EIF: A Mediated Pass-Through Framework for Inference as a Service

***Abstract*—In order to effectively provide INaaS (Inference-as-a-Service) in resource-limited cloud environments, two major challenges must be overcome: achieving low latency and providing multi-tenancy. This paper presents EIF (Efficient INaaS Framework), which uses a heterogeneous CPU-FPGA architecture to address these challenges via (1) *temporal multiplexing* that exploits the sparsity of neural-net models, (2) *spatial multiplexing* via software-hardware co-design virtualization techniques, and (3) *streaming-mode inference* which overlaps data transfer and computation. The prototype EIF is implemented on an Intel PAC (shared-memory CPU-FPGA) platform. For evaluation, 12 types of DNN models were used as benchmarks, with different size and sparsity. Based on these experiments, we show that in EIF, the temporal multiplexing technique can improve the user density of an AI accelerator from 2× to 6×, with marginal performance degradation. In the prototype system, the spatial multiplexing technique supports eight physical accelerators on one FPGA. By using a streaming mode based on a mediated pass-through architecture, EIF can overcome the FPGA on-chip memory limitation to improve multi-tenancy and optimize the latency of INaaS. To further enhance INaaS, EIF utilizes the MapReduce function to provide a more flexible QoS. Together with the temporal/spatial multiplexing techniques, EIF can support 48 users simultaneously on a single FPGA board in our prototype system. In all tested benchmarks, cold-start latency requires only approximately 5% of the total response time.**

***Index Terms*—Inference-as-a-Service, Mediated Pass-Through, CPU-FPGA platform**

I. Introduction

INaaS (Inference as a Service) [17] has received a great deal of attention from cloud providers due to the ability to abstract away low-level hardware details, while providing isolation between services, improved scalability, simplified management, and reduced costs.

Existing CPU [1], [6] and GPU [3], [12] INaaS solutions have significant latency limitations, largely due to cold starts (necessary loading of both the model and library processes before inference) that take 73% and 91% of the total response time [1], respectively. Using FPGAs is an attractive alternative due to the elimination of library loading times, while also providing high performance and low power. For instance, Amazon AWS EC2 F1 [9] provides the AI cloud service using FPGAs in the computing infrastructure. However, used in resource-limited cloud environments, existing FPGA solutions suffer from limited multi-tenancy in sharing an FPGA board [7], while still have the high cold start latency resulting from model loading.

In order to effectively provide INaaS (Inference-as-a-Service) in resource-limited cloud environments using FPGAs, two major challenges must be overcome: achieving low latency and multi-tenancy. This paper presents EIF (Efficient INaaS Framework), which uses a heterogeneous CPU-FPGA architecture to address these challenges via (1) *spatial multiplexing* via software-hardware co-design virtualization techniques. (2) *temporal multiplexing* that exploits the sparsity of neural-net models. (3) *streaming-mode inference* which overlaps data transfer and computation. One of the key contributions of EIF is the use of a combination of both temporal and spatial multiplexing to improve multi-tenancy. In Section IV, we describe how we support an efficient temporal multiplexing method, where threads are scheduled onto shared resources. This temporal multiplexing method leverages the concept of *Sparsity-Driven Multi-Thread Co-Execution* (SDMC), which allows multiple hardware threads to execute on the same resource simultaneously by exploiting the inherent sparsity inside a DNN model. Also in Section IV, we present a hardware/software co-design virtualization technique to support an efficient spatial multiplexing method, which are customized in EIF specifically for inference.

In addition to the multiplexing techniques, EIF uses a streaming mode based on a mediated pass-through architecture (detailed in Section IV). By doing so, EIF can overlap the data transfer and computation to overcome the on-chip memory limitation to improve multi-tenancy and optimize the model-loading latency suffered by existing FPGA-based INaaS. Also based on mediated pass-through architecture, EIF is equipped with the MapReduce function to support a more flexible QoS.

A prototype EIF has been implemented on an intel PAC card, which is a CPU-FPGA shared memory platform. With the spatial multiplexing technique, the EIF can support multiple DNN accelerators on one FPGA board. To perform DNN inference, the implemented EIF enhances the commonly used systolic array as DNN accelerator with the customized temporal multiplexing mechanism. Due to the temporal multiplexing technique, a single DNN accelerator can simultaneously support from two to six DNN user requests. Compared with a dedicated systolic array for one user, the performance degradation is only from 0.5% to 19.5%. And the throughput on one accelerator board can be improved from 15.7× to 40.8×. Using the streaming technique, the prototype EIF greatly reduced the on-chip memory limitation to improve multi-tenancy, supporting up to 48 users simultaneously on one FPGA board. Compared with the traditional INaaS platform, on which cold-start occupies a non-trivial part of total response time, the cold start in EIF only occupies around 5% of the overall latency.

The remainder of this paper is organized as follows. Section

II provides background for this study. An overview of the EIF architecture is given in Section III. EIF adopts a host-device architecture, with the CPU being the host and the FPGA accelerator being the device (accelerator). Section IV presents the design details of the key modules of EIF: Container and Hypervisor modules in the CPU and the accelerator modules (Data Gathering Unit (DGU) and Computing Unit (CU)) in the FPGA. In Section V, the experimental setup is given, followed by detailed experimental results and evaluation. Related works are discussed in Section **??**, followed by the conclusions in Section VI.

II. BACKGROUND

A. FPGA Virtualization

Through virtualization techniques, an FPGA can be multiplexed spatially [10], [23] and temporally [5], [13]. Spatial multiplexing allows different accelerators to simultaneously occupy the same FPGA. Temporal multiplexing allows a hardware infrastructure to be shared by multiple virtual machines (VMs) at different time slices.

However, the current virtualization techniques do not solve the major challenges faced by INaaS, mainly because these techniques are not customized for inference tasks.

First, a very limited number of inference tasks can be supported on one FPGA board because of the on-chip memory limitation. For example some large DNNs such as DCGAN [15] and ZFNet [22], even cannot fit fully in on-chip memory. The limitation of FPGA on-chip memory precludes the use of existing techniques in multi-tenancy-oriented services for datacenter applications.

Furthermore, for INaaS, previous virtualization works have too many redundant operations, such as the memory segmentation in Optimus [10], which adds unnecessary overhead to the entire INaaS system. One of EIF's important contribution is to provide customized, efficient hardware-software co-design virtualization techniques for INaaS, while with the marginal overhead.

Finally, although the above techniques achieve high efficiency and low overhead in spatial multiplexing, temporal multiplexing techniques such as preemptive temporal multiplexing are not suitable for inference. The reason is most temporal multiplexing techniques [16] have a detrimental effect on the performance of an individual INaaS request, which is highly latency sensitive. In EIF, we will introduce a new customized temporal multiplexing technique which will cause marginal overhead with increase of user number.

B. Mediated Pass-through Architecture

An mediated pass-through architecture [10], [21] provides two methods of communication between devices: *performance critical* operations are passed through and directly access hardware resources, while *privileged* operations are trapped-and-emulated to provide isolation between guests and the hardware infrastructure.

Moreover, a mediated pass-through architecture is a good solution to solve the inherent problem of INaaS, shortage of resources inside an individual container. In such architecture, computing-intensive calculations can be off-loaded to a device with strong computing capability (accelerators such as a GPU, FPGA, or TPU). There are many techniques that use mediated pass-through architecture to support VM-based services using GPU, such as GPUvm [18]. However, these techniques suffer from the cold start issue, when used to support INaaS. Compared with FPGAs-supporting INaaS, GPU-based INaaS has a very high cold start penalty because it requires to load the model and the necessary library beforehand, which becomes a major part of the total response time. To the best of our knowledge, there is currently no work on utilizing FPGA-based mediated pass-through architecture to support INaaS

C. Sparsity in DNN Models

Sparsity in the DNN model is defined as the fraction of zeros in the layer's weight and input. In practical DNN models, many of input and weight data turn out to be zero [11]. The corresponding multiplications and additions related these zero values do not contribute to the final result and can be regarded as ineffective. The reason of the sparsity is from the nature and structure of DNN, like the activation function of CNN. Figure 1 shows the average total percentage of multiplication operands that are ineffective in different DNN models. This fraction varies from 37%, to up to 50% and the average across all networks is 44%. Many existing sparse NN accelerators [2] solve the inefficiency from the sparsity by compressing data or eliminating ineffective computation. In EIF, we utilize this the sparsity in DNN models to develop a temporal multiplexing method to allow multiple inference tasks can share one computing hardware.

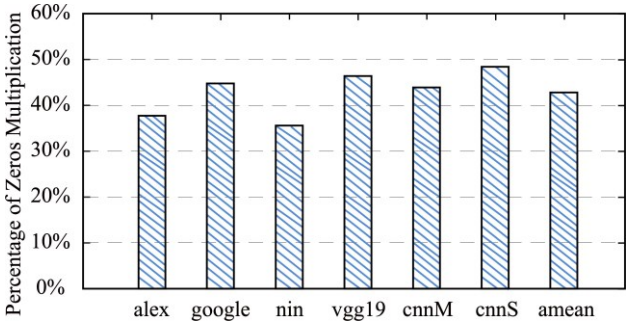


Fig. 1: Fractions of ineffective multiplication in DNN models.

III. EIF OVERVIEW

EIF targets a use case in which cloud providers offer INaaS based on the CPU-FPGA Shared-Memory platform. To provide an unprecedented level of multi-tenancy and low latency for FPGA-based INaaS, several goals should be achieved through hardware and software co-design:

Multi-tenancy. EIF targets to achieve the highest level of multi-tenancy within one FPGA device. To this end, the EIF framework must achieve both temporal and spatial multiplexing for FPGA. For these purposes, we must solve the following problems: (1) limited on-chip memory resource, and (2) efficient method of temporal and spatial multiplexing.

Isolation. EIF aims to provide enough security for the system: isolation between user-end and infrastructure. Because the user is untrusted, in EIF framework, the containers cannot directly access any type of infrastructure hardware resource.

Efficiency. EIF targets to provide INaaS with low latency. For achieving this goal, EIF must utilize the high computing ability from FPGA accelerator and optimize the high cold start problem which is very common in current INaaS platforms.

To achieve the goals and overcome the challenges we mentioned above, our EIF architecture is mainly derived by the following three observations:

First, the FPGA virtualization technique [19] can provide isolation between users and infrastructure, and improve the multi-tenancy of the FPGA board. FPGA virtualization technique [21] usually virtualize the FPGA with spatial multiplexing and temporal multiplexing. The techniques for spatial multiplexing have already been proven efficient enough. But traditional temporal multiplexing technique from the operating system level, like preemptive multitasking, is detrimental to the response latency of DNN inference which is a very latency-sensitive job. Thus, in EIF, we design a customized and low-overhead temporal multiplexing method with the deployment of sparsity inside the DNN model with the SDMC mode [8].

Second, for a higher level of multi-tenancy for FPGA support INaaS, we must overcome the restriction imposed by relatively limited on-chip memory compared to the model size of an AI inference task. A DNN model can be thought of as a series of several sub-model. If we can perform a DNN task in a streaming mode, we just need to keep the parameters of a sub-model in on-chip memory instead of the entire DNN model [4].

Third, for an inference task, the time to prepare the operation and store the result is usually as long as the time to compute the required operations. If data transmission/reception and computing operation are independently processed in heterogeneous processors, the response time can be optimized by overlapping these two processes [4].

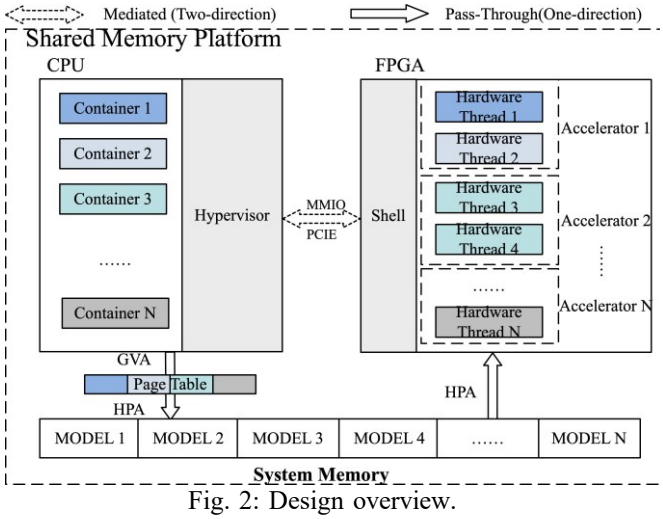


Fig. 2: Design overview.

The overall architecture of the EIF system, as shown in Figure 2, is guided by the design principles discussed above. It is implemented on a shared-memory platform with a CPU and an FPGA.

There are two major sets of components on the CPU side: the containers and Hypervisor. The design follows a mediated pass-through [10], [21] architecture in which the containers (of CPU) are responsible for instructing the accelerators, and managing data transfer between the end-users and the underlying accelerators; whereas the Hypervisor of EIF traps all control operations (Memory-Mapped IOs, MMIOs) from the containers to redirect these operations to the appropriate physical accelerator.

There are also two major sets of components on the FPGA side: a Shell and a set of accelerators. The Shell component is a reserved portion of the FPGA. The board manufacturer (e.g., Intel, Xilinx) provides the Shell, which serves as the IO interface for the FPGA. At the beginning of configuring the entire framework, the FPGA is configured with a fixed number and types of accelerators through the Shell. And the infrastructure information of configured FPGA will be kept in the Hypervisor.

IV. Design of Key EIF modules

In this section, we will describe the design details of each of the key components of the EIF architecture and how they achieve the goals described in Section III. In Subsection A, we will describe the design details of the EIF components implemented in the CPU. In Subsection B, the design details of the EIF components implemented in the FPGA will be described.

A. Design of EIF Components in the CPU

EIF follows a host-device architecture, with the CPU being the host. As stated, there are two major sets of components on the CPU side: A set of containers and a Hypervisor, the design of which follows a Mediated Pass-through Architecture. The containers host INaaS services for user requests, and the Hypervisor virtualizes the hardware for spatial multiplexing FPGA and isolates the containers (user-end) and FPGA infrastructure.

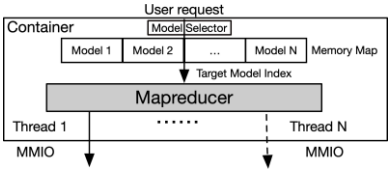


Fig. 3: Designs in container.

a) Container: Figure 3 shows the designs of a container of the EIF framework. The incoming user requests contain the following information: required model type, whether it needs MapReduce, and if so, the configuration of MapReduce. Note that how the upper level of the scheduling system gets this information is beyond the scope of this work. The three main parts of a container of the EIF system are: Model Selector, Host Memory Map, and MapReducer. When a new user request arrives, the Model Selector will select the target model that is specified in user request by searching all the models

stored in the memory. Each container includes a Host Memory Map, containing information that which data inside the host memory is the requested model data of user request from the perspective of GVA (Guest Virtual Address). Containers will inform these GVAs to the accelerator for further translation to HPA (host physical address). The container also contains a MapReduce function, through which a user request can be mapped into multiple hardware threads for supporting flexible QoS. In this way, one user request can be concurrently executed by multiple hardware threads.

b) Hypervisor: After processing the user requests, the containers send control operations to the Hypervisor, as shown in Figure 4. The Hypervisor traps control operations (MMIOs) from containers and redirects these operations to target the available threads of an accelerator. The Hypervisor will add an offset to this MMIO, and hardware logic will use this offset to address the target physical accelerator. As shown in Figure 4, the major part of the Hypervisor is a Hardware Monitor. The Hardware Monitor is used for arbitrating the redirected operations to the multiplexed FPGA resources. The offset added by the Hypervisor is provided by the Hardware Monitor based on the availability of hardware infrastructure. The Hardware Monitor keeps track of all the information including the total number and availability of infrastructure hardware threads and redirects MMIO operations to available hardware threads. As shown in Figure 4, in the EIF system, we implement the Hardware Monitor as a MUX tree. The root level of the MUX tree is the whole FPGA board. The first level of the MUX tree represents the accelerators. The second level represents the threads. For the scheduling policy of the MUX tree, there are two choices: weighted scheduling policy and unweighted scheduling policy (round-robin). Whether deploying weighted or unweighted scheduling policy depends on whether the configuration of FPGA is homogeneous or heterogeneous, which we will discuss in Section V.

In summary, based on the mediated pass-through architecture (CPU (Container, Hypervisor) as the host, FPGA accelerator as the device), the computing workload are offloaded from containers into the FPGA side. And through this, we improve the response time by leveraging the high computing ability of accelerators. And through hypervisor design, spatial multiplexing can be achieved to improve multi-tenancy on the FPGA board.

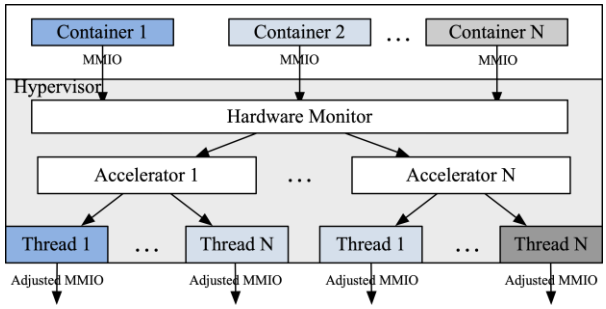


Fig. 4: Details of Hypervisor.

B. Design Principle of Temporal Multiplexing

As we mentioned in Section II, the common software temporal multiplexing methods have detrimental overhead to individual user request. In this subsection, we will introduce the design principle of EIF to support an efficient temporal multiplexing method from hardware perspective. Based on that, one DNN accelerator can simultaneously support multiple DNN requests as hardware threads.

We exploit DNN's sparsity feature to deploy an *Sparsity-Driven Multi-Thread Co-Execution Mode* (SDMC) [8] to achieve an efficient temporal multiplexing method inside the accelerator. The basic idea is that if one zero exists in either operand of a multiplication, this multiplication can be discarded and directly set the output as zero. So when two threads share the same computation unit (multiplier), only the effective (non-zero) thread will need to be calculated. The hardware thread in EIF is akin to the thread concept in software operating system, where multiple threads share the same hardware unit. If two threads both need to be calculated, the unexecuted one will be kept in the FIFO buffer. When two threads are both ineffective (both have zero-operand in other words), the first entry on the FIFO buffer will be issued and executed. At a certain point in time, only multiplication of one thread will be selected by the multiplexer and processed by the hardware unit. Through this hardware method, close-to- zero overhead temporal multiplexing can be achieved between multiple hardware threads.

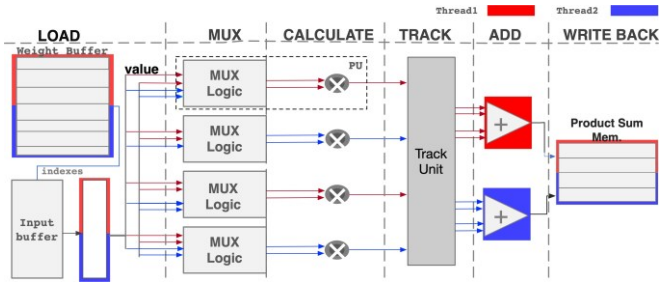


Fig. 5: Dataflow of PEs based on SDMC mode.

Figure 5 shows how two threads, one multiplier, and four processing units pipeline (PU) works, based on the Sparsity-Driven Multi-Thread Co-Execution mode. The four stages of the PE are as follows:

- Load: Each thread (thread 1, red; thread 2, blue) loads the weight and corresponding input from the corresponding part of on-chip memory.
- Branch: This stage determines the effective thread(s), there are four combinations, that are 00, 01, 10, and 11. When two threads are both ineffective (00), the first entry from the FIFO buffer will be loaded and executed. When there are two threads competing for the multiplier (11), the first thread will be executed, and the second thread will be kept into FIFO. Either 01 or 10 implies there is only one effective thread within the two threads and the effective one will be executed while the ineffective one will be ignored and directly outputs zero.

- Calculate and Track: The operands of effectual thread issued in BRANCH stage will enter the multiplier for calculating, while tracking the thread tag.
- Add and Writeback: Add the partial sums of each thread together and write the result back to buffer. This is the result of one stride move in MAC (multiply–accumulate) operation.

Unlike traditional methods of temporal multiplexing from software operating level, hardware temporal multiplexing with SDMC does not support traditional preemption. Fairness between threads is ensured by the computation effectiveness of threads.

In summary, based on the SDMC mode, EIF achieves the temporal multiplexing inside the DNN accelerator. Traditional DNN accelerator only support one user request. With the enhancement of SDMC mode, one DNN accelerator can support multiple hardware threads as figure 4 shows.

C. Design of Accelerator Components in the FPGA

The FPGA accelerator plays the role of device in the EIF host-device architecture, accelerating the computation offloaded by the CPU. Figure 6 shows the overall structure of the accelerator. The proposed architecture of the accelerator consists of two heterogeneous processors: Data Gathering Unit (DGU) and Computing Unit (CU). The DGU of the accelerator is mainly responsible for receiving model data and for controlling the CU. The CU performs computation tasks and reports the results to the DGU. DGU and CU can work independently and overlap in parallel the processes of model loading and inference to perform the inference in a streaming mode. With the streaming mode, the CU can start performing inference with the first slice of model data, without waiting the whole model loaded into the on-chip memory (weight buffer). The cold start from model loading is also mitigated from loading whole model to loading the first model slice. And during the processing of performing inference, only the current-performed slice of model data is needed to be kept inside the weight buffer. As shown in Figure 6, the major parts of DGU are Shadow Page Table (SPT), DMA Engine, and Data Control Unit. The CU is comprised of a systolic array of PE's, along with the Weight and Activation Buffers.

a) Shadow Page Table: EIF deploys SPTs (Shadow Page Tables) in the hardware side for translating the GVA (guest virtual address) into HPA (host physical address). This HPA is necessary for the DMA Engine to retrieve the target DNN model. Each accelerator is equipped with an individual SPT for an effective translation from GVA to HPA once it receives GVAs of the target model through MMIO from the CPU.

b) DMA Engine: To efficiently retrieve models from the host memory, each accelerator needs the ability to issue DMA individually. To this end, we equip each accelerator with an individual DMA Engine. The accelerators will use the SPTs (Shadow Page Tables) to translate GVA (guest virtual address) into HPA (host physical address). With these HPAs, the DMA Engine can issue DMA requests and retrieve the target model via a streaming mode. In this way, the number of accelerators

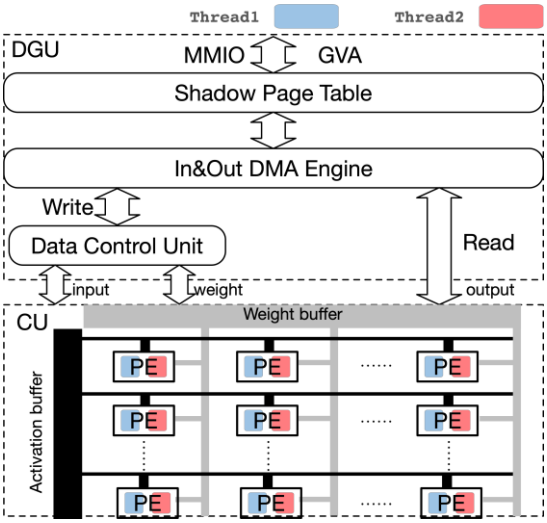


Fig. 6: Overall structure of an accelerator.

that can be built on an FPGA is only limited by PCI-E resources.

c) Data Control Unit: The Data Control Unit (DCU) is responsible for requesting the input data through the DMA engine, filling the input buffer of the activation buffer, draining out the output buffer of the activation buffer, and generating the output signal for containers.

d) Weight Buffers: We deploy the double buffering technique for executing the entire inference task in a streaming mode. There are two levels of buffers for processing a model. When the first-level buffer is almost drained out, the data kept in the second-level buffer will be loaded into the first-level buffer for the next calculation in advance. And next sub-model kept in memory will be loaded into the second-level buffer during the timing of calculation and be ready for fill into the first-level buffer.

e) Architecture of Computing Unit (CU): We deploy a systolic array (SA) based GEMM (general matrix-matrix multiplications) accelerator as the Computing Unit of EIF's accelerators. For state-of-the-art DNN models, fully-connected (FC) layers, multilayer perceptron (MLPs), and Convolutional Layer can be mapped into GEMM operations naturally, and be executed in parallel. The PE (Processing Element) of the systolic array design follows the SDMC mode as we described. With that, the systolic array (as the DNN accelerator) can simultaneously perform the GEMM operations from multiple threads. The total number of co-executed threads in one systolic array should be preconfigured by cloud provider. The provider can implement different configurations of systolic arrays, e.g., SA with four threads or six threads, to adapt different types of inference tasks, which we will discuss in Section V.

Figure 7 shows an example of the Computing Unit, which contains 32 PEs and performs two threads concurrently. In our implemented Systolic Arrays, there are four rows of PEs, and each row contains eight PEs. A Multicast bus connects PEs on the same row for sharing filter weights among PEs inside the same row (green lines in Figure 7). Since each row of

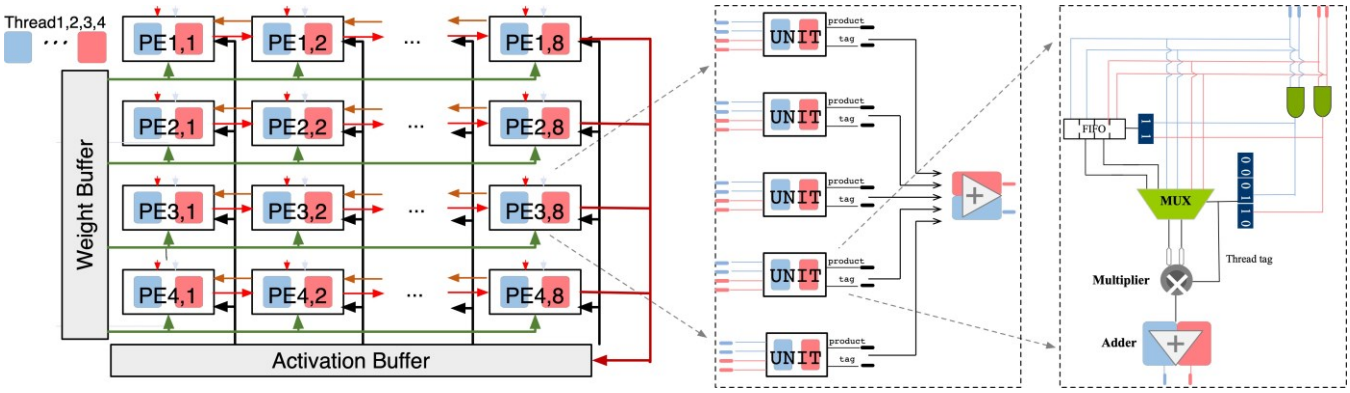


Fig. 7: Overall structure of an example Computing Unit Architecture. The CU has an architecture of systolic array. Each PE processes tensor computing instead of a single MAC. Our PE design improves the usage efficiency of accumulator

PEs calculates partial sums for the same output channel, a bus connecting inter-PE in the same row is added (red lines).

In summary, with the streaming technique by overlapping operations in DGU and CU, the FPGA accelerators of EIF optimize the problem of high cold start and solve the on-chip memory limitation to support more DNN requests on one-chip board. And with the temporal multiplexing technique based on the SDMC mode, EIF furtherly improves the level of multi-tenancy on the FPGA board.

V. Evaluation

We evaluate EIF mainly regarding the following metrics:

- Efficiency: What is the overhead of temporal multiplexing and spatial multiplexing. Are multiplexing techniques deployed by EIF framework efficient enough?
- Multi-tenancy: Through the streaming-mode inference, EIF framework successfully overcomes the on-chip memory limitation. Then how many user requests can be simultaneously supported on a single FPGA?
- Cold Start and response time: One of the major contributions of EIF framework is mitigating the ”cold start” of INaaS. Compared with other frameworks of INaaS, what are the advantages of EIF in terms of ”cold start”? Regarding overall response time, what kind of speeding-up can the EIF framework provide compared with other INaaS frameworks.

A. Experimental Setup

- Hardware: We implemented the prototyped EIF on the Intel PAC with Intel Arria 10 GX FPGA. Unlike CPU or GPU, the EIF implementation deploys 16-bit fixed-point arithmetic. For the computing logic inside the accelerator, the clock frequency is 100 MHZ. And system clock for AXI protocol and reading/writing data from or to DRAM is 250 MHZ.
- Software: We implemented the hypervisor based on KVM (Kernel-based Virtual Machine). And we add the function of Hardware Monitor into the hypervisor.
- Baseline: We compared the performance of the EIF-based INaaS with CPU-based INaaS and GPU-based INaaS. Existing GPU and CPU based INaaS usually overlook the problem of cold start and have not include the streaming (pre-fetching) technique to overlap the model loading and computing. For fair comparison, based on existing CPU/GPU methods [17], we upgrade our CPU and GPU INaaS baseline with the streamming techinque. And because EIF causes no library loading time, our baseline CPU, GPU baseline also are based on the assumption that the library (e.g., Pytorch, Tensorflow) is container-resident. For GPU-based INaaS, the model will load from host memory to GPU local memory. For CPU- based INaaS, the model will be directly used by data in the host memory. We chose NVIDIA GeForce TITAN RTX as our GPU platform and Intel(R) Xeon(R) W-2255 CPU as the CPU platform. The baseline of CPU-based INaaS was evaluated on the Docker container [14]; and the baseline of GPU-based supporting INaaS was through the NVIDIA-Docker [20].
- Benchmarks: We deployed 12 different AI inference tasks on EIF, in Table I, as benchmarks. We tested overhead for both temporal multiplexing and spatial multiplexing. In Table I, the ”Weight Size” of a model is the model size for GPU-based and CPU-based INaaS. The ”.bin file” is the .bin input file for EIF. The .bin file is created under a certain order specified by EIF. Since we are targeting INaaS, our experiments use a batch size of one.

TABLE I. Parameters of benchmarks

Model	Weight Size	bin file	Sparsity	Net size
DCGAN 1-3	248 MB	262.6 MB	70%, 60%, 50%	large
DARKNET 1-3	459 MB	486.2 MB	70%, 60%, 50%	large
LeNet 1-3	19.1 KB	19.9 KB	70%, 60%, 50%	small
ZFNET 1-3	75.5 MB	28.4 MB	70%, 60%, 50%	middle

B. Evaluation of Temporal Multiplexing

The overhead of temporal multiplexing is mainly stemmed from inter-threads waiting time caused by preemptions of hardware resources between threads. In order to test the overhead of temporal multiplexing at different accelerator configurations under the different task types, we select AI models with different sparsity levels, as listed in Table I, as our benchmarks.

The results in Figure 8 show the comparison of 12 models, in Table I, to the dedicated execution of one thread (response time is normalized as one, no inter-threads waiting time).

For each model, there are three bar graphs, illustrating two, four, six threads and their normalized response time overhead (percentage of slowing down). With two-threads accelerator, all types of tasks exhibited close-to-zero overhead. Even with a six-threads accelerator, no tasks will exceed 20% overhead. Through the overall results shown in Figure 8 and model information in Table I, we found that response time overhead is highly correlated with the sparsity of inference task models and the amount of co-executed hardware threads. For lower overhead, matching tasks with adequate accelerator configuration is non-trivial. In order to adapt various levels of tasks with different sparsity, for the provider, it is highly recommended to deploy a heterogeneous configuration of FPGA. That is, configure an FPGA board as a set of accelerators with different numbers of threads. In addition, for the scheduling policy in hypervisor, instead of deploying round-robin policy, a weighted scheduling policy should be deployed. The weight for scheduling is according to sparsity. Depending on this sparsity, the scheduling policy should schedule the task to an adequate accelerator.

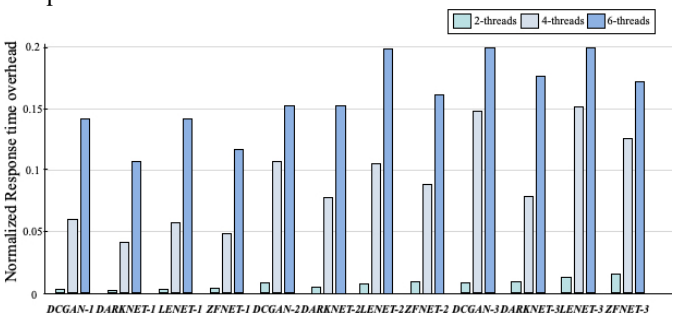


Fig. 8: Response time overhead compared with normalized response time under accelerators with different co-executing threads configuration. All these benchmarks are executed without mapreduce acceleration.

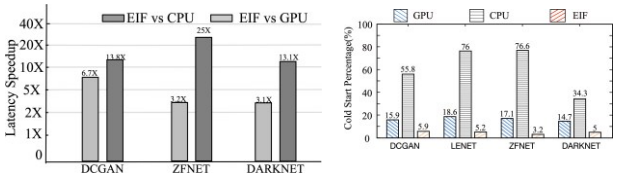
C. Evaluation of Spatial Multiplexing

For the overhead of spatial multiplexing, we evaluated the overhead of virtualization technique from software and hardware, our experiment shows the overhead of redirecting MMIOs at software the side is close-to-zero. Regarding the scalability of the EIF framework, we tested the range of the number of users that can be supported on one board. To avoid DMA congestion, we have not tested the maximum number of accelerators. For our implemented EIF framework, we built eight accelerators, and each accelerator can support up to six hardware threads. In this way, 48 guest requests can be supported on a single FPGA board.

D. Evaluation of Latency Breakdown

We evaluate the latency benefit of EIF from the FPGA-accelerator-based INaaS, include the overall response time and optimization of cold start. We compared the response time of inference tasks respectively on GPU-based INaaS, CPU-based INaaS, and EIF-based INaaS. The response time comprises model loading time, computing time. Figure 9(a) shows the normalized speedup performance among GPU-based INaaS, CPU-based INaaS, and EIF. Since we conducted an inference task that has a batch size one, latency-optimized CPU outperforms throughput-optimized GPU. For EIF, the response time is the speedup of tasks running on a dedicated accelerator, without temporal multiplexing. LENET is too small and will not be the benchmark for speedup evaluation. At 100MHZ computing clock rate, EIF outperforms CPU-based INaaS and GPU-based INaaS from 3.1× to 6.7×, and from 13.1× to 25×, respectively.

For the cold start (from model loading), GPU and CPU require loading at least model data of one layer to start the computation, while the EIF can start computing by loading model data of the first slice. Figure 9(b) shows the cold-start percentage of the total response time over these benchmarks. For EIF, it can be concluded that the problem of high cold start is largely optimized. The percentage of cold start only takes around 5% of total response time. And the values of cold start are stable between different DNN models.



(a) Speedups of EIF under dedicated accelerator, no temporal multiplexing (b) Percentage of cold start in overall response time

Fig. 9: Performance evaluation of EIF

E. Evaluation of Resource Utilization

Table II summarizes resource utilization by total 48 threads EIF (Eight homogeneous, six-threads accelerators, 16 bits data) from Quartus synthesis. In EIF, each thread has a double- buffer for model data and an activation buffer for reading input and writing output data. The activation buffer is actually constructed using two buffers: one for input and one for output to avoid overwriting. The activation layer is built by a lookup table IP core. And our implemented EIF utilizes the existing IP core for PCI-E DMA transmission. Each accelerator has a DMA channel as the individual DMA engine.

TABLE II. Resource utilization of 48-threads EIF

Resource type	Utilization	Available
Logic utilization	148,800	251,680
Total LABs	18,032	25,168
Block RAM	13,762,560	43,642,880
DSP Blocks	640	1,687

F. Compared with modern FPGA virtualization techniques.

Recently, Optimus [10] is the first work to virtualize the Shared-Memory FPGA platform. This work targets virtualizing a CPU-FPGA Shared-Memory platform to support multiple uncertain accelerators. Unlike Optimus, EIF targets provide virtualization for a certain type of FPGA accelerator. In INaaS task, EIF mainly outperforms Optimus because EIF is fully customized for INaaS. Assuming that basing on same structure of systolic array as the accelerator, EIF can shows advantages in temporal multiplexing. EIF achieve the temporal multiplexing by developing the sparsity of DNN models. The overhead of temporal multiplexing, as shown in figure 8 is marginal. But in Optimus, which deploy the

temporal multiplexing mechanism from software operating system level, e.g., Round-Robin Scheduling, the overhead of temporal multiplexing will approximately linearly increase with more users on one accelerator.

VI. CONCLUSION

This paper presents EIF, a mediated pass-through architecture for FPGA-based INaaS. EIF provides both spatial and temporal multiplexing to achieve high tenant density on a single CPU-FPGA device. EIF further optimizes INaaS by overlapping data transmission with computation to decrease the latency. Our experiments show that (1) up to 48 user requests can be simultaneously supported on one FPGA board.
(2) The overhead of spatial multiplexing is close to zero, and temporal multiplexing is very effective. (3) Compared with GPU-based INaaS, EIF can achieve from 13.1× to 25× speeding up in response time. And compared with CPU-based INaaS, EIF can achieve from 3.1× to 6.7× speeding up in response time. (4) EIF optimizes the cold start only occupy around 5% of total response time.

REFERENCES

[1] A. Bhattacharjee, A. D. Chhokra, Z. Kang, H. Sun, A. Gokhale, and G. Karsai, "Barista: Efficient and scalable serverless serving system for deep learning prediction services," in *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2019, pp. 23–33.

[2] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[3] J. Hu, A. Bruno, B. Ritchken, B. Jackson, M. Espinosa, A. Shah, and C. Delimitrou, "Hivemind: A scalable and serverless coordination control platform for uav swarms," *arXiv preprint arXiv:2002.01419*, 2020.

[4] J. Jo, S. Cha, D. Rho, and I.-C. Park, "Dsip: A scalable inference accelerator for convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 53, no. 2, pp. 605–618, 2017.

[5] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach, "Sharing, protection, and compatibility for reconfigurable fabric with amorphos," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 107–127.

[6] M. R. D. Kodandarama, M. D. Shaikh, and S. Patnaik, "Serfer: Serverless inference of machine learning models."

[7] J. Lallet, A. Enrici, and A. Saffar, "Fpga-based system for the acceleration of cloud microservices," in *2018 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*. IEEE, 2018, pp. 1–5.

[8] C. Liu, K. Li, M. Song, J. Zhao, K. Li, T. Li, and Z. Zeng, "Coexe: an efficient co-execution architecture for real-time neural network services," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[9] M. Lorusso, D. Bonacorsi, D. Salomoni, R. Travaglini, D. Michelotto, D. C. Duma, and P. Veronesi, "Accelerating machine learning inference using fpgas: the pynq framework tested on an aws ec2 f1," 2022.

[10] J. Ma, G. Zuo, K. Loughlin, X. Cheng, Y. Liu, A. M. Eneyew, Z. Qi, and B. Kasikci, "A hypervisor for shared-memory fpga platforms," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 827–844.

[11] X. Ma, F.-M. Guo, W. Niu, X. Lin, J. Tang, K. Ma, B. Ren, and Y. Wang, "Pconv: The missing but desirable sparsity in dnn weight pruning for real-time execution on mobile devices," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 04, 2020, pp. 5117–5124.

[12] D. M. Naranjo, S. Risco, C. de Alfonso, A. Pe´rez, I. Blanquer, and G. Molto´, "Accelerated serverless computing based on gpu virtualization," *Journal of Parallel and Distributed Computing*, vol. 139, pp. 32–42, 2020.

[13] M. Paolino, S. Pinneterre, and D. Raho, "Fpga virtualization with accelerators overcommitment for network function virtualization," in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2017, pp. 1–6.

[14] A. M. Potdar, D. Narayan, S. Kengond, and M. M. Mulla, "Performance evaluation of docker container and virtual machine," *Procedia Computer Science*, vol. 171, pp. 1419–1428, 2020.

[15] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv preprint arXiv:1511.06434*, 2015.

[16] R. V. Rasmussen and M. A. Trick, "Round robin scheduling–a survey," *European Journal of Operational Research*, vol. 188, no. 3, pp. 617–636, 2008.

[17] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: A of opportunities, challenges and applications," 2020.

[18] Y. Suzuki, S. Kato, H. Yamada, and K. Kono, "Gpuvm: Gpu virtualization at the hypervisor," *IEEE Transactions on Computers*, vol. 65, no. 9, pp. 2752–2766, 2015.

[19] A. Vaishnav, K. D. Pham, and D. Koch, "A survey on fpga virtualization," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 131–1317.

[20] P. Xu, S. Shi, and X. Chu, "Performance evaluation of deep learning tools in docker containers," in *2017 3rd International Conference on Big Data Computing and Communications (BIGCOM)*. IEEE, 2017, pp. 395–403.

[21] Y. Xu, J. Yao, Y. Dong, K. Tian, X. Zheng, and H. Guan, "Demon: An efficient solution for on-device mmu virtualization in mediated pass-through," in *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2018, pp. 57–70.

[22] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I 13*. Springer, 2014, pp. 818–833.

[23] J. Zhang, Y. Xiong, N. Xu, R. Shu, B. Li, P. Cheng, G. Chen, and T. Moscibroda, "The feniks fpga operating system for cloud computing," in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, 2017, pp. 1–7.