its infancy with two major limitations, i.e., the inefficient runtime management due to the tight coupling between compilation and resource allocation, and the high programming complexity when exploiting scale-out acceleration. The root cause is that FPGA resources are *not virtualized*.

In this paper, we propose a full-stack solution, namely ViTAL, to address the aforementioned limitations by virtualizing FPGA resources. Specifically, ViTAL provides a homogeneous abstraction to decouple the compilation and resource allocation. Applications are offline compiled onto the abstraction, while the resource allocation is dynamically determined at runtime. Enabled by a latency-insensitive communication interface, applications can be mapped flexibly onto either one FPGA or multiple FPGAs to maximize the resource utilization and the aggregated system throughput. Meanwhile, ViTAL creates an illusion of a *single* and *large* FPGA to users, thereby reducing the programming complexity and supporting scale-out acceleration. Moreover, ViTAL also provides virtualization support for peripheral components (e.g., on-board DRAM and Ethernet), as well as protection and isolation support to ensure a secure execution in the multi-user cloud environment.

We evaluate ViTAL on a real system—an FPGA cluster composed of the latest Xilinx UltraScale+ FPGAs (XCVU37P). The results show that, compared with the existing management method, ViTAL enables fine-grained resource sharing and reduces the response time by 82% on average (improving Quality-of-Service) with a marginal virtualization overhead. Moreover, ViTAL also reduces the response time by 25% compared to AmorphOS (operating in high-throughput mode), a recently proposed FPGA virtualization method.

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7102-5/20/03...$15.00
https://doi.org/10.1145/3373376.3378491

ACM Reference Format:
Yue Zha and Jing Li. 2020. Virtualizing FPGAs in the Cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3373376.3378491

1 Introduction

Integrating hardware accelerators into cloud infrastructures is a major trend to enhance the cloud computing performance. Among the available options, Field-Programmable Gate Arrays (FPGAs) become attractive due to their ability to achieve high throughput and predictable latency while providing low power, time-to-value and flexibility of accelerating diverse applications, from machine learning [12][57][70], data analysis [42][26][33] to graph processing [18][19][21]. As such, in recent years, FPGAs have begun to be deployed in commercial cloud platforms (e.g., Amazon AWS [1] and Microsoft Azure [49]) to support on-demand acceleration.

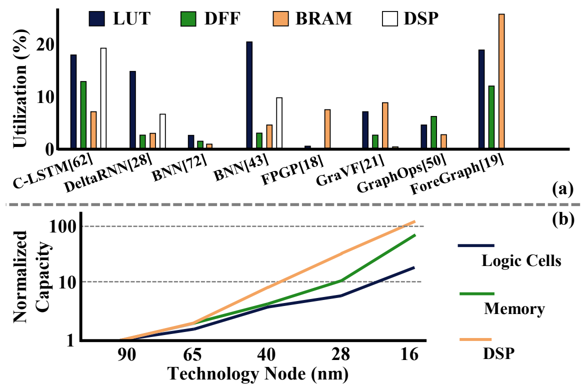

Figure 1. (a) The amount of resource used by several representative FPGA applications. The results are normalized to the capacity of Xilinx VU13P FPGA. (b) The FPGA capacity keeps growing due to technology advances.

Despite decades of research on FPGAs in embedded systems, the system support for FPGAs in the context of the cloud computing environment is still in its infancy and has two major limitations. The *first* limitation is the inefficient utilization of the programmable resources provided by the FPGA cluster. One simple strategy currently adopted by cloud vendors (e.g., Amazon AWS [1]) is to manage the pool of FPGA resources at a per-device granularity, i.e., allocating one physical FPGA device *exhaustively* to one application (Fig. 2a). Since the increasingly diverse applications/services that are deployed in cloud use different amounts of resources (Fig. 1a), such coarse-grained management method may result in

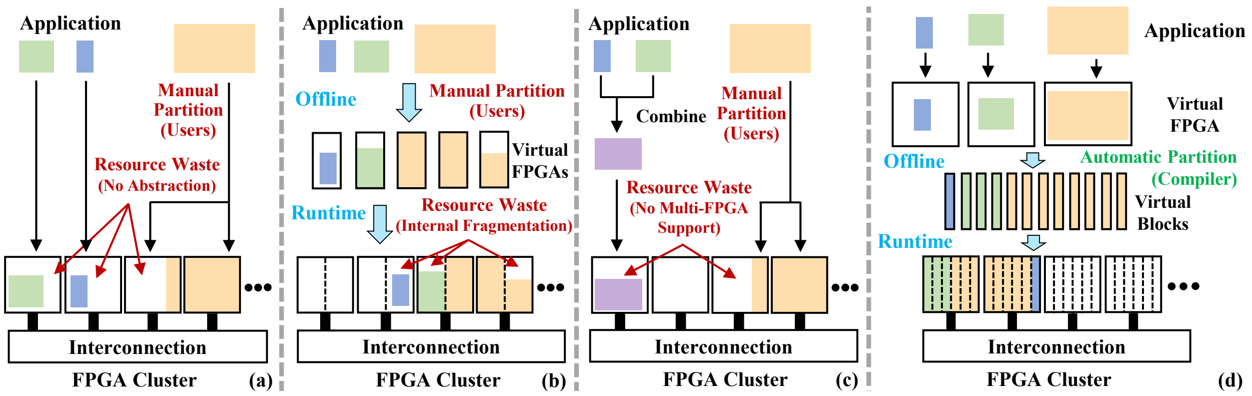

Figure 2. (a) A conceptual diagram to illustrate the management method used in existing FPGA clouds, which has a low resource utilization due to the lack of abstraction. (b) Several works [6][11] (including AmorphOS [35] when operating in low-latency mode) propose a slot-based method to manage FPGA cluster at a sub-FPGA granularity. They improve the resource utilization over the existing method, but the improvement can be limited due to the internal fragmentation issue. (c) The high-throughput mode of AmorphOS [35] achieves better FPGA sharing by combining multiple applications during offline compilation, but may still not be able to make full use of resources due to the lack of multi-FPGA support. (d) ViTAL provides a homogeneous abstraction that comprises identical virtual blocks to enable a fine-grained FPGA sharing and scale-out acceleration.

internal fragmentation, i.e., applications cannot fully utilize the allocated resources. This fragmentation issue is further exacerbated by the growing capacity of FPGAs (Fig. 1b) due to the potentially low device utilization.

The *second* limitation is the high programming complexity when exploiting scale-out acceleration using multiple FPGAs. As commercial FPGA compilation tools (e.g., Vivado [23]) only support application development on a single FPGA device, scaling out applications to multiple FPGAs is non-trivial. Due to the lack of system support, it requires users to manually partition applications across the physical FPGA boundary, with extremely careful handling of low-level hardware details such as inter-FPGA communication bandwidth, network topology, etc. to ensure high performance, functional correctness and high utilization. Such process is tedious and labor-intensive, greatly increasing the programmers' burden.

The root cause of these two limitations is that the FPGA resources are *not virtualized*. Virtualizing FPGAs is a more challenging task compared to CPUs/GPUs due to the fundamental difference between their architecture and compute models, i.e., an FPGA application describes the physical hardware circuits wired together under *spatial* resource constraints, while a CPU/GPU application is a sequence of pre-defined instructions executing in the *temporal* domain. As such, the virtualization techniques developed for CPU/GPU-based clouds cannot be trivially applied to FPGA-based clouds.

Virtualizing FPGAs faces a major challenge, i.e., a dilemma between runtime overhead and resource utilization due to the tight coupling between compilation and resource allocation. Unlike CPUs, the compilation process for FPGAs is typically long, which may take hours or even days depending on the complexity of applications, and applications need to be recompiled in case of any physical resource changes (either capacity or location) at runtime. While runtime re-compilation may help to improve the resource utilization, the time-consuming FPGA compilation incurs a prohibitive runtime overhead and limits physical resource allocation to the offline compile time. Such dilemma, together with the inability to dynamically respond to actual load and resource availability in elastic clouds leads to an inefficient utilization of FPGA resources.

There are some initial research efforts in virtualizing FPGAs for cloud (Table 1). Despite showing good promise, they only *partially* address these limitations. For instance, the overlay architecture [5][8][32] reduces the programming complexity by providing a high-level application-dependent abstracted architecture. It helps hide hardware details and improve programming productivity but is only applicable to a single FPGA and does not support a multi-user cloud computing environment. Several frameworks [53][19][29] have been proposed to support multi-FPGA computing, but they only support static resource allocation at compile time and have limited flexibility for runtime resource management, thus, they do not address the *first* limitation. Several works [63][11][6][38][22] propose a slot-based method to improve the resource utilization by managing FPGA clusters at a sub-FPGA granularity as shown in Fig. 2b. However, such improvement can be limited due to internal fragmentation. These works do not provide system support for scale-out acceleration and thus do not address the *second* limitation. The recent work AmorphOS [35] achieves better FPGA sharing than prior methods by providing two operating modes: low-latency mode and high-throughput mode. The low-latency mode employs the same slot-based method (Fig. 2b) and thus has the same limitations. The high-throughput mode of AmorphOS combines multiple applications into a single application, and statically compiles it onto a single physical FPGA (Fig. 2c). While significantly improving the per-device resource utilization, this mode may still not be able to make full use of FPGA resources due to the lack of multi-FPGA support. Moreover, this mode neither decouples the compilation and resource allocation. Thus, it

Table 1. Existing methods are not done in the context of the cloud environment or cannot fully address the aforementioned limitations. On the contrary, ViTAL efficiently virtualizes FPGAs in the cloud environment and fully addresses these limitations.

Method	FPGA Sharing	Resource Utilization	Scale-out Acceleration	Virtualization Overhead
Overlay Architecture [5][8][32]	Not Support	Medium	No Support	Low
Slot-based method ∓ [11][63][38]	Support	Medium	No Support	Low
Multi-FPGA Framework [53][19][29]	No Support	Medium	Support	Low
AmorphOS (High-Throughput Mode) [35]	Support	High	No Support	High
ViTAL	Support	High	Support	Low

∓ Including AmorphOS operating in the low-latency mode.

needs to offline compile *all* possible combinations of applications to avoid runtime overhead, and also needs to recompile all related combinations when one application changes. Detailed discussion is presented in Section 6.

In this paper, we propose a full-stack solution, dubbed ViTAL[1], that can effectively virtualize the FPGA clusters in the context of the cloud environment and address *both* limitations. Specifically, to address the *first* limitation, the ViTAL stack provides a new system abstraction (Architecture Layer, Section 3.2) to enable a dynamic fine-grained resource management. It abstracts the rich heterogeneous resources of FPGA clusters (i.e., logic cells, on-chip memories, DSPs, IOs and routing network) into a *homogeneous* and simplified view of an array of identical virtual blocks. As shown in Fig. 2d, each block is a partition of an FPGA with standardized resource types, capacity and interface. Compared to using an entire FPGA to implement an application, ViTAL's compilation tool partitions an application into a group of virtual blocks that can be dynamically managed at runtime. Such block-level resource management effectively alleviates the internal fragmentation issue and improves the resource utilization. As virtual blocks can be mapped onto either one FPGA or multiple FPGAs at runtime (Fig. 2d), the virtual-to-physical mapping across the physical boundary of an FPGA device may result in latency mismatch for inter-block communication or even timing failures. Such issue is addressed by the latency-insensitive inter-block interface in our abstraction. As the decision whether a group of blocks should be mapped onto the same FPGA (on-chip interconnection) or across the FPGA device boundary (chip-to-chip/node-to-node communication) cannot be resolved at the offline compile time and can only be determined at runtime, this latency-insensitive interface effectively simplifies the partition process by hiding the bandwidth/latency difference and provides an identical interface for both cases to attain high performance and functional correctness (avoiding timing failure). In addition to the virtualization of FPGA's on-chip resources, we also provide mechanisms to virtualize peripherals (e.g., on-board DRAMs and Ethernet).

To address the *second* limitation, ViTAL creates an illusion of a single and infinitely large FPGA to users (Programming Layer, Section 3.1) to reduce the programming complexity. Custom tools are developed and integrated into the commercial FPGA compilation flow (Compilation Layer, Section 3.3) to map applications onto the proposed abstraction: a group of virtual blocks. These custom tools are designed to perform two major tasks: 1) allocating a certain number of virtual blocks according to the amount of resources required by an application, and 2) transparently partitioning the application into the allocated blocks and generating the corresponding latency-insensitive communication interface. To map partitioned applications onto physical FPGAs, an FPGA is divided into three regions (Fig. 4b). The communication and service regions are reserved by the system to implement the communication interface and to provide virtualization support for peripheral devices. The user region is further divided into several identical physical blocks to implement virtual blocks. Specifically, a virtual block is offline mapped onto a physical block, and can be relocated to another physical block at runtime without recompilation.

A system controller (System Layer, Section 3.4) is included to perform the runtime resource management and expose APIs to ease system integration. The runtime management system dynamically maps applications (compiled virtual blocks) onto one FPGA or multiple FPGAs to maximize resource utilization. Note that mapping applications onto multiple FPGAs requires an inter-FPGA communication, which may potentially reduce the throughput. Therefore, we develop a communication-aware runtime management policy (Section 3.4) to balance the resource utilization and the overall system throughput. Finally, this runtime management also provides isolation support through our virtual-to-physical mapping strategy to ensure a secure execution environment.

We validate the feasibility of ViTAL with a rigorous performance evaluation on a real system. Real system prototyping using commercial FPGAs is not a trivial task since the commercial-grade FPGA architecture is more complex than the simplistic FPGA architecture used in publicly available tutorials/text books or even some of the popular open-source tools (such as VTR [47]). It introduces extra heterogeneity (e.g., clock region and multi-die package) and increases the difficulty of implementing the proposed Architecture Layer. Moreover, the implementation of the new compilation flow requires non-trivial development efforts as the proposed compilation flow is also radically different from the commercially available one from FPGA vendors. To alleviate the pressure for large-scale software development, we made the best efforts by reusing some of the proprietary tools from FPGA vendors, and developing a set of custom tools either from scratch or by leveraging a recently released open-source tool from Xilinx

[1]**ViTAL**: **Vi**rtualization s**T**ack for FPG**A**s in the c**L**oud

called RapidWright [41]. We also address the incompatibility issue of these tools to integrate them into a unified compilation flow. We summarize some of the key learnings obtained from this real system prototyping in Section 3.2 and 3.3.

In particular, we made the following major contributions:

1. We propose ViTAL, a full-stack solution that virtualizes FPGA in the cloud environment and addresses the aforementioned two limitations. Specifically, ViTAL provides a homogeneous abstraction to decouple the compilation and resource allocation, thereby enabling fine-grained FPGA sharing. Moreover, ViTAL reduces the programming complexity and supports scale-out acceleration by creating an illusion of a single and infinitely large FPGA to users.

2. We demonstrate the benefits of ViTAL on a real system—an FPGA cluster with the latest Xilinx UltraScale+ FPGAs (XCVU37P). We perform an extensive evaluation for each layer and their comprehensive interactions of the ViTAL stack on representative benchmarks. The results show that ViTAL improves the response time by 82% compared with the non-sharing baseline with a marginal virtualization overhead.

The rest of the paper is organized as follows. Section 2 provides the background information. Section 3 presents the details of each layer in the ViTAL stack, while Section 4 describes the details of the custom tools that are integrated into the commercial compilation flow (Vivado). Section 5 presents the evaluation results. Section 6 discusses the related work, followed by Section 7 to conclude the paper.

2 Background

2.1 FPGA Architecture

State-of-the-art FPGAs have an island-style heterogeneous architecture that comprises a 2D array of configurable logic blocks (CLBs), switch blocks (SBs), connection blocks (CBs) and hard IP blocks (Fig.3a). Specifically, CLBs contain several look-up tables (LUTs) and each LUT stores a truth table to implement an arbitrary 6-input (or less) logic function. SBs and CBs form an extensive bit-wise network to route the interconnections among CLBs and hard IP blocks. Hard IP blocks are included to perform specific functions, e.g., BRAM for on-chip data storage and DSP for floating-point operation.

2.2 FPGA Compilation Flow

FPGA compilation is a process that *directly* maps the data flow of applications onto the physical hardware. As shown in Fig. 3b, the compilation process can be divided into two parts. Specifically, if applications are written using high-level programming languages (e.g., OpenCL [30]), then a front-end (high-level synthesis tool) will be included to convert the applications into Verilog RTL code. This RTL code is then processed by the back-end that contains three sub-steps. In the first sub-step (parser), the Verilog code is synthesized into different levels of intermediate representation (IRs), including control data-flow graphs (CDFGs), data-flow graphs (DFGs) and a netlist of primitives (e.g., logic gates and hard IP blocks). In the second sub-step (technology mapping), the logic gates in the netlist are further mapped into appropriate-size LUTs and flip-flops. The last sub-step performs physical optimization, including clustering/packing, placement and routing. This step is time-consuming and can take up to *several hours, or even days*, since it needs to place up to *millions of* primitives onto the physical hardware and route the numerous interconnections between them. The runtime of this sub-step is not expected to be substantially reduced since it is a *NP-Complete* problem with a limited parallelism [66].

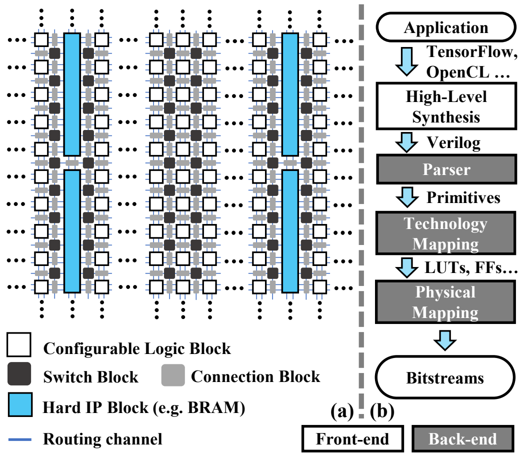

Figure 3. (a) A conceptual diagram of a typical FPGA architecture. Note that the commercial-grade FPGA architecture introduces additional features that are not drawn in this diagram for simplicity. (b) A typical FPGA compilation flow.

3 The ViTAL Stack

The ViTAL stack comprises *four* layers to virtualize the FPGA resources and support the scale-out acceleration. This section presents the details of each layer.

3.1 Programming Layer

ViTAL supports various programming languages, such as C/C++, OpenCL and other domain-specific languages [52][48] [13], as the compilation tool (Section 3.3) provides a standard interface to the popular high-level synthesis tools [17][15] [64]. This improves the programming productivity and code portability across different types of FPGAs. In addition, ViTAL also 1) creates an illusion of a *single* and *infinitely large* FPGA to support scale-out acceleration transparently, and 2) provides a runtime resource management system to improve utilization and concurrency (Section 3.4). These two features help to further reduce the programming complexity since 1) the physical FPGA boundary and interconnection bandwidth/topology are abstracted away, and 2) users can develop applications as if they have the total unrestricted control of entire FPGA resources, regardless of the resource usages of any other applications running concurrently on the FPGA cluster.

3.2 Architecture Layer

This architecture layer provides a new system abstraction that serves as an intermediate layer between the physical FPGAs and the compilation layer (Section 3.3) to help decouple the compilation and resource allocation. Besides the on-chip resources provided by FPGAs (e.g., LUTs and BRAMs), this abstraction also provides virtualization support for the peripheral devices attached to the FPGA (e.g., on-board DRAMs) to ensure a secure execution environment.

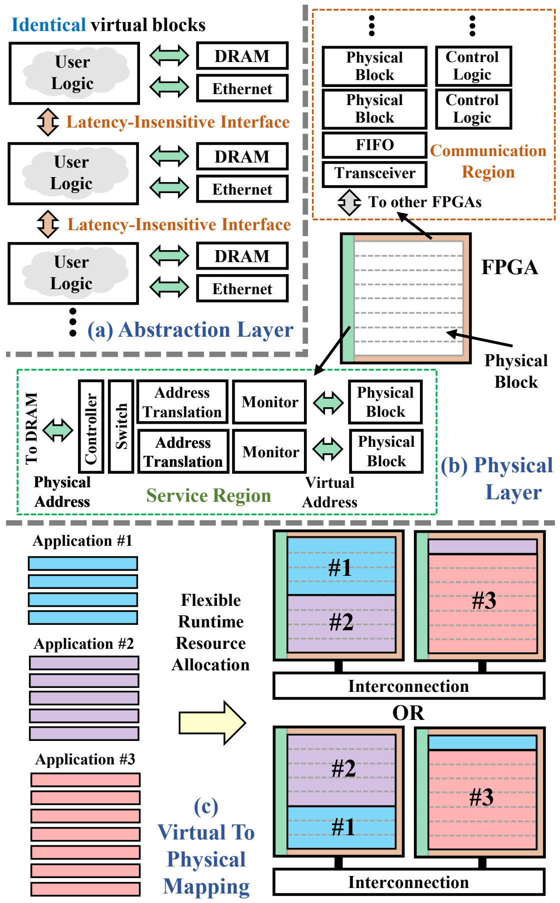

Figure 4. (a) ViTAL provides a homogeneous abstraction that comprises an array of *identical* virtual blocks. The inter-block communication is realized by a latency-insensitive interface to hide the bandwidth/latency difference between on-chip and off-chip interconnection. (b) The physical FPGA is divided into three regions to support this abstraction. Note that the actual layout of these regions is tailored to a specific type of FPGA. (c) The user logic contained in one virtual block is offline compiled into one physical block, and can be relocated into an *arbitrary* physical block at runtime without recompilation. This enables a fine-grained and flexible FPGA sharing.

The key to decouple the compilation and resource allocation is to generate a *position-independent* mapping for applications. Note that the mapping generated by the existing FPGA compilation tool is *position-dependent* because 1) the state-of-the-art FPGA architecture comprises a sea of heterogeneous resources, a certain type of which (e.g., on-chip memories) is only available at several fixed spatial locations (columns) of the chip (Fig. 3a), and 2) the bandwidth/latency of the on-chip interconnect network (configurable routing fabric) is different from that provided by the off-chip network. Such heterogeneity in both compute/memory resources and interconnect networks introduces a spatial constraint into the mapping, i.e., applications can only be deployed onto a fixed location that is determined at the offline compile time. Therefore, the system cannot perform a dynamic resource management in response to the load and resource availability without a time-consuming recompilation of applications.

In order to enable a position-independent compilation, ViTAL provides a *homogeneous* abstraction to hide the aforementioned heterogeneity. As shown in Fig. 4a, this abstraction comprises an array of *identical* virtual blocks, and all virtual blocks provide the same amount of programmable resources and the same interface to the peripheral devices, e.g., on-board DRAMs. Moreover, each virtual block also contains a latency-insensitive interface for inter-block communication. This interface effectively hides the latency difference between the on-chip and off-chip interconnect network. Therefore, virtual blocks deployed on different FPGAs and those deployed on the same FPGA can have an identical inter-block communication interface. This identical interface simplifies the partition process performed by the compilation tool (Section 3.3) and guarantees the functional correctness (avoid timing failure). A clock enable signal is generated by this interface to clock-gate user logic when no input data is available. The compilation tool (Section 3.3) automatically generates this interface without adding extra burden to users.

ViTAL also provides virtualization support for the peripheral devices attached to the physical FPGAs. For instance, ViTAL provides a virtual memory support to share the off-chip DRAM. User applications use virtual address to access the data stored in the off-chip DRAM, which is then translated into the physical address. The memory access from applications are monitored to ensure a secure execution environment.

To support this abstraction, a physical FPGA is partitioned into three regions, i.e., **Service Region**, **Communication Region** and **User Region** (Fig.4b). The **Service Region** and the **Communication Region** are reserved by the system and are not exposed to users. Specifically, the **Service Region** contains dedicated modules to realize the virtualization support provided for the peripheral devices. The **Communication Region** includes buffers and control logic to implement the latency-insensitive interface. The **User Region** is further divided into a group of identical physical blocks. Note that to create identical physical blocks on FPGAs that have heterogeneous resources, we need to carefully partition the FPGA. As illustrated in Fig.3a, the existing FPGAs have column-based architecture comprising multiple columns where each column contains the same type of resources (e.g., LUTs/BRAMs/DSPs). To keep the physical blocks identical, we choose to partition a physical FPGA in the row direction where the periodicity in the architecture is preserved. During operation, one virtual block is deployed into one physical block. Specifically, the user logic in one virtual block is offline compiled into one physical block (Section 3.3). Since all physical blocks are identical, the virtual block can be relocated into an *arbitrary* physical block at runtime without recompilation (Fig. 4c). Note that the resources provided by one physical block is only a small fraction of the resources provided by the entire FPGA, thereby enabling a fine-grained resource management.

Key learning from real system prototyping: Inherited from multiple product generations, the contemporary commercial FPGAs have more complex architectural features (e.g., clock region and multi-die package) than the simplistic FPGA architecture (Fig. 3a) that is frequently cited in textbooks or publicly available tutorials. These features introduce extra heterogeneities that may further break the periodicity of the column-based FPGA architecture and increase the difficulty of creating identical physical blocks in the user region. For instance, the modern commercial FPGAs introduce clock regions that contain buffers and interconnection network to distribute clock signals. To account for such added heterogeneity brought by clock resources, we apply a constraint on the relative position between physical blocks and clock regions to ensure that the clock skew within the physical block remains the same among all blocks, thereby avoiding timing failure. Moreover, commercial FPGAs comprise multiple dies in a single package to increase the capacity, but introduce bandwidth/latency difference between the intra-die and inter-die routing network. We thus apply another constraint to hide such heterogeneity and guarantee the physical block does not cross the die boundary. To account for these practical constraints, the physical view of a partitioned Xilinx FPGA is shown in Fig. 7.

3.3 Compilation Layer

The compilation layer provides a generic compilation flow to map applications written in various high-level programming languages onto different FPGAs across vendors, and generates mapping results that can be managed by the runtime system (Section 3.4). To implement the comprehensive flow while keeping the development cost under control, we reuse the proprietary FPGA tool (Vivado in our implementation) and develop a set of new tools either from scratch or by leveraging the open-source RapidWright tool from Xilinx [41]. These tools are *modularly* integrated into a unified compilation flow as shown in Fig. 5. Therefore, ViTAL's compilation layer, by design, is flexible as 1) it reuses the development environment (the front-end) of commercial FPGA compilation tools (such as Vivado) and thus can effectively leverage the existing efforts built atop these tools. For instance, it can be easily extended to support various high-level programming languages/frameworks (Section 3.1). 2) It provides a standard interface with the back-end of commercial FPGA tools to support various FPGA families from different vendors.

As shown in Fig. 5, the compilation flow contains six steps: synthesis, partition, latency-insensitive interface generation, local place-and-route (P&R), relocation and global P&R.

1) **Synthesis**: This step converts applications written in high-level programming languages into Verilog RTL code, then synthesizes Verilog RTL code into a netlist of primitives (e.g., LUTs). We reuse the front-end of the commercial tool in this step to preserve the existing development environment.

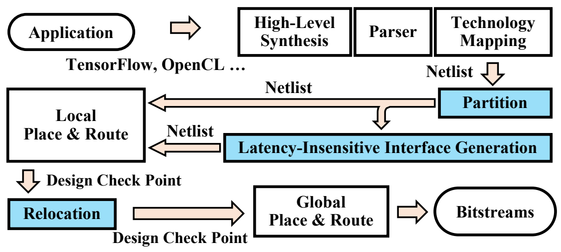

Figure 5. The ViTAL compilation flow. Custom tools (blue) are modularly integrated into the proprietary FPGA tool.

2) **Partition**: This step partitions a given netlist generated from previous step into a group of virtual blocks. The goal of this step is to minimize the number of inter-block connections, thereby lowering the requirement on the inter-block communication bandwidth. We develop a custom tool to implement this step, and the key algorithm is described in Section 4.

In principle, this partition step can be performed at different levels [54], i.e., 1) the level of control data-flow graphs (CDFGs), 2) the level of data-flow graphs (DFGs) and 3) the level of netlists. One key design decision we made in ViTAL is to partition applications at the *netlist* level, due to the following reasons, 1) netlist is a generic intermediate representation (IR) and does not depend on the programming languages. Therefore, performing partition on netlist makes it easy to extend our tool to support various programming languages. 2) Netlist also provides an accurate estimation on the low-level resource usages (e.g., number of LUTs and BRAMs), which is difficult to be obtained in the other two levels. The detailed resource estimation can be effectively utilized by the custom tool to improve the quality of the partition.

3) **Latency-Insensitive Interface Generation**: This step generates the circuits that realize the latency-insensitive interface. The generated circuits contain two parts, 1) FIFOs for data buffering and 2) a control logic that manages buffers (handles back-pressure) and generates clock enable signals. We develop a custom tool to generate these circuits by analyzing the dataflow graph of the user logic in the virtual block.

4) **Local Place-and-Route:** This step maps the user logic in a virtual block into a physical block. It also maps the generated latency-insensitive interface (Step 3) into the communication region. To map applications onto commercial FPGAs, we reuse the P&R stage of commercial tools in this step.

5) **Relocation:** This step relocates a mapped virtual block (obtained in Step 4) to a different physical block without recompilation to support different resource allocations. Without this step, we need to compile a virtual block into all possible physical blocks it might be relocated to, which substantially

increases the compilation time (>10×). To minimize the compilation overhead, we develop a custom tool that leverages the APIs in RapidWright [41] to achieve low-overhead relocation.

6) **Global Place-and-Route:** In this step, the individually implemented components are integrated into a complete design, and the interconnection between these components are routed to generate the final mapping results. This step can be realized by reusing the P&R stage of commercial FPGA tools.

Key learning from real system prototyping: We identified two major incompatibility issues when implementing the proposed compilation flow. Specifically, to implement the Step 4 and 6, we reuse the hierarchical design [67] and partial reconfiguration [68] provided by Vivado. Nevertheless, these two functions are not compatible, since the hierarchical design is a bottom-up flow[2], while the partial reconfiguration is a top-down flow[3]. Simply integrating them into a unified compilation flow may lead to timing failure or even unroutable designs. By adding constraints and changing the sequence of commands, we modified the partial reconfiguration into a bottom-up flow and resolved this incompatibility. Moreover, we leverage the APIs from RapidWright to relocate a mapped virtual block without recompilation (Step 5). However, these APIs change the top-level module of a given design after relocation, thus, Vivado cannot recognize this design and is not able to generate the final mapping results. We modified several configurations in the Vivado to resolve this incompatibility.

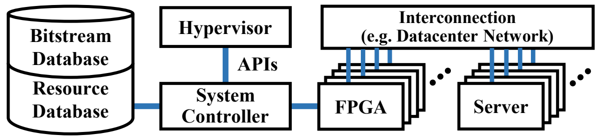

Figure 6. A conceptual diagram illustrates the ViTAL system. The system controller performs runtime resource management and exposes APIs for an easy system integration.

3.4 System Layer

The system layer performs runtime resource management to deploy the compiled applications (Section 3.3) onto the physical FPGAs. A runtime management policy is provided to enable an efficient FPGA sharing and improve the aggregated system throughput. It also provides a complete isolation of applications to ensure a secure execution environment.

As shown in Fig. 6, a system controller is added to perform the resource management. It maintains a resource database to store the status of all physical blocks, and a bitstream database to store the mapping results of user applications. When the high-level system (e.g., Hypervisor) requests to deploy an application into the cluster, the system controller first searches the resource database to find FPGAs that have enough resource, and then searches the bitstream database to find the corresponding bitstream. Finally, the physical FPGAs are reprogrammed with the bitstream using partial reconfiguration [34] (a feature that supports fast reconfiguration without affecting other co-running applications). The system controller also provides APIs to communicate with the high-level system, which enables an easy system integration.

[2]Map each module first, then map top-level design.

[3]Map top-level design first, then map each contained module.

As shown in Fig. 4c, one application can be flexibly partitioned and deployed onto multiple physical FPGAs to effectively alleviate the fragmentation issue caused by the physical FPGA boundary. Nevertheless, mapping an application onto multiple FPGAs requires an inter-FPGA communication, which may incur additional overheads. In order to minimize the inter-FPGA communication and improve the scalability of the ViTAL system, a *communication-aware* runtime management policy is provided, which allocates the physical blocks in a multi-round manner. In the first round, it tries to find a single physical FPGA that has a sufficient amount of physical blocks to deploy an application. It then increases the number of physical FPGAs in the following rounds until a feasible allocation is found. This policy effectively reduces the number of FPGAs used to implement one application and minimizes inter-block communication overhead. Further exploration on more comprehensive runtime policy will be our future work.

In principle, ViTAL supports the case that the virtual blocks of multiple applications can be mapped into the same physical block if these applications share the same function. Although this strategy may further improve the resource utilization, we did not apply it to the current ViTAL system implementation due to two reasons. 1) Multiplexing one physical block among multiple users reduces the performance (throughput) delivered to a single user. 2) In current cloud (e.g., AWS), the compiled applications are encrypted [2], therefore, it is difficult for the system controller to determine whether two virtual blocks perform the same function or not. Consequently, one physical block is not shared among multiple virtual blocks in ViTAL. This enables a complete isolation and effectively protects applications from different types of attack, such as side-channel attack [71][40]. We plan to study more comprehensive sharing in our future work.

3.5 Discussions

In this section, we address potential additional questions regarding ViTAL.

3.5.1 Handle Back-Pressure and Avoid Deadlock

The control logic in the latency-insensitive interface is carefully designed to handle back-pressure and avoid deadlock. Specifically, when the output buffer is full, the user logic will be clock-gated by the control logic and its execution will be halted. As the user logic does not generate new output data, it effectively controls the back-pressure from the upstream user logic. Moreover, we provide additional mechanisms to avoid deadlock. For instance, buffers in the interface are correctly initialized and user logic is controlled in a fine-grained manner (Section 3.3). These mechanisms guarantee that at least one input buffer is not empty during execution – a condition, once satisfied, can avoid deadlock, as theoretically proved in [4].

3.5.2 System Reserved Resource

The programmable resources in the communication and service regions are reserved by the system and are not available to users. Thus, it is desirable to minimize the amount of system reserved resources and to allow more resources for user applications. Note that the resources dedicated to the service region is small. However, the communication region could consume non-negligible resources, since the user region has many physical blocks (due to a fine-grained partition), all of which would require some form of inter-block communication and a large amount of buffers for storing the input/output data. Nevertheless, we find that buffers for inter-block communication on the same FPGA can be eliminated, since such on-chip communication has a deterministic latency which can be resolved during offline compile time. The control logic thus can calculate the arrival time of input data based on this latency, and resume the execution of user logic to consume the input data when it arrives. Removal of buffers for intra-FPGA communication significantly reduces the resource occupied by the communication region. More detailed evaluation can be found in Section 5.3. Finally, we also note that system regions have a pre-defined functionality and a limited variance across different applications. Thus, circuits in these regions can be implemented by dedicated hard IP blocks to further reduce the amount of system reserved resource. We plan to investigate this additional optimization in our future work.

4 ViTAL Compilation: Partition Algorithm

ViTAL compilation flow comprises custom tools to partition applications and generate the latency-insensitive interface. The method for the interface generation is relatively straightforward. On the contrary, partition can be a non-trivial task as the quality of the final mapping is highly sensitive to this step. Thus, we describe the partition algorithm in this section.

A placement-based partition algorithm is applied to partition applications into a group of virtual blocks. Specifically, applications are first placed onto a pre-defined 2D space, and then are partitioned based on the placement results. This algorithm simultaneously minimizes the number of inter-block connection and maximizes the operation frequency of applications by simply solving a linear equation system (low runtime complexity). Details are described in following subsections.

4.1 Packing

A greedy algorithm is applied to quickly pack user logic into coarse-grained clusters, thereby significantly reducing the runtime complexity of the global placement step (Section 4.2). Specifically, a randomly selected unpacked logic primitive is used as the seed of a cluster. Then an unpacked logic primitive with the highest *attraction score* (Algorithm 1) is packed into this cluster. This operation is repeatedly performed until the size of this cluster reaches the given capacity. Then another randomly selected unpacked logic primitive is used to build a new cluster. Finally, small clusters are merged into other clusters to reduce the number of clusters.

4.2 Global Placement

This step places the packed user logic onto a pre-defined 2D space and then partitions it based on the placement results. Specifically, each virtual block is assigned with a position (x,y) and an aspect ratio α. The packed user logic is placed onto this 2D space using the quadratic placement method [69].

Algorithm 1 Calculate the attraction score

```
Input: The data-flow graph $G = (V, E)$, a logic primitive
    $n \in V$ and a logic cluster $C$
    $S_1 = \{v | v \in V \cap (n,v) \in E \cap v \neq n\}$
    $S_2 = \{v | v \in C \cap (n,v) \in E\}$
    $Score = |S_2| / |S_1|$
```

(1) **Solve Linear Equation System:** The quadratic placement algorithm minimizes the total length of routing paths by solving a linear equation system. Specifically, if the Euclidean distance is used as the metric to estimate the total length of routing paths, then a linear equation system can be built according to following equations:

$$L = \sum_{i,j} w_{ij} [\alpha(x_i - x_j)^2 + (y_i - y_j)^2] \tag{1}$$

$$\frac{\partial L}{\partial x_i} = \sum_{i} w_{ij}(x_i - x_j) = 0, \frac{\partial L}{\partial y_i} = \sum_{i} w_{ij}(y_i - y_j) = 0 \tag{2}$$

where (x_i,y_i) is the position of packed cluster i, and $w_{ij} = 0$ if packed cluster i is not connected with packed cluster j, otherwise $w_{ij} > 0$. By solving this linear equation system (use Eigen C++ library [27]), the packed clusters will be placed into the optimal positions with a minimal total interconnection length. Note that solving this linear equation system also minimizes the number of inter-block connections, since two connected clusters will not contribute to the total routing length (Equation 1) if they are placed in the same virtual block.

(2) **Create Legal Placement:** The placement result obtained from step (1) might be illegal (at least one virtual block is over-utilized), as it ignores the capacity of virtual blocks. Thus, one more step is performed to create a legal placement based on the result obtained from step (1). This step uses the simulated annealing algorithm [60] to resolve the over-utilization issue, and the cost function used in this process is:

$$Cost = \frac{\sum(\alpha|x_i - x_i'| + |y_i - y_i'|)}{N_{cluster}} + \frac{\sum f_i}{N_{block}} \tag{3}$$

where (x_i', y_i') is the position of cluster i obtained from step (1), (x_i, y_i) is the new position of cluster i, and $N_{cluster}$ is the number of packed clusters. Function f_i will output zero if $block_i$ is not over-utilized, otherwise, it will output a large positive number. N_{block} is the number of virtual blocks.

With this cost function, this step simultaneously eliminates the over-utilization and minimizes the total move distance of all clusters. Minimizing the total move distance is necessary to avoid a significant increase of the total routing length. After this simulated annealing process, a recovery process is performed to reduce the total length of interconnection, and the density preserving refinement method proposed in [44] is adapted into our tool to perform this recovery process.

(3) **Add Pseudo Cluster and Pseudo Connection:** The placement results obtained from Step 2 is included into the linear equation system by adding pseudo clusters/connections and using Equation 4 to calculate the total length.

$$L' = L + \sum_{ij} \beta_{ij}[\alpha(x_i - x''_j)^2 + (y_i - y''_j)^2] \qquad (4)$$

where (x''_j, y''_j) is the position of the pseudo cluster j, which is obtained from Step (2). β_{ij} is the weight between cluster i and pseudo cluster j, which is zero if $i \neq j$. Then a new linear equation system can be built based on this equation. The pseudo clusters serve as *anchors* that coerce the initial locations (obtained in Step 1) to alleviate the over-utilization.

(4) **Repeat Step (2) and (3):** Step (2) and (3) are iterated with a slowly increased β to slowly pull away clusters from the over-utilized virtual blocks. This iteration is terminated when the gap between the total interconnection length (Equation 1) obtained from Step (2) and Step (3) is less than 20%.

5 Evaluation

In this section, we evaluate the ViTAL stack on the latest commercial FPGAs (XCVU37P). As the ViTAL stack comprises multiple layers (Section 3), we provide an extensive evaluation for each layer to comprehensively evaluate the whole stack.

5.1 Benchmark Selection

Three sets of benchmarks with varying size and complexity are used to evaluate the Architecture layer, Compilation Layer and System Layer, respectively.

The first benchmark is relatively small and is synthetically generated to evaluate the abstraction (Architecture Layer). Specifically, it generates random data traffic to identify the maximum bandwidth provided by the latency-insensitive interface for the inter-FPGA/inter-die communication.

The second benchmark is a set of machine learning applications. It is applied to evaluate the offline compilation performance (Compilation Layer). We choose machine learning applications because they are the representative workloads for state-of-the-art cloud FPGAs (e.g., Microsoft Azure [25]). Nevertheless, the real-world FPGA-based cloud workloads are proprietary and are not accessible in public domain, thus, we choose to use the open-source DNNweaver [55] to generate DNN benchmarks, which has also been used in prior work [35]. By adjusting the input design parameters (e.g., number of processing units) of this framework, we further provide three variants of accelerator designs (small, medium and large) for each benchmark to better account for the varying performance/cost demands in the dynamic cloud environment. The details of these benchmarks and corresponding accelerator designs are listed in Table 2.

The third benchmark is the largest and most complex one among the three. It contains multiple applications that can concurrently run on the FPGA cluster, and is used to evaluate the runtime performance (System Layer) in the cloud setting. In principle, real-world cloud workload sets are preferred in this evaluation to prove the generality of ViTAL. However, as there is no publicly available real-world cloud workloads using FPGAs, in this experiment, we follow the approach that has been widely used in prior work [51] and synthetically generate several workload sets with different compositions (Table 3) to provide a comprehensive evaluation, while further study on the benchmark development for the cloud FPGAs will be our future work. In this evaluation, each workload set comprises a sequence of DNN benchmarks (from the second benchmark set), and the requests for deploying these benchmarks are issued with a random time interval to emulate the dynamic cloud environment. For each condition (composition and time interval), multiple workload sets are generated and the average result is reported.

Table 2. The resource usage of three accelerator designs (small, medium and large accelerator) is reported.

Network	Resource Usage				#Block
(Data Set)	LUT	DFF	DSP	BRAM	
CIFAR-10	23.5k	23.3k	42	2.6Mb	1
FULL	94k	93.2k	168	10.4Mb	4
(CIFAR-10)	164.5k	163.1k	294	18.2Mb	7
	55.2k	52.9k	104	6.1Mb	2
LeNet	138k	132.3k	260	15.3Mb	5
(MNIST)	220.8k	211.6k	416	24.5Mb	8
	23.3k	23.7k	48	3.0Mb	1
NiN	70.0k	71.1k	144	9.0Mb	3
(ImageNet)	210k	233.2k	432	26.9Mb	9
	80.7k	80.6k	156	9.4Mb	3
AlexNet	188.3k	188.1k	364	21.9Mb	7
(ImageNet)	269k	268.7k	520	31.3Mb	10
	46.0k	45.3k	84	5.3Mb	2
VGG-CNN-S	115k	113.3k	210	13.3Mb	5
(ImageNet)	184k	181.3k	336	21.3Mb	8
	24.9k	24.9k	50	3.1Mb	1
Overfeat	74.7k	74.7k	150	9.4Mb	3
(ImageNet)	149.4k	149.4k	300	18.8Mb	6
	77.2k	75.0k	144	9.0Mb	3
VGG-16	128.7k	125k	240	14.9Mb	5
(ImageNet)	257.3k	250k	480	29.9Mb	10

5.2 Experimental Setup

We evaluate ViTAL on a custom-built FPGA cluster that comprises four latest Xilinx UltraScale+ FPGA (XCVU37P). Each FPGA board provides four 1×4 ganged 28Gb/s QSFP+ cage for high speed optical communication, and these four FPGAs share access to a 100 Gbps bidirectional ring for the inter-FPGA communication. Each FPGA also provides two DIMM sites and each supports up to 128GB DDR4x72 with ECC. We did not choose AWS F1 platform nor Microsoft Catapult [58] due to tool constraints (limited flexibility exposed to users) and/or licensing issues. As the setup of the custom-built FPGA cluster is similar to that of the AWS F1 platform, we expect that ViTAL can achieve a similar performance on AWS F1 platform.

Vivado 2018.3 is used to compile applications and generate bitstreams. For ViTAL, the proposed compilation flow (Fig. 5) is used to compile applications. Partial reconfiguration [68] is applied to configure each physical block individually at runtime, thus an application can be deployed into the FPGA cluster without affecting other co-running applications.

Table 3. Several representative compositions are considered when synthetically generating workload sets.

Set Index	Composition
1	100% **S**
2	100% **M**
3	100% **L**
4	50% **S** + 50% **M**
5	50% **S** + 50% **L**
6	50% **M** + 50% **L**
7	33% **S** + 33% **L** + 34% **L**
8	20% **S** + 20% **M** + 60% **L**
9	20% **S** + 60% **M** + 20% **L**
10	60% **S** + 20% **M** + 20% **L**

*__S/M/L__ represents the small/medium/large accelerator design.

The management method that allocates FPGA resource at a per-device granularity is used as a baseline in the evaluation. The reason for choosing that is the wide adoption of such management method by cloud vendors [1]. To make our evaluation more complete, we also include the recent work AmorphOS [35] in our comparison. Among AmorphOS's two operating modes, i.e., low-latency mode and high-throughput mode, we specifically choose the high-throughput mode in our experiment, as it achieves a higher resource utilization (and aggregated system throughput) than prior slot-based method [11][63] (including the low-latency mode of AmorphOS [35]). In the rest of this section, we refer to AmorphOS as AmorphOS running in high-throughput mode, unless otherwise specified.

5.3 Architecture Layer Evaluation

As discussed in Section 3.2, a physical FPGA is partitioned into regions to support the proposed system abstraction (Fig. 4). To determine the optimal FPGA partition, we perform design space exploration 1) to maximize the amount of resources exposed to users, and 2) to maintain a fine-grained resource management. As the commercial FPGA introduces extra heterogeneities and thus adds more constraints (Section 3.2), our search space is relatively small (<10 possible partitions). We exhaustively evaluate all cases and find the optimal partition that is shown in Fig. 7.

In the optimal FPGA partition, region 1 is the user region, while region 2-6 are the communication/service region and are reserved by the system. Specifically, region 1 contains an array of physical blocks and each physical block contains two sub-blocks (region 1a and 1b). Region 2 and 3 are used to implement the latency-insensitive interface for the inter-FPGA/inter-die communication. Region 4 is the service region and implements the essential circuits to securely share the DRAM interface to all physical blocks. Region 5 comprises high-speed transceivers to realize the inter-FPGA communication, and region 6 contains pipeline registers to connect the transceivers with the latency-insensitive interface. We further evaluate the proposed optimization method of removing buffers for intra-die communication (discussed in Section 3.5.2). From the results, we confirm that the amount of system reserved resource is reduced by 82.3% compared with the case without such optimization and is effectively controlled below 10% of the total resources.

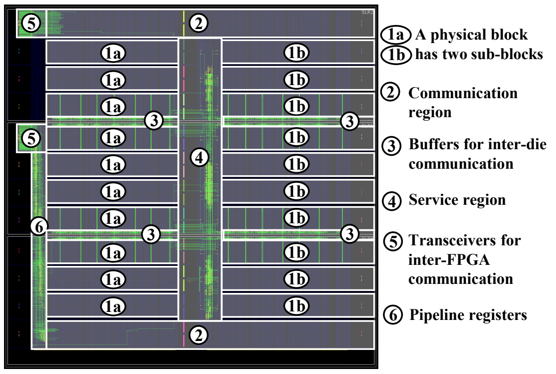

Figure 7. The latest commercial FPGA (XCVU37P) from Xilinx is partitioned into regions to support the proposed system abstraction. Region 1 is exposed to users, while region 2-6 are reserved by the system. The circuits in the system-reserved regions are pre-implemented and cannot be modified by users. The mapping results are obtained from Vivado 2018.3.

Table 4. The bare-metal performance

Resource provided by a physical block			
LUTs	DFFs	DSPs	BRAM
79.2k	158.4k	580	4.22Mb
Communication Performance			
Type	**Inter-FPGA**		**Inter-Die**
Maximum Bandwidth	$94 Gb/s$		$312.5 Gb/s$

We apply the first benchmark set to evaluate the latency-insensitive interface. Specifically, we measure the maximum inter-block communication bandwidth provided by the inter-FPGA/inter-die connection. The bare-metal performance is presented in Table 4.

5.4 Compilation Layer Evaluation

The second benchmark set (Table 2) is applied to evaluate the ViTAL compilation flow. At first, we measure the compilation time and report the detailed breakdown. As shown in Fig. 8, the compilation time is dominated by the place&route stage (83.9%) of the commercial tool, while the runtime of custom tools only takes 1.6% of the total compilation time on average. This confirms that ViTAL has a low compilation overhead although it adds custom tools into the commercial FPGA compilation flow. Such low overhead comes from the low runtime complexity of these custom tools. Specifically, the partition step, which dominates the runtime of the custom tools, has a small search space and only needs to map user applications into a small number of virtual blocks (<=10 in our evaluation). On the contrary, the place&route stage needs to map hundreds of thousands of logic primitives (e.g., LUTs). Thus, the partition step has a much shorter runtime than the place&route stage. Note that to achieve high resource utilization (and thus aggregated system throughput), AmorphOS needs to transition from its slot-based sharing mode into its high-throughput mode, which requires offline compiling hundreds of combinations for the evaluated benchmark set. In addition to the compilation time, we also evaluate the mapping quality of the ViTAL compilation. To minimize inter-block communication, our results show that the algorithmic optimization (Section 4) in the partition step effectively reduces the required bandwidth of inter-block interconnections by 2.1× on average across the evaluated benchmarks.

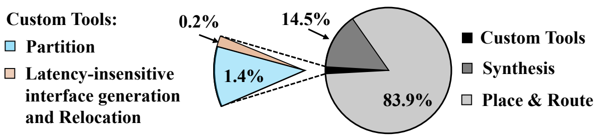

Figure 8. The breakdown of the ViTAL compilation time.

5.5 System Layer Evaluation

The workload sets in Table 3 (the third benchmark) are applied to evaluate the performance of the system layer on the custom-built FPGA cluster. We first evaluate the response time of user applications, which includes the wait time (the time spent waiting for the necessary resources to deploy accelerators) and the service time (the execution time of the deployed accelerator). It is a widely used metric to measure the quality of service (QoS) [59]. Overall, ViTAL reduces the response time by **82%** on average compared to the baseline (Fig. 9) due to the fine-grained sharing in ViTAL compared with the coarse-grained sharing (per-device) in the baseline system. Moreover, compared with AmorphOS running in high-throughput mode, ViTAL also achieves **25%** reduction in response time on average. The reason for such performance improvement over AmorphOS is that ViTAL implements fine-grained sharing across *multiple* FPGAs (for instance, it is observed that 5%~40% of evaluated applications have been partitioned and deployed onto multiple FPGAs in this experiment), whereas the high-throughput mode of AmorphOS only achieves fine-grained sharing on a *single* FPGA and does not provide multi-FPGA support. Due to the lack of the multi-FPGA support, we also observe that the response time improvement on AmorphOS is limited in certain scenarios when workloads cannot be combined and deployed on the same FPGA (e.g., workload set #3). We expect such a case will grow more common because of the increasing size and diversity of applications.

Our evaluation results further confirm ViTAL effectively alleviates the resource fragmentation issue and improves the resource utilization by 15.9% compared to AmorphOS running in high-throughput mode. Additionally, the fine-grained resource sharing (Fig. 10) in ViTAL also improves concurrency. Our evaluation results show the number of applications that can concurrently run in the FPGA cluster increases by **2.3×** over that in the baseline system. It indicates that the cloud providers can serve more user requests with the same amount of FPGA resources. We also observe that our partition tool (Section 4) and the proposed runtime policy (Section 3.4) effectively minimize the latency overhead introduced by the latency-insensitive interface, making it become negligible compared with the total execution time of the evaluated benchmarks (< 0.03%). Finally, the FPGA utilization (number of utilized physical blocks) remains high (above **93%**) in the ViTAL system, which confirms the effectiveness of ViTAL in managing resources in a dynamic cloud environment.

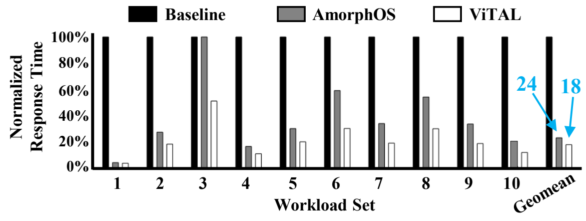

Figure 9. The normalized response time.

6 Related Work

6.1 FPGA Abstraction

The overlay architecture [5][39][65][16][8][45] provides an abstraction layer to enable code portability across different types of FPGAs. This method is functionally analogous to the Java Virtual Machine (JVM). Specifically, applications are first compiled onto the abstracted architecture, and then mapped onto the physical architecture. The major limitation of these abstractions is that they are typically *application specific* and developed for a *single* FPGA in a non-sharing environment. Therefore, unlike ViTAL, they cannot be applied to enable an efficient resource sharing among multiple users and support scale-out acceleration in the cloud.

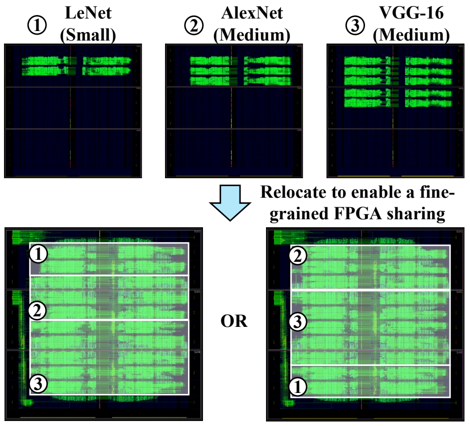

Figure 10. Applications can be relocated to realize a flexible FPGA sharing. Mapping results are obtained from Vivado.

Several frameworks [53][19][29][12] have been proposed to support scale-out acceleration using multiple FPGAs. Some commercial evaluation platforms (e.g., Cadence Protium S1

[7]) also use multiple FPGAs for scale-out emulation. These frameworks/platforms provide valuable experience on application partitioning across FPGAs, however, they do not address the tight coupling between compilation and resource allocation as ViTAL does. Thus, the resource allocation decision in these frameworks/platforms still needs to be made at the offline compile time, and these frameworks cannot enable a fine-grained resource sharing among users.

Other frameworks [36][61][14] are proposed to abstract the communication interface of FPGAs (e.g., PCIe and DRAM channel), therefore, users can focus on the kernel development. However, these abstractions are not developed to decouple the compilation and resource allocation, thus they cannot enable FPGA sharing.

Although these abstractions are insufficient to address the aforementioned limitations, we note that they can be combined with the abstraction provided by ViTAL to offer another level of programmability and further improve productivity.

6.2 Cloud FPGA Virtualization

SCORE [20][9][10] is an early pioneer work that falls into the broad context of FPGA virtualization before cloud computing become ubiquitous. It aims to reduce the FPGA programming complexity and support scale-out acceleration by providing a stream-oriented compute model and a new abstraction. To support such compute model and abstraction, it further proposes to partition physical FPGAs into compute and memory regions. The methodology presented in SCORE is inspiring but it mainly targets the (non-sharing) single-user, single-application environment for embedded computing systems. Later, this partition-based method is extended by several works [11][6][63][38][22][37] to target multi-tenant cloud computing. These works abstract the FPGA resources into a pool of identical slots (slot-based method), and all slots provide the same amount of resources and the same interface (e.g., PCIe and Ethernet). Physical FPGAs are then partitioned into identical regions, and one region is used to implement one slot. This method decouples the compilation and resource allocation. However, the improvement in resource utilization can be limited due to the internal fragmentation issue. Moreover, these works do not support scale-out acceleration as ViTAL does and still require users to manually partition applications into multiple slots.

The recent work AmorphOS [35] tackles the problem to achieve better FPGA sharing by providing two modes of operation, i.e., low-latency mode and high-throughput mode. Specifically, 1) the low-latency mode applies the same slot-based method and thus still has the internal fragmentation issue. 2) The high-throughput mode wraps multiple applications into a single application, which is then compiled onto a single physical FPGA. Compared with ViTAL, the high-throughput mode of AmorphOS 1) has a lower resource utilization (Section 5.5) due to the lack of multi-FPGA support, and 2) has a higher offline compilation overhead (Section 5.4), as it does not decouple the compilation and resource allocation.

6.3 OS Support for FPGA

Operation system support for FPGA is an active area of research, and representative works include BORPH [56], ReconOS [46], hthreads [3] and FUSE [31]. These works abstract applications running on FPGAs as hardware threads or processes. The interface provided by these hardware threads/processes is same as that of the threads/processes running on CPU. Therefore, operation system can manage and communicate with these hardware threads/processes as if they were running on the CPU. Compared with ViTAL, one common limitation of these works is that they do not address the coupling issue between the compilation and resource allocation.

The OS support provided by LEAP [24] uses a latency-insensitive communication interface to enable multiple-FPGA acceleration. This is somewhat similar to the communication interface used in ViTAL. However, the major difference between LEAP and ViTAL is that LEAP requires users to manually partition applications into modules and uses special pragmas to create this latency-insensitive interface, while these are automatically done by the compilation tool in ViTAL. Moreover, LEAP still needs to perform the resource allocation at the offline compile time, while ViTAL performs the resource allocation at runtime to dynamically respond to the actual load and resource availability in the elastic clouds.

7 Conclusion

In this paper, we present a full-stack solution to virtualize FPGA clusters in the cloud environment. Specifically, ViTAL provides a homogeneous abstraction to effectively decouple the compilation and resource allocation. It allows ViTAL to dynamically perform the resource allocation at runtime to improve the management efficiency. Moreover, ViTAL creates an illusion of a *single/large* FPGA to users and provides a complete compilation flow. The compilation flow automatically maps applications onto the proposed abstraction and efficiently supports scale-out acceleration. We evaluate the ViTAL stack on a real system. Compared with the existing resource management method, ViTAL reduces the response time by 82% due to the fine-grained sharing with a marginal compilation overhead. The response time in ViTAL is also 25% lower compared to AmorphOS (operating in high-throughput mode), a recently proposed FPGA virtualization method. We also note that 1) ViTAL can be extended to virtualize a *heterogeneous* FPGA cluster comprising different types of FPGAs, and 2) ViTAL is not limited to FPGAs, but can be generally applied to virtualize other dataflow reconfigurable accelerators (such as TPU or other domain-specific accelerators) in the cloud. We plan to explore these directions in our future work.

Acknowledgements

We thank the anonymous reviewers for their insightful comments and feedback to help improve the quality of the paper. We also would like to thank Jialiang Zhang and Lifu Zhang for system prototyping and benchmark preparation. This work was in part supported by DARPA Young Faculty Award D16AP00122.

References

[1] Amazon. 2016. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/.

[2] Amazon. 2017. Accelerated Computing on AWS. http://asapconference.org/slides/amazon.pdf.

[3] David Andrews, Wesley Peck, Jason Agron, Keith Preston, Ed Komp, Mike Finley, and Ron Sass. 2005. hthreads: A Hardware/Software Co-Designed Multithreaded RTOS Kernel. In *Emerging Technologies and Factory Automation, 10th IEEE Conference on*, Vol. 2. IEEE, 8 pp.–338.

[4] Daniel Brand and Pitro Zafiropulo. 1983. On Communicating Finite-State Machines. *J. ACM* 30, 2 (1983), 323–342.

[5] Alexander Brant and Guy GF Lemieux. 2012. ZUMA: An Open FPGA Overlay Architecture. In *FCCM*. IEEE, 93–96.

[6] Stuart Byma, J Gregory Steffan, Hadi Bannazadeh, Alberto Leon Garcia, and Paul Chow. 2014. FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack. In *FCCM*. IEEE, 109–116.

[7] Cadence. 2017. Protium S1 FPGA-Based Prototyping Platform. https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/system-design-verification/protium-s1-fpga-based-prototyping-platform-ds.pdf.

[8] Davor Capalija and Tarek S Abdelrahman. 2013. A High-Performance Overlay Architecture for Pipelined Execution of Data Flow Graphs. In *FPL*. IEEE, 1–8.

[9] Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, John Wawrzynek, and André DeHon. 2000. Stream Computations Organized for Reconfigurable Execution (SCORE). In *International Workshop on Field Programmable Logic and Applications*. Springer, 605–614.

[10] Eylon Caspi, André DeHon, and John Wawrzynek. 2001. A Streaming Multi-Threaded Model.

[11] 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 11th ACM Conference on Computing Frontiers*. ACM, 3.

[12] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. 2018. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro* 38, 2 (2018), 8–20.

[13] Eric S Chung, John D Davis, and Jaewon Lee. 2013. Linqits: Big data on little clients. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 261–272.

[14] Eric S Chung, James C Hoe, and Ken Mai. 2011. CoRAM: an in-fabric memory architecture for FPGA-based computing. In *FPGA*. ACM, 97–106.

[15] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491.

[16] James Coole and Greg Stitt. 2010. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 13–22.

[17] Tomasz S Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P Singh. 2012. From OpenCL to High-Performance Hardware on FPGAs. In *FPL*. IEEE, 531–534.

[18] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 2016. FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search. In *FPGA*. ACM, 105–110.

[19] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. 2017. ForeGraph: Exploring Large-Scale Graph Processing on multi-FPGA Architecture. In *FPGA*. ACM, 217–226.

[20] André DeHon, Yury Markovsky, Eylon Caspi, Michael Chu, Randy Huang, Stylianos Perissakis, Laura Pozzi, Joseph Yeh, and John Wawrzynek. 2006. Stream Computations Organized For Reconfigurable Execution. *Microprocessors and Microsystems* 30, 6 (2006), 334–354.

[21] Nina Engelhardt and Hayden Kwok-Hay So. 2016. GraVF: A Vertex-Centric Distributed Graph Processing Framework on FPGAs. In *FPL*. IEEE, 1–4.

[22] Suhaib A Fahmy, Kizheppatt Vipin, and Shanker Shreejith. 2015. Virtualized FPGA Accelerators for Efficient Cloud Computing. In *Cloud Computing Technology and Science, 2015 IEEE 7th International Conference on*. IEEE, 430–435.

[23] Tom Feist. 2012. Vivado Design Suite. *White Paper* 5 (2012), 30.

[24] Kermin Fleming and Michael Adler. 2016. The LEAP FPGA Operating System. In *FPGAs for Software Programmers*. Springer, 245–258.

[25] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. 2018. A Configurable Cloud-Scale DNN Processor For Real-Time AI. In *ISCA*. IEEE Press, 1–14.

[26] Phil Francisco et al. 2011. The Netezza data appliance architecture: A platform for high performance data warehousing and analytics.

[27] Gaël Guennebaud and Benoît Jacob and others. 2010. Eigen v3. http://eigen.tuxfamily.org.

[28] Chang Gao, Daniel Neil, Enea Ceolini, Shih-Chii Liu, and Tobi Delbruck. 2018. DeltaRNN: A Power-efficient Recurrent Neural Network Accelerator. In *FPGA*. ACM, 21–30.

[29] Tong Geng, Tianqi Wang, Ahmed Sanaullah, Chen Yang, Rui Xu, Rushi Patel, and Martin Herbordt. 2018. FPDeep: Acceleration and Load Balancing of CNN Training on FPGA Clusters. In *FCCM*. IEEE, 81–84.

[30] Intel. 2017. Intel FPGA SDK For OpenCL. https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html.

[31] Aws Ismail and Lesley Shannon. 2011. FUSE: Front-end User Framework for O/S Abstraction of Hardware Accelerators. In *FCCM*. IEEE, 170–177.

[32] Abhishek Kumar Jain, Suhaib A Fahmy, and Douglas L Maskell. 2015. Efficient Overlay Architecture based on DSP Blocks. In *FCCM*. IEEE, 25–28.

[33] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, et al. 2015. BlueDBM: An Appliance for Big Data Analytics. In *ISCA*. IEEE, 1–13.

[34] Cindy Kao. 2005. Benefits of Partial Reconfiguration. *Xcell journal* 55 (2005), 65–67.

[35] 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 *OSDI*. USENIX Association, 107–127.

[36] Robert Kirchgessner, Greg Stitt, Alan George, and Herman Lam. 2012. VirtualRC: a virtual FPGA platform for applications and tools portability. In *FPGA*. ACM, 205–208.

[37] Oliver Knodel and Rainer G Spallek. 2015. Computing framework for dynamic integration of reconfigurable resources in a cloud. In *2015 Euromicro Conference on Digital System Design*. IEEE, 337–344.

[38] Oliver Knodel and Rainer G Spallek. 2015. RC3E: Provision and Management of Reconfigurable Hardware Accelerators in a Cloud Environment. *arXiv preprint arXiv:1508.06843* (2015).

[39] Dirk Koch, Christian Beckhoff, and Guy GF Lemieux. 2013. An efficient FPGA overlay for portable custom instruction set extensions. In *FPL*. IEEE, 1–8.

[40] Jonas Krautter, Dennis RE Gnad, and Mehdi B Tahoori. 2018. FPGAhammer: Remote Voltage Fault Attacks on Shared FPGAs, suitable for DFA on AES. In *IACR Transactions on Cryptographic Hardware and Embedded Systems*. 44–68.

[41] Chris Lavin and Alireza Kaviani. 2018. RapidWright: Enabling Custom Crafted Implementations for FPGAs. In *FCCM*. IEEE, 133–140.

[42] Sheng Li, Hyeontaek Lim, Victor W Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G Andersen, O Seongil, Sukhan Lee, and Pradeep Dubey. 2015. Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-Value Store Server Platform. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 476–488.

[43] Yixing Li, Zichuan Liu, Kai Xu, Hao Yu, and Fengbo Ren. 2017. A 7.663-TOPS 8.2-W Energy-efficient FPGA Accelerator for Binary Convolutional Neural Networks. In *FPGA*. ACM, 290–291.

[44] Tao Lin, Chris Chu, Joseph R Shinnerl, Ismail Bustany, and Ivailo Nedelchev. 2013. POLAR: Placement based on novel rough legalization and refinement. In *ICCAD*. IEEE Press, 357–362.

[45] Cheng Liu, Ho-Cheung Ng, and Hayden Kwok-Hay So. 2015. QuickDough: a rapid FPGA loop accelerator design framework using soft CGRA overlay. In *FPT*. IEEE, 56–63.

[46] Enno Lübbers and Marco Platzner. 2009. ReconOS: Multithreaded Programming for Reconfigurable Computers. *ACM Transactions on Embedded Computing Systems* 9, 1 (2009), 8.

[47] Jason Luu, Jeffrey Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Nooruddin Ahmed, et al. 2014. VTR 7.0: Next generation architecture and CAD system for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems* 7, 2 (2014), 6.

[48] Ricardo Menotti, Joao MP Cardoso, Marcio M Fernandes, and Eduardo Marques. 2009. Automatic Generation of FPGA Hardware Accelerators Using A Domain Specific Language. In *FPL*. IEEE, 457–461.

[49] Microsoft. 2018. Real-time AI: Microsoft announces preview of Project Brainwave. https://blogs.microsoft.com/ai/build-2018-project-brainwave/.

[50] Tayo Oguntebi and Kunle Olukotun. 2016. GraphOps: A Dataflow Library for Graph Analytics Acceleration. In *FPGA*. ACM, 111–117.

[51] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. 2016. Energy Efficient Architecture for Graph Analytics Accelerators. In *ISCA*. IEEE, 166–177.

[52] M Akif Özkan, Oliver Reiche, Frank Hannig, and Jürgen Teich. 2016. FPGA-based Accelerator Design From A Domain-Specific Language. In *FPL*. IEEE, 1–9.

[53] Jongse Park, Hardik Sharma, Divya Mahajan, Joon Kyung Kim, Preston Olds, and Hadi Esmaeilzadeh. 2017. Scale-out Acceleration for Machine Learning. In *MICRO*. ACM, 367–381.

[54] Christian Plessl and Marco Platzner. 2004. Virtualization of Hardware-Introduction and Survey.. In *ERSA*. Citeseer, 63–69.

[55] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From High-Level Deep Neural Models to FPGAs. In *MICRO*. 17.

[56] Hayden Kwok-Hay So and Robert Brodersen. 2008. A Unified Hardware/Software Runtime Environment for FPGA-based Reconfigurable Computers using BORPH. *ACM Transactions on Embedded Computing Systems* 7, 2 (2008), 14.

[57] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. 2016. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In *FPGA*. ACM, 16–25.

[58] TACC. [n.d.]. Catapult - Texas Advanced Computing Center. https://www.tacc.utexas.edu/systems/catapult/.

[59] Hien Nguyen Van, Frédéric Dang Tran, and Jean-Marc Menaud. 2010. Performance And Power Management For Cloud Infrastructures. In *2010 IEEE 3rd International Conference on Cloud Computing*. IEEE, 329–336.

[60] Peter JM Van Laarhoven and Emile HL Aarts. 1987. Simulated Annealing. In *Simulated annealing: Theory and applications*. Springer, 7–15.

[61] Kizheppatt Vipin, Shanker Shreejith, Dulitha Gunasekera, Suhaib A Fahmy, and Nachiket Kapre. 2013. System-Level FPGA Device Driver With High-Level Synthesis Support. In *FPT*. IEEE, 128–135.

[62] Shuo Wang, Zhe Li, Caiwen Ding, Bo Yuan, Qinru Qiu, Yanzhi Wang, and Yun Liang. 2018. C-LSTM: Enabling Efficient LSTM using Structured Compression Techniques on FPGAs. In *FPGA*. ACM, 11–20.

[63] Jagath Weerasinghe, Francois Abel, Christoph Hagleitner, and Andreas Herkersdorf. 2015. Enabling FPGAs in hyperscale data centers. In *Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom), 2015 IEEE 12th Intl Conf on*. IEEE, 1078–1086.

[64] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs. In *DAC*. ACM, 29.

[65] Tobias Wiersema, Ame Bockhorn, and Marco Platzner. 2014. Embedding FPGA overlays into configurable systems-on-chip: ReconOS meets ZUMA. In *ReConFigurable Computing and FPGAs, 2014 International Conference on*. IEEE, 1–6.

[66] Yu-Liang Wu and Douglas Chang. 1994. On the NP-completeness of regular 2-D FPGA routing architectures and a novel solution. In *ICCAD*. IEEE Computer Society Press, 362–366.

[67] Xilinx. 2017. Vivado Design Suite User Guide Hierarchical Design. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug905-vivado-hierarchical-design.pdf.

[68] Xilinx. 2018. Vivado Design Suite User Guide Partial Reconfiguration. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug909-vivado-partial-reconfiguration.pdf.

[69] Yonghong Xu and Mohammed AS Khalid. 2005. QPF: efficient quadratic placement for FPGAs. In *International Conference on Field Programmable Logic and Applications*. IEEE, 555–558.

[70] Jialiang Zhang and Jing Li. 2017. Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network. In *FPGA*. ACM, 25–34.

[71] Mark Zhao and G Edward Suh. 2018. FPGA-based Remote Power Side-Channel Attacks. In *2018 IEEE Symposium on Security and Privacy*. IEEE, 229–244.

[72] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. 2017. Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs. In *FPGA*. ACM, 15–24.