TAPU: A Transmission-Analytics Processing Unit for Accelerating Multifunctions in IoT Gateways

Huanghuang Liang, Qianlong Sang, Chuang Hu, *Member, IEEE*, Yili Gong, *Member, IEEE*, Dazhao Cheng, *Senior Member, IEEE*, Xiaobo Zhou, *Senior Member, IEEE*, and Yu Wang, *Fellow, IEEE*

***Abstract*—Internet of Things (IoT) gateways integrate various sensors and compute initial decisions before transmitting data to the cloud for further processing. As the functions they need to support become increasingly complex, gateways must upgrade their hardware. Network functions (NF) and video analytics (VAs) are two typical examples of hardware requirements: NFs need specialized hardware accelerators, while VAs need parallel processing power. However, gateways are typically constrained by factors, such as power, size, and cost, leading to a need to multiplex functions and minimize hardware overprovisioning. This article proposes a novel accelerator, the transmission-analytic processing unit (TAPU), which uses multi-image FPGA to accelerate VAs and NFs for IoT gateways. We preconfigure one image for VAs and one image for NFs, then multiplex the FPGA resources in the time dimension. The TAPU system design requires both hardware and software revisions. In the hardware design, we discuss our considerations on hardware choice and present a new abstraction of hardware functions to overcome the challenge of application development on different multi-image FPGAs. For the software, we develop a fully functional TAPU system to adapt to dynamic network and VAs workloads. Our evaluation shows that TAPU utilization can reach 92%, considerably increasing VAs and network processing throughput over the current approach. We further evaluate TAPU through two case studies that support a campus traffic monitoring system and an office surveillance system, demonstrating excellent performance improvement and low overhead.**

***Index Terms*—FPGA offloading, hardware acceleration, Internet of Things (IoT) gateways, network functions (NFs), video analytics (VAs).**

I. Introduction

INTERNET of Things (IoT) gateway is emerging as a critical component in connecting legacy and next-generation devices to the Internet [1]. Acting as a bridge between devices and the cloud, the IoT gateway integrates protocols for networking, performs edge analytics on the data, and facilitates a secure data flow between edge devices and the cloud. With the rapid expansion of the number of connected devices (from 6.1 billion in 2017 to 14.4 billion in 2022 [2]), the influx of data, and the pressing need for data security, today's IoT gateways must be able to support an ever-growing range of functions. For instance, it must handle networking protocols to ensure the secure transmission of critical data. Additionally, video analytics (VAs) must be supported. For instance, in-situ cameras have been installed in smart homes to identify and avert falls among the elderly population [3]. To reduce cellular costs, transmission latency, and privacy issues, it is necessary to execute VAs within the IoT gateway rather than relaying all videos to the cloud.

As the functions of the IoT gateways become increasingly complex, the current generation faces the challenge of upgrading their hardware. To fulfill the fundamental requirements of first-generation IoT gateways, which are designed to facilitate communication protocol compatibility and device management, a CPU alone is sufficient. However, the second generation of IoT gateways necessitates regular network functions (NFs) and VAs, necessitating a CPU and GPU combination to meet the computational requirements, as illustrated in Fig. 1(a). For instance, GPU configurations are available in Dell Edge Gateways for IoT [4] and ADLINK IoT gateway solutions [5]. Nevertheless, the current generation of IoT gateways necessitating advanced NFs, such as data encryption, decryption, and data compression, results in a CPU overload and consequent low utilization of the GPU, as shown in Fig. 1(b). Specifically, the execution of advanced software NFs requires a significant amount of CPU resources, leaving limited or no CPU resources for data preprocessing of VAs, resulting in low GPU utilization.

A viable solution to address the limitations mentioned above is to offload VAs and advanced NFs to a hardware accelerator, as illustrated in Fig. 1(c). This accelerator should be able to provide parallel processing power (e.g., GPU) for VAs and should also be programmable and customizable to support specialized NFs. However, customizing a dedicated hardware accelerator for each function can be costly. In addition, it may result in low utilization and larger size, which is not ideal for IoT gateways due to their stringent constraints in terms of cost and size. Therefore, what is needed is a multiplex-enabled accelerator for VAs and NFs that is both programmable and customizable while also being cost-effective and compact.

Manuscript received 27 October 2022; revised 18 February 2023 and 31 March 2023; accepted 15 May 2023. Date of publication 15 June 2023; date of current version 9 October 2023. This work was supported in part by the Zhejiang Lab Open Research Project under Grant K2022PI0AB01; in part by the Special Fund of Hubei Luojia Laboratory under Grant 220100016; and in part by the Fundamental Research Funds for the Central Universities under Grant 2042023kf0132. *(Corresponding authors: Chuang Hu; Dazhao Cheng.)*

Huanghuang Liang, Qianlong Sang, Chuang Hu, Yili Gong, and Dazhao Cheng are with the School of Computer Science, Wuhan University, Hubei 430072, China (e-mail: hhliang@whu.edu.cn; qlsang@whu.edu.cn; handc@whu.edu.cn; yiligong@whu.edu.cn; dcheng@whu.edu.cn).

Xiaobo Zhou is with the State Key Laboratory of Internet of Things for Smart City and the Department of Computer and Information Sciences, University of Macau, Macau, China (e-mail: waynexzhou@um.edu.mo).

Yu Wang is with the Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122 USA (e-mail: wangyu@temple.edu).

Digital Object Identifier 10.1109/JIOT.2023.3279892

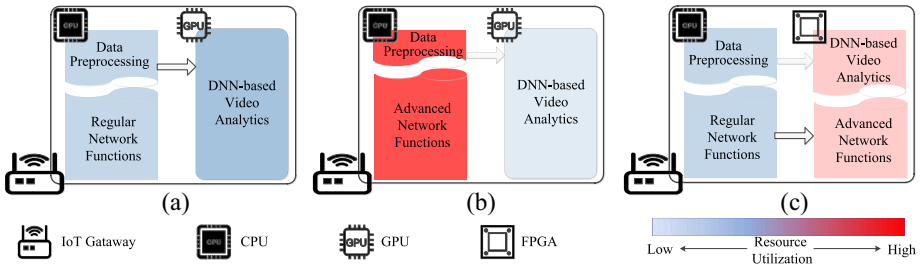

Fig. 1. IoT gateway and its functions. (a) IoT gateway with a GPU for data analytics and regular NFs on CPU. (b) Current IoT gateway with a GPU for VAs and advanced NFs on CPU. (c) IoT gateway with offloading both advanced NFs and data analytics to the hardware accelerator.

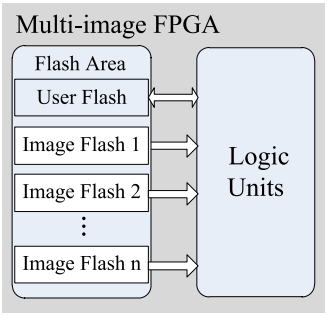

Fig. 2. Multi-image FPGA.

In this article, we present a new accelerator, transmission-analytics processing unit (TAPU), which utilizes multi-image FPGAs to speed up VAs and NFs for data transmission in IoT gateways. Fig. 2 illustrates the architecture of the multi-image FPGA. With a multi-image FPGA, multiple images can be prestored in the FPGA image flash, and it can rapidly switch between these images. The recently released dual-image Intel Max10 FPGA is an example. We can preconfigure one image for analytics and another for NFs. As a result, we can multiplex the analytics and NFs accelerator by switching images. The FPGA's inherent programmability and ability to customize hardware make it possible to design the accelerator with high performance and low power consumption.

To bring TAPU into reality, we face three challenges:

First, the system design for TAPU requires both hardware and software revisions, as it is the first study to utilize multi-image FPGAs as accelerators for IoT gateways. Regarding hardware design, we discuss the FPGA choice and abstract the accelerator modules as hardware functions that developers can call, similar to a software function. For TAPU's software, an offline manager is provided to determine how to preconfigure the two images of the FPGA, and a runtime manager is included in determining how to switch images in response to runtime variations Section III.

Second, it is crucial to determine how to offload the dynamic workload of VAs and network processing to TAPU to optimally utilize its computational capacity. Our approach prioritizes network packet processing with TAPU, leaving the residual computational capacity to accelerate VAs based on the required execution time. To estimate the residual computational capacity, we have developed two schemes for the nonpreemption and general cases Section IV.

Third, minimizing the overall latency in executing VAs tasks is challenging, given the dynamic nature of the residual computation capacity. Motivated by the fact that DNN layers for VAs have different execution efficiencies on CPU and FPGA, we can optimize analytical latency by scheduling the execution location of DNN layers. To this end, we propose a CPU–FPGA analytic model and design an analytic task offloading (ATO) algorithm to make the most of the limited residual computation capacity Section V.

We implemented a prototype of TAPU and evaluated it through trace-driven experiments. Our results demonstrate that TAPU can enhance the performance of VAs and network packet transmission by 1.49 and 2.33 times, respectively, compared to the current approach, with a utilization rate of up to 92.51%. Additionally, we developed two case studies to demonstrate the end-to-end operations of TAPU in the field and its impact on NFs and VAs. These case studies include a campus traffic monitoring system and an office surveillance system.

II. Background

A. Network Functions

As a bridge connecting IoT devices to the cloud, traditional IoT gateways pass data on both ends, deal with various communication protocols, and manage traffic flow like a router. However, current IoT gateways also need to encrypt sensitive data, compress video or image data, and filter unnecessary data to reduce bandwidth consumption and cloud pressure. For example, data files are often compressed before being sent to the cloud, and IoT gateways use different compression methods, such as Gzip, to process these files. Furthermore, each network packet has a latency tolerance, and the processing time using NFs should not exceed this latency. Otherwise, it will lead to packet loss and blocking. Therefore, IoT gateways need to handle network packets promptly to ensure that IoT devices remain connected and functioning properly. This is achieved by invoking various network services, which consume most of the CPU's processing resources.

B. Video Analytics

In order to respond in time and reduce the burden on the cloud, existing IoT gateways typically process data locally, with VAs being the most commonly used. Through analyzing videos, IoT gateways can extract characters or objects in images and feed the results back to the cloud or edge

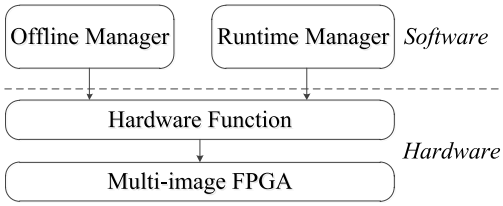

Fig. 3. Overview of the system design of TAPU.

to perform tasks, such as OCR, vehicle counting (VC), and object tracking [6]. For image data from IoT devices, the IoT gateway usually uses DNN for local image processing. An example is Microsoft Rocket, which includes an application called Microsoft Vision Zero [7], which is pretrained for the City of Bellevue VC. The neural network consists of layers of nodes, and the results of nodes in the same layer can be calculated in parallel. However, this implies that using a general-purpose computing unit, such as the CPU, to calculate each node serially is very slow. Therefore, the inference of neural networks often requires the use of parallel processing units, such as GPUs, to provide computation capacity.

C. Multi-Image FPGA

FPGAs consist of a collection of reconfigurable logic blocks that can be programmed to provide customized functions. However, reconfiguring an FPGA can be time-consuming, making it difficult to use in real-time applications. To overcome this limitation, FPGA images, which contain programming information for an FPGA, have been developed. These images can offload certain workloads and accelerate specific tasks, such as programming specific circuits for VAs, much faster than using general-purpose circuits, such as CPUs. Traditional single-image FPGAs typically take minutes to program the FPGA according to a new image. However, recent FPGA developments have led to the emergence of multi-image FPGAs, which can store multiple different images in the user flash memory (UFM) and support fast switching with a bit of switching time (e.g., approximately 9 ms for Intel Max10), as depicted in Fig. 2. This makes it possible to multiplex multiple images in the time dimension, enabling elastic resources and enabling applications, such as NFs and VAs, to be executed simultaneously.

III. System Design

The system design of TAPU is shown in Fig. 3. TAPU consists of both hardware and software designs for transmission and VAs.

A. Hardware Design

TAPU adopts FPGA as its hardware due to its superior performance and programmability. There is a wide variety of FPGAs available, with TAPU opting to use the multi-image FPGA. We will explain the rationale behind this choice and also provide an overview of the hardware functions designed to facilitate the use of the multi-image FPGA.

1) FPGA Choice: TAPU employs a multi-image FPGA as its accelerator to accommodate both NFs and VAs, which share the FPGA resource in a limited way. The choice of a multi-image FPGA enables two ways of sharing the FPGA resource: 1) the *space* dimension, where we implement NF using a part of the logic units, and the rest is used for VAs and 2) the *time* dimension, where one image is designated for NFs and the other for VAs, and only one image is configured to the logic units at a given time.

TAPU opts for sharing the time dimension due to several reasons. First, the limited programmable logic units (PLUs), including look-up tables, registers, and block RAMs, make it difficult to incorporate all functions into the FPGA simultaneously. Even with careful design, each function requires dedicated logic units, making it impractical to share the FPGA on the space dimension. For instance, implementing a medium-sized DNN model, such as GoogLeNet [8], on an Intel Max10 FPGA requires almost 90% of the available logic units. While using a more advanced FPGA with more resources is a possibility, the high cost of such FPGAs makes them impractical. Second, sharing on the time dimension offers more flexibility in resource allocation compared to sharing on the space dimension. With space dimension sharing, the rate of resource occupation for each function is fixed. Modifying the resource allocation for a function requires reprogramming the FPGA after modifying the image, which can take hours. On the other hand, sharing the time dimension allows for the FPGA's resource allocation to be adapted by simply adjusting the time slots.

Sharing resources on the time dimension requires fast switching between images, which is enabled by multi-image FPGAs. For instance, the Intel Max10 can switch between two images in near-zero time (less than 9 ms). Therefore, TAPU has opted for a multi-image FPGA as its hardware platform instead of a traditional single-image FPGA.

2) Skeletons: TAPU offloads various functions to multi-image FPGA and controls the time for image to allocate FPGA resources to functions. However, implementing this technology can be challenging and time-consuming for the upper layer as it requires writing code that covers specifics of FPGA offloading and carefully handling vendor-specific details. We design *skeletons* which includes several function call to reduce the development efforts of the upper layer using the multi-image FPGA.

Fig. 4 shows the Verilog code for image switching in Intel Max10. In line 2, the image file *IMAGE*0 is loaded into the image area. The images are stored in fixed memory areas in Intel Max10, known as master/slave image areas, with the image being stored in the slave image area ($SLAVE_IMAGE$) in this case. Lines 3–5 initialize the write/read parameters, and after some auxiliary code is omitted, line 7 switches the image into the FPGA logic unit supported by the module ($altera_dual_boot$) from Intel. This process is completed by using the dual configuration Intel FPGA IP Core Avalon Memory-Mapped Address Map for Intel MAX10 devices. Fig. 5 shows the Verilog code for switching to an image in Xilinx Artix-7. Lines 2–4 initiate the write/read parameters and assign the storage memory address of *IMAGE*0.

TABLE I
SKELETON

Skeleton	*Function Name*	*Description*
Hardware Setup	FPGA_ON()	Turns on or awake FPGA.
	FPGA_OFF()	Turns off FPGA.
	FPGA_INIT()	Initializes FPGA.
	FPGA_SLEEP()	Switches FPGA to sleep mode.
Image Management	IMG_UPLOAD_VIDEO()	Uploads a video analytics image into FPGA UFM.
	IMG_UPLOAD_NETWORK()	Uploads a network functions image into FPGA UFM.
	IMG_SWITCH_VIDEO()	Switches to video analytics image on FPGA UFM.
	IMG_SWITCH_NETWORK()	Switches to network functions image on FPGA UFM.
Data Processing	DATA_WR()	Writes in data onto external memory.
	DATA_RD()	Reads in data onto external memory.
	VIDEO_TASK_PROCESS()	Executes video analytics task on external memory.
	NETWORK_TASK_PROCESS()	Executes network functions task on external memory.

```
1  module SWITCH_IMG(input clk, input nrest,
       input [66:0] in, output [32:0] res,
       output BOOT_SEL, output CFG_SEL);
2    assign SLAVE_IMAGE = IMAGE0;
3    wire read, write;
4    assign [2:0] ADDR = [34:32] in;
5    assign [31:0] WD_DATA = [66:35] in;
6    ...
323  altera_dual_boot adb(.clk(clk),.nreset(
         nreset),.avmm_rcv_address(ADDR),.
         avmm_rcv_read(read),.
         avmm_rcv_writedata(WD_DATA),.
         avmm_rcv_write(write),.
         avmm_rcv_readdata(res)
324  );
325 endmodule
```

Fig. 4. FPGA image switching codes of Intel Max10.

```
1  module SWITCH_IMG(input clk, input
       nreset, input [63:0] in, input
       boot_sel, output [31:0] res);
2    assign [31:0] IMG0_ADDR = [31:0] in;
3    reg [31:0] CSIB;
4    reg I, RDWRB;
5    ...
253  ICAPE2 #(
254     .DEVICE_ID(DEVICE_ID),
255     .ICAP_WIDTH(ICAP_WIDTH),
256     .SIM_CFG_FILE_NAME("None")
257   ) ICAPE2_inst (.O(res),.CLK(clk),.
          CSIB(CSIB),.I(I),.RDWRB(RDWRB)
258  );
259 endmodule
```

Fig. 5. FPGA image switching codes of Xilinx Artix-7.

This differs from Intel Max10, as Xilinx Artix-7 stores images in dynamically allocated memory areas instead of fixed image areas. Lines 6–11 switch images into the FPGA logic units. Here, Xilinx Artix-7 completes this process using a primitive (*ICAPE2*) from Xilinx.

The skeletons offer the upper layer a programming model that completely abstracts away the low-level details of the multi-image FPGA. The APIs for interacting with the FPGA at the upper layer are presented in Table I. These hardware APIs allow easy utilization of the multi-image FPGA through simple function calls, without requiring knowledge of the underlying implementation details. we provide a step-by-step explanation below to demonstrate how to execute a VAs

```
1  def Video_Analytics(img, frame):
2    if FPGA_STATUS!=ON:
3      FPGA_ON()
4    IMG_UPLOAD_VIDEO(img)
5    IMG_SWITCH_VIDEO()
6    DATA_ADDR,size=DATA_WR(frame)
7    RES_ADDR=VIDEO_TASK_PROCESS(
         DATA_ADDR,SIZE)
8    result=DATA_RD(RES_ADDR)
9    FPGA_OFF()
10   return result
```

Fig. 6. VAs codes using TAPU hardware functions.

task using the hardware APIs, as shown in Fig. 6. First, the function `FPGA_ON()` is called to turn on the FPGA when it is off. After that, the function `IMG_UPLOAD_VIDEO()` and `IMG_SWITCH_VIDEO()` are called to switch the image to the VAs image and transfer the frames to FPGA by calling function `DATA_WR()`. Then, the function `VIDEO_TASK_PROCESS()` is called to process the frame of the video to get the analytic result. Finally, the result is obtained from the FPGA by calling function `DATA_RD()`, and the function `FPGA_OFF()` is called to turn off the FPGA.

B. Software Design

We have designed software tools to facilitate application development and utilize FPGA resources. For example, TAs shown in Fig. 3, the software of TAPU includes two components: 1) TAPU Offline, which is responsible for determining which functions should be offloaded to the FPGA and how to preconfigure the FPGA images and 2) TAPU Runtime handles runtime switching and adapts to runtime variations to maximize TAPU resource utilization. We have developed two separate tools: 1) the Offline Manager and 2) the Runtime Manager. With these tools, developers can more efficiently and effectively utilize FPGA resources in their applications.

1) Offline Manager: The offline manager is aimed to liberate the developers from taking a lot of time in FPGA programming and deciding on images of FPGA. For the former objective, the offline manager designs a database called *Image Lib*, which stores preconfigured images. For the latter one, the offline manager designs an *NFs Image Decision*

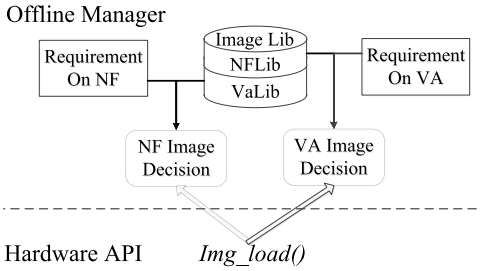

Fig. 7. Offline manager.

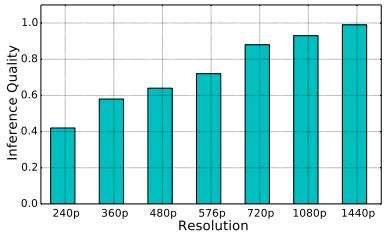

Fig. 8. Inference quality profile of the YOLOv2 model.

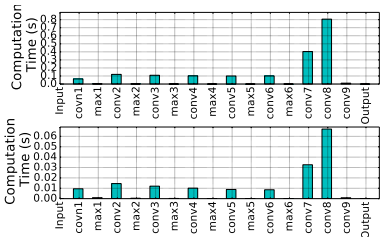

Fig. 9. Computation latency of YOLOv2 layers by CPU (top) and FPGA (bottom).

module to determine which NFs are offloaded to FPGA and a *VAs Image Decision* module to determine which DNN model Image is used for VAs. Fig. 7 provides a visual representation of the modules within the Offline Manager.

Image Library: FPGA development and programming often take several hours to synthesize and implement hardware description language (HDL) source code to the FPGA image. This can create a significant barrier to fast development, especially when there is a demand for rapid prototyping and iteration. To address this challenge, TAPU has implemented an image library (Image Lib) which aims to provide available images and includes two distinct components: 1) the NF library (NFLib) and 2) the VAs models library (VALib). The NFLib contains images that implement commonly used NFs, such as IPsec (a secure network protocol suite that encrypts and authenticates data packets) and Gzip. On the other hand, the VALib contains images that implement commonly used the VAs models, including the lightweight DNN model YOLOV2 which has moderate prediction accuracy, and the complex DNN model VGG which has high prediction accuracy. However, Image Lib cannot contain all NFs and the video vision models in advance. Therefore, TAPU provides an API for developers to add their images to the respective image library.

NF Image Decision: The IoT gateway contains lots of network operations, but the number of FPGA images is limited. So it is unfeasible to offload all NFs to FPGA. Consequently, we need to determine which NFs should be offloaded to FPGA. We design the decision module which determines the NF image based on the performance of processing the NFs on different hardware and the "stability" of these NFs.

The first consideration is the performance of processing NFs. Operating NFs on FPGA is suboptimal occasionally, and running them on CPU is sometimes favorable. There are two types of processing in the data plane: 1) *shallow packet processing* and 2) *deep packet processing*. Shallow packet processing involves executing operations based on the packet header, such as NAT and firewall. In contrast, deep packet processing involves executing operations on the entire packet data, such as IPsec gateway and flow compression. As packet headers follow a unified protocol specification, shallow packet processing typically does not require considerable computing resources. Performing shallow packet processing on a CPU is relatively fast. However, deep packet processing typically requires more compute resources as the data of higher layers lacks regular patterns, resulting in longer processing times that require more CPU cycles [9]. Therefore, an NF image decision module prioritizes offloading deep packet processing types of NFs to FPGA. If still, images are available, it offloads shallow packet processing types of NFs. The second consideration for the NF image decision module is the stability of NFs. Stable NFs are preferred to be offloaded to FPGA, as modifying or debugging their design on FPGA is time-consuming. An NF image decision module determines the stability of an NF by counting its changing frequency based on the development log. This parameter is used to decide which NFs to offload to FPGA.

In brief, an NF image decision module determines which NFs to offload to FPGA based on the packet processing type and NF stability.

VA Image Decision: Unlike NFs, only one DNN model is necessary for VAs at the IoT gateway. Different VAs models and input resolutions of video lead to typical compute workloads and levels of inference. For instance, as Fig. 8 shows, the inference quality (F1 score) is affected by the resolution of the input frame.

The problem we face is selecting a DNN model that can meet the requirements of video resolution and inference quality while minimizing the VAs execution time. To this end, we propose a two-step approach: First, we profile the relationships among the DNN model, input resolution, inference quality, and VAs execution time. These parameters can be accurately derived in advance. For instance, we derive the processing delay of each layer of the YOLOv2 model on Raspberry Pi 3 model B (representing CPU) and Intel Max10 FPGA, and the results are depicted in Fig. 9. Second, we select the model that meets VAs requirements while achieving the minimum execution time.

2) Runtime Manager: The Runtime Manager is responsible for managing the allocation of FPGA resources. Precisely, it controls the switching of FPGA images to assign FPGA

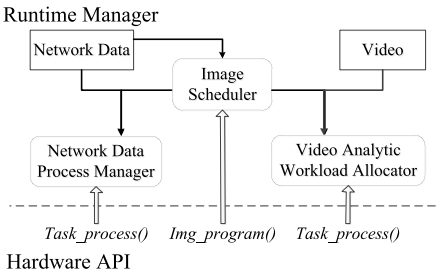

Fig. 10. Runtime manager.

resources to NFs and VAs, striving to maximize the utilization of FPGA resources. The Runtime Manager comprises the modules illustrated in Fig. 10.

Image Scheduler, forming the core of the Runtime Manager, is designed to facilitate the allocation of FPGA resources to NFs and VAs by controlling the switching of FPGA images.

Image Scheduler first allocates resources for NFs before utilizing the residual computation capacity to accelerate VAs. When allocating FPGA resources, it prioritizes NFs as Ethernet packet sizes can range from 64 to 1500 bytes. According to FPGA-based acceleration solutions, NFs processing such small amounts of data only require ultralow microsecond-level latency. In contrast, video requires millisecond-level latency and has data sizes ranging from tens to hundreds of megabytes. The latency for NFs is substantially lower than that for VAs. The Image Scheduler then estimates the network traffic. It assigns enough FPGA resources for NFs (switching to the appropriate NFs image in time) and the remaining FPGA resources to VAs. The calculation of residual resources and the estimation of network traffic are discussed in Section IV. Then, Image scheduler sends the allocation result to the network data process manager and VAs workload allocator and calls hardware API `IMG_SWITCH_VIDEO` or `IMG_SWITCH_NETWORK` to switch the corresponding image when resources are allocated to that function.

Network Data Process Manager is designed to schedule network packet processing upon receipt of the resource allocation result from the Image Scheduler. Upon allocation of resources to NFs, this module invokes the hardware API `NETWORK_TASK_PROCESS` to execute NFs on all unprocessed network packets.

When TAPU employs a trace-driven estimation scheme (TDES) to estimate the network traffic, as described in Section IV-B, this module also serves another purpose. It sends a preemption signal to the Image Scheduler Module when incoming network packets will exceed the delay tolerance. Upon receiving a preemption signal, the Image Scheduler Module will immediately switch the FPGA to the NFs Image.

VAs Workload Allocator is responsible for scheduling the execution of *VAs tasks* when FPGA resources are assigned for VAs operations. In particular, a VAs task involves the processing of a single frame of video using a designated DNN model.

The execution of VAs tasks can be carried out on either the local CPU or an FPGA. However, not all types of layers of a DNN have higher performance when processed by an FPGA. In some cases, it is more efficient to process certain layers on the CPU than on an FPGA, such as when the fully connected layer has small data and a big model. In this case, the performance of the CPU is not too far off from that of the FPGA. Thus, TAPU opts to schedule VAs tasks in layer granularity with a CPU–FPGA analytic model. The VAs workload allocator runs the ATO optimization algorithm described in Section V to minimize the overall delay in processing a VAs task using the allocated FPGA resource.

TABLE II
MAIN NOTATIONS OF CAPACITY ESTIMATION PROBLEM

Symbol	Meaning
D_p	Period
D_r	Residual computation capacity per period
v_{max}	Maximum generation speed of network packets
d	The tolerance delay
s	The network packet processing throughput of FPGA
v	The number of packets generated in the previous period
v_p	The number of packets
P_i	The i-th period
w_i	The actual number of network packets generated in P_i
v_i	The predicted number of network packets generated in P_i

IV. RESIDUAL COMPUTATION CAPACITY ESTIMATION

We are now analyzing the estimation of residual computing capacity. While the FPGA is shared in the temporal dimension, we use the amount of time that may be given for VAs to measure the residual computational capacity. We divide time into *periods*. Let D_p be the time length for a period. Each D_p consists of a subperiod D_n for the NFs and a subperiod D_r for VAs. Table II contains the notations used in the problem formulation.

The traffic load from the application largely determines the duration of the subperiod of NFs virtualization (NFV). Accurately estimating the application traffic load significantly depends on whether the system is designed for a single purpose or a general application. For instance, single applications, such as traffic surveillance cameras, which transmit video of traffic to a remote control center and can recognize car license plates appearing in the video, are examples of single-purpose systems. On the other hand, general-purpose engines are developed to facilitate VAs for a suite of upper-layer applications, such as a mobile phone with functions like speech recognition, fingerprint authentication, and face recognition. Our framework can be applied to both.

If the system is designed for a single application, we can develop a comprehensive traffic load model that accounts for the application's characteristics. There have been numerous studies conducted in this area, such as Tanwir et al. [10] employed a Markov modulated gamma process to model 3-D video traffic, and Park et al. [11] proposed a probability density function to model the network traffic for games. Three primary methods for general application traffic estimation are available for modeling traffic. The first assumes sources continually send data at a constant rate to the network [12]. The second method utilizes probabilistic functions to model traffic behavior, which

typically observes the application traffic following a specific distribution and adjusts the distribution parameters to model the network traffic [13]. Finally, the third method employs traffic traces to evaluate network performance [14].

The distinctiveness of our case lies in *preemption*, i.e., network packet processing is the primary concern and VAs acceleration can be interrupted when necessary. Therefore, we initially examine an extreme case wherein application traffic estimation should not be subject to any preemption, which we refer to as nonpreemption estimation scheme (NPES). We then analyze a more general situation in which preemption is allowed and estimate application traffic based on the traffic load of previous periods, which we term *TDES*. This approach is straightforward and can be applied in a variety of contexts.

A. Nonpreemption Estimation Scheme

We propose an NPES for the purpose of estimating the residual computation capacity, with the aim of ensuring no preemption and maximizing the residual computation capacity D_r in each period.

The residual computation capacity of a period is contingent upon: 1) the duration of the period; 2) the quantity of packets processed in the period; and 3) the network packet processing throughput of the FPGA. To precisely calculate the residual computation capacity of each period, knowledge of the number of packets processed in the period is required. Therefore, for the period i, NPES opts to buffer all network packets and process them at the commencement of the following period $i+1$.

The first objective of NPES is to ensure that no preemption occurs. As stated, each packet has an associated delay tolerance d, which is determined by the application. To prevent preemption, the packet must be processed before the tolerated delay is exceeded. We make the assumption that the FPGA's network packet processing throughput w is greater than the maximum generation speed of the network packets $v_{\max}$, i.e., $s > v_{\max}$.

Theorem 1: If the network packet processing throughput w of FPGA is more significant than the maximum generation speed of network packets $v_{\max}$, and the period D_p is not greater than $(d/2)$, then NPES can operate without incurring preemption.

Proof: No preemption means all packets can be processed before the tolerance delay d. Thus, we prove that if $s > v_{\max}$ and $D_p \le (d/2)$, all packets can be processed before validating the tolerated delay. Let D_w denote the delay a packet p_j needed to wait to be processed, we prove that $D_w \le d$.

D_w consists of two parts: 1) the time in which the packet is buffered in its generation period i, denoted as D_b and 2) the time needed to wait for NIC to process the unprocessed packets generated before p_j in the period $i+1$, denoted as D_c. It is obvious that $D_b \le D_p$. Let N denote the number of unprocessed packets generated before p_j. We have $D_c = (N/s)$. As the maximum generation speed of the network packets is $v_{\max}$, thus $N \le D_p v_{\max}$, combined with $D_c = (N/s)$ and $v_{\max} \le s$, we have $D_c \le D_p$. Therefore, $D_w = D_b + D_c \le 2D_p$. If $D_p \le (d/2)$, we have $D_w \le d$. ■

Processing network packet on FPGA is faster than the generation of network packets, so a longer period will lead to more residual computation capacity. Therefore, the length of the period is set to be $(d/2)$, i.e., $D_p = (d/2)$.

Let v be the number of network packets generated in the previous period and s denote the throughput of network packet processing on FPGA. We can compute $D_r = (d/2) - (v/s)$.

B. Trace-Driven Estimation Scheme

We propose a TDES for a general case that allows preemption to maximize the residual computation capacity D_r in each period. Our approach is to predict the number of packets generated in the period. If the exact number of packets, denoted as v_p, can be predicted, then the residual computation capacity D_r can be computed as $D_r = D_p - (v_p/s)$, where s is the network packet processing throughput of FPGA.

We used network traffic traces to estimate application traffic rather than relying on a constant injection rate and probabilistic function. The trace-based method generally yields a more accurate prediction than the latter [15]. To this end, we employed an auto-regressive (AR) [16] predictor to predict the upcoming period's traffic load based on the preceding periods' traffic load.

Let K denote the number of periods whose network information is used to estimate the network traffic of the next period P_p. Let P_i denote the ith period before P_p, where $i = 1, 2, \ldots, K$. Let w_i and v_i denote the actual and predicted number of network packets generated in period P_i, respectively. Let v_p denote the number of network packets generated in period P_p. The AR model can be present as $v_p = \eta_1 w_1 + \eta_2 w_2 + \cdots + \eta_K w_K + \alpha$, where α is the white noise, η_i $(1 \le i \le K)$ is the smoothing value which reflects the periodic characteristics of the network packet. We adopt the following equation to compute the mean square error (MSE) between the actual packet quantity w_i and the periodic network packets q_i

$$\text{MSE} = \frac{1}{K}\sum_{i=1}^{K}(w_i - v_i)^2. \tag{1}$$

The smaller MSE can better reflect the periodic characteristics. Thus, we carefully choose K and η_i $(1 \le i \le K)$ to make MSE smallest.

Now, we consider the length of D_p, which affects the residual computation capacity. Each packet has a delay tolerance of d, which means it can be buffered at the most d time or a preemption happens. Since it is difficult to predict the exact generation time of network packets, we consider the worse case that a packet is generated at the beginning of the period. Thus, we set the length of the period to be d, i.e., $D_p = d$. We can compute the residual computation power $D_r = d - (v_p/s)$.

V. Video Analytics Task Offloading Optimization

The residual computation capacity allocated for VAs at any given time may not be sufficient to execute a full VA task. To take advantage of the limited residual computation capacity, we propose the CPU–FPGA analytic model, wherein

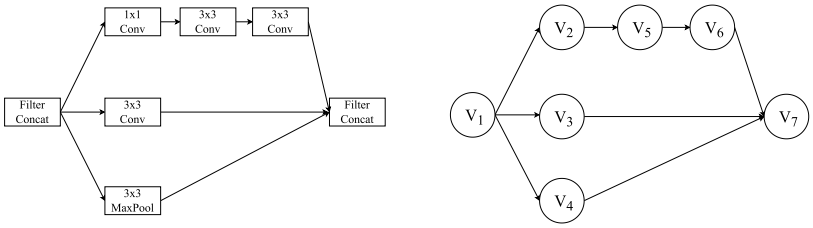

Fig. 11. Different representation of the Inception v4 model. (a) Layer representation. (b) Graph representation.

TABLE III
MAIN NOTATIONS OF OFFLOADING OPTIMIZATION PROBLEM

Symbol	Meaning
$\mathcal{G}$	The DAG of DNN
$\mathcal{V}$	The set of vertices representing the layers of DNN
$\mathcal{E}$	The set of links
$\mathcal{V}_c$	The vertices processed by CPU
$\mathcal{V}_f$	The vertices processed by FPGA
d_i^c	The time needed to process v_i by CPU
d_i^f	The time needed to process v_i by FPGA
d_i^m	The time needed to transfer the output data of v_i
D_c	The processing delay at CPU
D_m	The transimission delay
D_f	The processing delay at FPGA

some layers of a DNN are executed on the CPU, while the remaining layers are carried out on the FPGA. We explore how to offload analytics from the CPU to the FPGA with the aim of minimizing the overall delay in executing VA tasks. We formulate the ATO problem as an NP-hard problem and design a heuristic algorithm to address it. The notations used in the formulation of the problem are listed in Table III.

A. CPU–FPGA Analytic Model

In this section, we introduce the CPU–FPGA analytic model in a formal manner.

1) DNN as Graph: A DNN is structured as a directed acyclic graph (DAG) communication diagram. Each node in the diagram symbolizes a single layer of the neural network, which is inseparable and must be processed either at the edge or in the cloud. The connections within the graph show the communication and dependencies among the layers.

Let $\mathcal{G} = (\mathcal{E}, \mathcal{V})$ denote the DAG of DNN, where $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$ is the set of vertices representing the layers of the DNN (especially, v_1 and v_n represent the input layer and output layer, respectively). $\mathcal{E}$ is the set of links. A link $(v_i, v_j) \in \mathcal{E}$ represents that v_i needs to feed its results after processing to v_j. Fig. 11(b) shows the DAG of the pure inception v4 network [17] in Fig. 11(a).

The processing time of each layer is dependent on where it is handled (i.e., CPU or FPGA). Let d_i^c and d_i^f be the processing time of v_i by CPU and FPGA, respectively. Let d_i^m denote the time of v_i needed to transfer its output data in the middle of the CPU/memory and FPGA through the PCI-E bus if v_i and its successor are processed at a different context (e.g., v_i is processed at CPU, while at least one of its successor is processed at FPGA).

We define $\mathbf{D}_c = \{d_1^c, d_2^c, \ldots, d_n^c\}$, $\mathbf{D}_m = \{d_1^m, d_2^m, \ldots, d_n^m\}$, $\mathbf{D}_f = \{d_1^f, d_2^f, \ldots, d_n^f\}$. They signify the *three major delays*: 1) computation delay at CPU; 2) transmission delay; and 3) computation delay at FPGA of each layer. These values can be determined in advance.

2) DNN Partitioning: Our objective is to partition DNN into two parts so that one part is managed by CPU and the other is managed by FPGA. Algebraically, we should find a bipartition of $\mathcal{V}$: $(\mathcal{V}_c, \mathcal{V}_f)$, where $\mathcal{V}_c \bigcup \mathcal{V}_f = \mathcal{V}$ and $\mathcal{V}_c \bigcap \mathcal{V}_f = \emptyset$. $\mathcal{V}_c$ are processed by CPU and $\mathcal{V}_f$ are processed by FPGA. Let $\{\mathcal{V}_m = v_i : (v_i, v_j) \in \mathcal{E} \wedge (v_i \in \mathcal{V}_c \wedge V_i \in \mathcal{V}_f \vee v_i \in \mathcal{V}_f \wedge V_i \in \mathcal{V}_c)\}$. The end result of the vertices in $\mathcal{V}_m$ will be transferred between CPU/memory and FPGA through the PCI-E bus. $\mathcal{V}_c$ will produce a processing lag at CPU. $\mathcal{V}_m$ will cause transmission latency. $\mathcal{V}_f$ will create a processing holdup at FPGA.

3) Delay Components and Constraint: Let D_{total} denote the overall delay to process a VA task. Once the division is constructed, each frame is handled by three phases, i.e., processed by CPU, sent from CPU/memory to FPGA through PCI-E bus, and processed by FPGA. The overall delay is the sum of the delays of the three stages.

The postponements of the three phases are portrayed as follows. In the CPU-processing phase

$$D_c = \sum_{v_i \in \mathcal{V}_c} d_i^c. \tag{2}$$

In the FPGA-computing stage

$$D_f = \sum_{v_i \in \mathcal{V}_F} d_i^f. \tag{3}$$

In the transmission stage

$$D_m = \sum_{v_i \in \mathcal{V}_m} d_i^t. \tag{4}$$

Thus, we have $D_{\text{total}} = D_c + D_m + D_f$.

Note that the CPU of our system is perpetually ready for carrying out VAs tasks, while the FPGA can be employed for executing VAs tasks solely within the restricted remaining computation power. Therefore, the delay of the FPGA-computing stage cannot exceed the allocated computation capacity. Let D_r denote the allocated residual computation

capacity. We have the following constraint:

$$D_f = \sum_{v_i \in \mathcal{V}_f} d_i^f \le D_r. \tag{5}$$

B. Problem Formulation and Analysis

The goal of the *ATO problem* in this article is to reduce the total postponement in carrying out a VAs task, given the assigned computation capability, the DNN architecture diagram, processing pauses of each layer at CPU and FPGA, and transmission delay of each layer. Note that hardware implementation (i.e., FPGA) of a DNN layer is significantly faster than that of one general-purpose processor, thus reducing the total latency is equivalent to reducing the aggregate delays of the CPU-processing phase and transmission phase. Therefore, the aggregate delays of the CPU-processing phase and transmission phase are the goal. The decision variable is the partition vertices $\mathcal{V}_P$. The constraint is that the delay of the FPGA-computing stage does not exceed the allocated computation capacity. In conclusion, we define *ATO* optimization problem as follows.

Problem 1 (ATO): Given $\mathcal{G}$, $\mathbf{D}_c$, $\mathbf{D}_f$, $\mathbf{D}_m$, and D_r, determine $\mathcal{V}_c$ and $\mathcal{V}_f$, subject to constraint (5), to minimize $D_{cm} = D_c + D_m$.

We analyze the complexity of the problem ATO. We prove problem ATO is NP-hard by reducing it to an introduced decision (IntroD) problem which is NP-complete. The following quandary IntroD can be defined.

Problem 2 (IntroD): Given $\mathcal{G}, \mathbf{D}_c, \mathbf{D}_f, \mathbf{D}_m, D_r$, and D_0, is there a partition $(\mathcal{V}_v, \mathcal{V}_f)$ so that $D_f \le D_r$ and $D_{cm} = D_c + D_m \le T_0$?

Theorem 2: Problem IntroD is NP-complete even if only graphs with no links are considered.

Proof: To demonstrate the NP-hardness, we map the Knapsack dilemma to problem IntroD. Let an instance of the Knapsack problem be provided, i.e., there are n items, the weights of the items are represented by w_i, the cost of the objects by p_i, the weight limit by W, and the cost limit by K. The target is to decide whether there is a collection X of elements so that $\sum_{i\in X} w_i \le W$ and $\sum_{i\in X} p_i \ge K$. Based on this, we define an instance of problem IntroD as follows: $\mathcal{V} = v_1, v_2, \ldots, v_n, \mathcal{E} = \emptyset$. Let $d_i^f = p_i, d_i^c = w_i$ (since $\mathcal{E} = \emptyset$, we can define $d_i^c = 0$). Introducing $A = \sum_{v_i\in\mathcal{V}} p_i$. Let $D_0 = W$ and $D_r = W - K$.

We declare that this instance of IntroD is resolvable if the original (3) Knapsack problem has a solution. Assuming that problem IntroD has a solution $(\mathcal{V}_c, \mathcal{V}_f)$, where $\mathcal{V}_c \cup \mathcal{V}_f = \mathcal{V}$ and $\mathcal{V}_c \cap \mathcal{V}_f = \emptyset$. This means that $D_c + D_m \le W$ and $D_f = \sum_{v_i\in\mathcal{V}_f} \le A - K = \sum_{v_i\in\mathcal{V}} p_i - K$. The latter one can be formulated as $K \le \sum_{v_i\in\mathcal{V}} p_i - \sum_{v_i\in\mathcal{V}_f} p_i = \sum_{v_i\in\mathcal{V}_c} p_i$. This proves that $X = \mathcal{V}_f$ is a solution to the original Knapsack problem.

Now, we assume that X resolves the Knapsack issue. Therefore, $\sum_{v_i\in X} d_i^c = \sum_{v_i\in X} w_i \le W = D_0$ and $\sum_{v_i\in X} p_i \ge K = A - D_r = \sum_{v_i\in\mathcal{V}} p_i - D_r$. The latter one can be formulated as $D_r \ge \sum_{v_i\in\mathcal{V}} p_i - \sum_{v_i\in X} p_i = \sum_{v_i\in\mathcal{V}/X} p_i = \sum_{v_i\in\mathcal{V}/X} d_i^f$. This verifies that $(\mathcal{V}/X, X)$ solves problem IntroD. ■

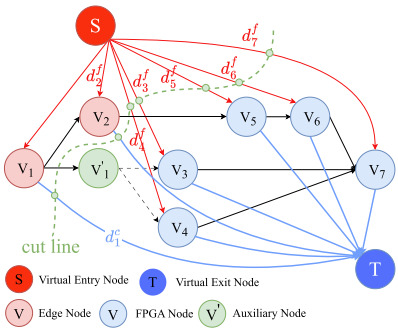

Fig. 12. Illustration of conversion to $s-t$ cut problem using inception v4 as an example.

The preceding evidence suggests that the particular case of IntroD, in which the graph has no edge, is analogous to the Knapsack problem.

Theorem 3: Problem ATO is NP-hard.

Proof: Problem IntroD can be reduced to problem ATO: ATO provides a solution where $D_f \le D_r$ and $D_{cm} = D_c + D_m$ is minimal; let this value be D_{cm}^*. Clearly IntroD is solvable iff $D_{cm}^* \le D_0$. ■

C. ATO Algorithm

We tackle the ATO conundrum. As Problem ATO is NP-hard, it is improbable to discover a worldwide ideal solution within a polynomial time frame. To this end, we invent the minimize VA delay (MVAD) algorithm, which furnishes a regionally optimal solution. The rationale for developing MVAD is as follows. First, we transform the partition problem into a weighted $s-t$ graph cut problem. Subsequently, we generate a set of graph cuts, each of which corresponds to a partition, and we choose the optimal division from this collection that adheres to the stipulated threshold on T_r.

We first illustrate how to convert the partition problem to an $s-t$ graph cut problem. A segmentation of the nodes of a graph into two distinct sets is a cut. The $s-t$ cut of $\mathcal{G}'$ is a cut that necessitates source s and sinks t to be in separate collections. The magnitude of a cut is specified as the aggregate of the price of each connection in the cut-set. The challenge here is that each node in $\mathcal{G}$ carries three delay values d_i^c, d_i^f, and d_i^m. The delay that adds to the total delay relies on where the node is processed. To this end, we construct an auxiliary graph $\mathcal{G}'$, so that each link merely captures a single delay value. The construct of $\mathcal{G}'$ is shown in Fig. 12: 1) in accordance with $\mathcal{G}$, two nodes S and T are incorporated to designate virtual entry and exit nodes, respectively; 2) we establish connections between S and each vertex $v_i \in \mathcal{V}$ with the magnitude of αd_i^f, referred to as "red links," to capture the FPGA-computing delay of v_i; 3) we interconnect each vertex $v_i \in \mathcal{V}$ and T by assigning a value of βd_i^c, referred to as "blue links," to account for the CPU-computing delay of v; and 4) we assign the value γd_i^t to link (v_i, v_j), to capture the transmission delays. Here, $\alpha \in R^+$, $\beta \in R^+$, $\gamma \in R^+$.

However, this is insufficient as one vertex may have multiple successors and its communication delay is counted multiple times. For example, v_1 in Fig. 11(b) has three outgoing links

but the communication delay of v_1 has to be counted at most once. To this end, we introduce *auxiliary vertices* into graph $\mathcal{G}'$. That is, for any vertex $v_k \in \mathcal{V}$ whose outdegree is greater than one, we add an auxiliary vertex v'_k and link (v_k, v'_k). The links from v_k to successors of v_k are now replaced from v'_k to successors of v_k. As shown in Fig. 12, the outdegree of vertex v_1 is greater than one, we thus add an auxiliary vertex v'_1 and link (v_1, v'_1). The links (v_1, v_3) and (v_1, v_4) are replaced by links (v'_1, v_3) and (v'_1, v_4), respectively.

Now, without considering S and T, if a vertex v has one successor, the link starting from v corresponds to its communication delay (black link shown in Fig. 12). If v has multiple successors, all the links starting from v (dashed links shown in Fig. 12) should not be considered since the delay has already been considered from v to v'.

In this way, each $s-t$ cut of $\mathcal{G}'$ with $\alpha = 1$, $\beta = 1$ and $\gamma = 1$ correspond to a DNN partition. More specifically, if cutting on the link from S to $v_i \in \mathcal{V}$ (red link shown in Fig. 12), then v_i will be processed by FPGA, i.e., $v_i \in \mathcal{V}_c$. If cutting on the link from $v_j \in \mathcal{V}$ to T (blue link show in Fig. 12), then v_j will be processed on the edge, i.e., $v_j \in \mathcal{V}_f$. If cutting on the link from $v_i \in \mathcal{V}$ to $v_j \in \mathcal{V} \bigcup \mathcal{V}_D$ (black link show in Fig. 12), then the information of v_i will be exchanged between CPU and FPGA, i.e., $v_i \in \mathcal{V}_m$. The total cost of severing red links amounts to FPGA-processing time D_f. The total cost of severing on blue links equals CPU-computation time D_c. The total cost of cutting black links amounts to transmission time.

The second step is to generate a set of partitions and select the best one. The approach is to find the *minimum* $s-t$ cut of $\mathcal{G}'$ with several different α, β, and γ values. Note that the smallest division of the graph problem can be solved in polynomial time, and we utilize Goldberg's algorithm [18], which works in $\mathcal{O}(n^3)$ time. In this way, a partition is generated with a specific α, β, γ tuple. Afterward, we assess if this division is adequate for ATO. If it exceeds all preexisting solutions, it is deemed a novel solution to ATO. We repeat the preceding process for a broad spectrum of α, β, and γ tuples.

Obviously, the outcome of the least s–t cut of $\mathcal{G}'$ is dictated by the proportion of three parameters as opposed to their exact figures. Thus, we can fix one of the three, say $\gamma = 1$, and merely modify the other two. Therefore, we have a 2-D search space for α and β. Our algorithm employs a two-stage approach for optimal speed. First, we initially search in the 2-D plane with a rough granularity to find the top outcome approximately. Then, in the second stage, we apply a finer granularity search in the vicinity of the best solution for further improvement. We reiterate the steps until the augmented performance is less than a limit ϵ or at most h round.

The overall algorithm `MVAD()` is shown in Algorithm 1. A function `min-cut()` (line 14) is designed to construct $\mathcal{G}'$ and compute the minimum cut of $\mathcal{G}'$. A function `search()` (lines 10–17) is aimed to search for the best cut in a given space $\mathbf{S} \triangleq [\alpha_l, \beta_l, \alpha_h, \beta_h]$, meaning that $\alpha_l \le \alpha \le \alpha_h$, $\beta_l \le \beta \le \beta_h$, and a granularity δ (lines 12 and 13), i.e., the step size of changing α and β is δ each time. For each α and β, `search()` calls `min-cut()` to compute the minimum cut. Lines 15 and 16 guarantee the cut fulfills the given limit on

Algorithm 1: MVAD Algorithm `MVAD()`

Input: $\mathcal{G}, \mathbf{D}_c, \mathbf{D}_f, \mathbf{D}_m, D_r, \epsilon, h, K$
Output: $\mathcal{V}_c, \mathcal{V}_f$

1 $D_{\min} \leftarrow +\infty$; $D'_{\min} \leftarrow 0$; $H \leftarrow 0$; $\delta \leftarrow 1$;
2 $\alpha_l \leftarrow 0$; $\beta_l \leftarrow 0$; $\alpha_u \leftarrow \frac{\sum(\mathbf{D}_c)}{\min(\mathbf{D}_m)}$; $\beta_u \leftarrow \frac{\sum(\mathbf{D}_f)}{\min(\mathbf{D}_m)}$;
3 $\mathbf{S} \leftarrow [\alpha_l, \beta_l, \alpha_u, \beta_u]$;
4 **while** $|D'_{\min} - D_{\min}| \ge \epsilon \wedge H \le h$ **do**
5 $\quad H \leftarrow H + 1$; $D'_{\min} \leftarrow D_{\min}$;
6 $\quad [\alpha, \beta, \mathcal{V}_c, \mathcal{V}_f, D_{\min}] \leftarrow$`Search`$(\mathbf{S}, \delta, D_{\min})$;
7 $\quad \alpha_l \leftarrow \alpha - \delta$; $\alpha_u \leftarrow \alpha + \delta$; $\beta_l \leftarrow \beta - \delta$; $\beta_u \leftarrow \beta + \delta$;
8 $\quad \delta \leftarrow \delta / K$;
9 return $\mathcal{V}_c, \mathcal{V}_f$;
10 **function** `Search`$([\alpha_l, \beta_l, \alpha_u, \beta_u], \delta, D^*_{\min})$
11 $\quad D_{\min} \leftarrow +\infty$;
12 $\quad$ **for** $\alpha \leftarrow \alpha_l; \alpha \le \alpha_u; \alpha \leftarrow \alpha + \delta$ **do**
13 $\quad\quad$ **for** $\beta \leftarrow \beta_l; \beta \le \beta_u; \beta \leftarrow \beta + \delta$ **do**
14 $\quad\quad\quad [\mathcal{V}_c, \mathcal{V}_f, D_{\min}, D_f] \leftarrow$`min-cut`$(\mathcal{G}, \alpha\mathbf{D}_c, \beta\mathbf{D}_f, \mathbf{D}_m)$;
15 $\quad\quad\quad$ **if** $D_{\min} \le D^*_{\min} \wedge D_f \le D_r$ **then**
16 $\quad\quad\quad\quad \alpha^* \leftarrow \alpha$; $\beta^* \leftarrow \beta$; $D^*_{\min} \leftarrow D_{\min}$;
17 $\quad$ return $\alpha^*, \beta^*, \mathcal{V}_c, \mathcal{V}_f, D^*_{\min}$;

FPGA computation capacity and the sum of CPU-computing and transmission delay is nonincreasing.

The overall algorithm begins by setting the search granularity δ to be 1 (line 1) and the search space large enough (line 2 and 3). Subsequently, it invokes `search()` (line 7) to traverse the given space $\mathbf{S}$ with a granularity δ, and returns the best α and β found currently. Then, `MVAD()` narrows down the search space $\mathbf{S}$ (line 7) to the neighborhood of the best α and β for the current iteration, and adjusts δ to a finer granularity (line 8). Such area $\mathbf{S}$ and fineness δ is yielded to `search()`. The endpoint for the cycle is that the enhanced performance is less than a limit ϵ (line 4), or the computation round exceeds h. Finally, it returns the cut with the best-found performance (line 9). Clearly, we can attain an ideal localized outcome with regard to the neighborhood of the final α and β. `MVAD()` is a polynomial-time algorithm with the computational complexity of $\mathcal{O}(hn^3)$.

VI. Implementation

We implemented a prototype of TAPU and incorporated it into an IoT portal, as shown in Fig. 14. We employed a Raspberry Pi to replicate an IoT gateway, which was combined with a Logitech BRIO camera for video recording. We arranged a cloud as the data-receiving terminal to communicate with the Raspberry Pi, and a laptop computer to demonstrate the cloud end. Two profound packet processing NFs, the data encryption standard (DES) algorithm, and the rapid data compression (Gzip) algorithm, as well as VAs, were offloaded to TAPU for acceleration.

A. Hardware Implementation

We select Intel Max10 [19], a dual-image FPGA as the hardware board for TAPU.

```
def IMG_SWITCH_VIDEO():
    if (Configuration_Area_1 == VIDEO_IMG){
        Relay0 = 1;
    else if (Configuration_Area_2 == VIDEO_IMG)
        Relay0 = 0;
    Relay1 = 1;
```

Fig. 13. Code of API IMG_SWTICH_VIDEO().

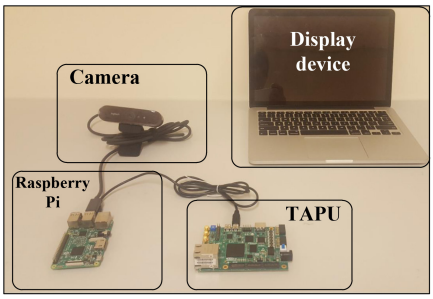

Fig. 14. Prototype implementation.

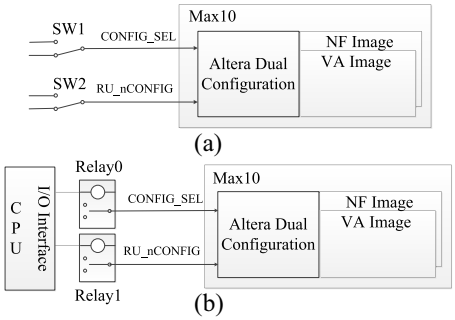

Fig. 15. Illustration of using relays to replace switches.

Integration of TAPU and Raspberry Pi: We interface Max10 to the Raspberry Pi. Max 10 has a USB Blaster that can be employed to connect the USB port of the Raspberry Pi by means of a USB-to-serial cable. We effectuate the communication between TAPU and Raspberry Pi using Thrift, an open-source flexible RPC interface. The RPC message is sent via the USB-to-serial cable. Furthermore, we delegate the responsibility for managing network packets in MAC and PHY layer to Max10, i.e., TAPU also functions as a network interface card (NIC) so that after processing, network packets can be expeditiously transferred to the cloud without wasting time for sending back processing result to NIC. We block the other NIC on the Raspberry Pi by editing the file "ifcfg-eth0" in Linux.[1]

FPGA Image Switch Control: TAPU provides the hardware API `IMG_SWITCH_VIDEO()` and `IMG_SWITCH_NETWORK()` which are used to switch programmed images in logic units. A particular challenge lies in the fact that the transition between images of FPGA must be achieved through automated commands. Nevertheless, in Max10, there are two control points, *SW*1 and *SW*2, see Fig. 15(a). *SW*1 is used to select the (next) image by the *CONFIG_SEL* pin and *SW*2 is used to trigger reconfiguration

[1]Intuitively, Raspberry Pi has two NICs and we only use the Max 10 NIC.

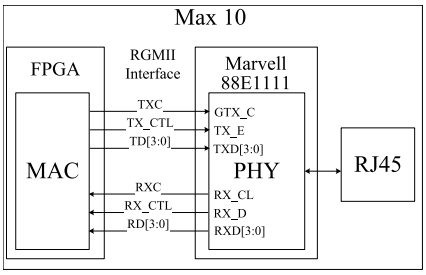

Fig. 16. Implementation of a processing network packet in MAC and PHY layer.

by the *RU_nCONFIG* pin. To finish a picture changeover procedure, *SW*1 and *SW*2 should be activated manually. We execute a hardware reconfiguration to substitute the manual switch *SW*1 and *SW*2 by relays depicted in Fig. 15(b). We remove the switch *SW*1 and *SW*2 and solider a relay at the original place of *SW*1 and *SW*2. We show the code of API `IMG_SWITCH_VIDEO` in Fig. 13, and the code of API `IMG_SWITCH_NETWORK` is similar. We can observe that when transitioning to the image stored in the flash area one, TAPU switches *CONFIG_SEL* pin by Relay0, then initiates the reconfiguration by lowering the *RU_nCONFIG* pins down by Relay1, and vice versa.

B. Network Functions Implementation

To achieve our goal of incorporating NFs DES and Gzip in a single image, we utilize the partial reconfiguration (PR) technique. This is necessary because only one image can be used for NFs in Max10 (the left one is allocated for VAs), and the logic units of Max10 FPGA are large enough to implement these two functions together. The kernel structure is based on a Verilog implementation by IBM described in [20].

We process network packets in both the MAC and PHY layers. In the MAC layer, we assemble Ethernet frames, which involves realigning the payload, modifying the source address, computing and appending the CRC-32 field, and inserting interpacket gap bytes. This process requires certain computation capability. In the PHY layer, we translate digital signals to analog signals, which necessitates a PHY chip. To address these challenges, we implement the MAC layer on the FPGA segment of Ma10, which has adequate computation capacity, and the PHY layer in the Marvell 88E111 chips of the Max 10 development board. The board includes a PHY chip, as shown in Fig. 16, and MAC communicates with PHY through RGMII interfaces. Subsequently, the PHY links the Ethernet cable through the modular connector RJ45 to transmit the packets.

C. Video Analytics Implementation

We have chosen to implement VAs using PipeCNN, an efficient OpenCL-based CNN accelerator on FPGAs. This accelerator provides a faster hardware development cycle and software-friendly program interfaces. We have conducted design space exploration to identify the optimal design that maximizes throughput or minimizes execution time, significantly reducing the development cycle. For our DNN

model, we have implemented YOLOv2, a state-of-the-art real-time object detection system that is widely used in VAs applications.

VII. Performance Evaluation

We use experiments based on real-world data to evaluate the implemented prototype.

A. Experiment Setup

Network Traffic Data Sets: For the data transferred from EdgeBox (i.e., Raspberry Pi) to the cloud, we use a real-world smart grid data set released in Smart* [21]. The Smart* data set is composed of UMass Smart* and UMass Smart* Microgrid. It comprises power data from three distinct Smart Homes, with electricity consumption and generation from wind turbines and solar panels recorded at one-second resolution, and electricity usage at the circuit level measured at a few-second interval. Additionally, the data set contains ambient information and some ground-truth data from the occupants. All types of data are regularly transferred to the cloud. For instance, the value of real-time voltage, which is sensed once per second, is uploaded to the cloud every 20 s.

VAs Setting: We configure the camera to capture 15 frames per second (FPS), with a resolution of 720 p. To detect animations in the video, we employ YOLOv2.

Evaluation Criteria: The network traffic of EdgeBox is regular. Thus, TAPU-based EdgeBox uses NPES by default.

Baseline: We compare TAPU with the following two approaches: 1) O-VA-FPGA offloads only VAs to Intel Max10 FPGA and implementing all NFs in software and 2) D-FPGA offloads VAs and NFs to two dedicated Intel Max10 FPGA without sharing.

Metric: First, we evaluate the throughput and latency of TAPU, O-VA-FPGA, and D-FPGA for VAs and network processing. Subsequently, we analyze the utilized ratio of FPGA, which is the ratio between the actually utilized FPGA resources and the total amount of FPGA resources. This ratio can demonstrate how effectively the FPGA resources have been utilized. Please note that the low utilized ratio of FPGA leads to the overprovision of hardware resource and increase the system cost. For example, when the utilization rate of a low-priced, low-computing FPGA is high, it can meet the performance requirements. There is no need to use a high-priced, high-computing chip. When the utilization rate is low, it will cause the overprovision of resources and increases the system cost. We evaluate the utilized ratio of the aforementioned three approaches.

B. Experiment Results

1) Throughput on Video Analytics: Fig. 17 shows that TAPU and D-FPGA outperform O-VA-FPGA in terms of VAs throughput. Specifically, the VAs throughput of TAPU and D-FPGA is 1.49 and 1.52 times that of O-VA-FPGA, respectively. This is attributed to the fact that advanced NFs take up most of the CPU resources, leaving a limited amount of CPU resources to be allocated for preprocessing of VAs, thus making it the bottleneck. Conversely, TAPU and D-FPGA are able to process the advanced NFs on FPGA, thus allowing more CPU resources to be allocated for preprocessing.

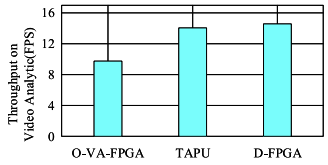

Fig. 17. Throughput on VAs.

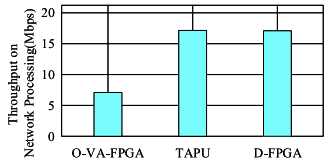

Fig. 18. Throughput on network processing.

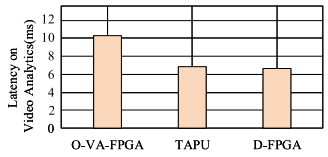

Fig. 19. Latency on VAs.

The throughput of TAPU (14.71 FPS) is observed to be slightly lower than that of D-FPGA (15 FPS). This discrepancy can be attributed to the fact that the FPGA in TAPU is shared between VAs and NFs, thus resulting in a minor difference in performance.

2) Latency on Video Analytics: In Fig. 19, the VAs latency of TAPU (67.98 ms), O-VA-FPGA, and D-FPGA are compared. We observe that TAPU and D-FPGA significantly outperform O-VA-FPGA. TAPU and D-FPGA have VAs latency that is 0.67 and 1.02 times that of O-VA-FPGA, respectively.

3) Throughput on Network Processing: Fig. 18 illustrates the network processing throughput of TAPU, O-VA-FPGA, and D-FPGA. It is evident that TAPU and D-FPGA significantly outperform O-VA-FPGA, with a throughput of 16.8 Mb/s, which is 2.33 times greater than that of O-VA-FPGA. This finding suggests that hardware acceleration is more effective than software-based processing, even when the majority of CPU resources are allocated for NFs.

Contrary to the results of VAs throughput, the network processing throughput of TAPU remains the same as that of D-FPGA. This is despite the fact that NFs and video share the same FPGA resource. This phenomenon can be attributed to TAPU's higher priority given to network processing over VAs. Specifically, FPGA resource is allocated to accelerate VAs only after the processing of network packets has been completed.

4) Latency on Network Processing: In Fig. 20, we compare the network processing latency of TAPU, O-VA-FPGA, and D-FPGA. As with the latency results on VAs, TAPU and D-FPGA demonstrate a significant improvement over O-VA-FPGA. TAPU and D-FPGA achieve a speed of 59.52 ms, which is 0.43 times faster than that of O-VA-FPGA.

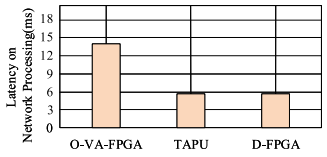

Fig. 20. Latency on network processing.

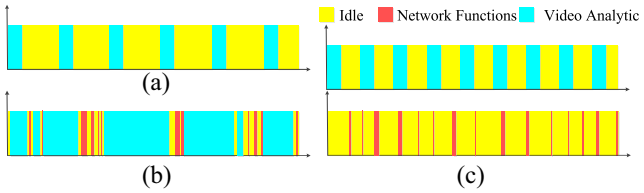

Fig. 21. Working status of FPGA. (a) O-VA-FPGA. (b)TAPU. (c) D-FPGA.

TABLE IV
AVERAGE UTILIZED RATIO OF FPGA

Approach	Utilized Ratio
O-VA-FPGA	37.29%
TAPU	92.51%
D-FPGA	36.32%

TABLE V
DEVELOPMENT EFFORT W/O SKELETON AMONG DIFFERENT APPLICATIONS

	APP.	*Skeleton Enable*	*# Code lines(Reduction Rate)*		
			Max10	*Artix-7*	*Max10+Artix-7*
Video	*VC*	×	21.1k	26.4k	47.5k
		✓	12.4k(-41.2%)	12.4k(-53.3%)	12.4k(-73.9%)
	VPD	×	32.5k	34.9k	67.4k
		✓	14.9k(-54.5%)	14.9k(-57.3%)	14.9k(-77.9%)
Network	*DES*	×	3.4k	4.5k	7.9k
		✓	2.2k(-58.8%)	2.2k(-51.1%)	2.2k(-72.1%)
	ZIP	×	5.2k	5.9k	11.1k
		✓	3.7k(-28.8%)	3.7k(-37.2%)	3.7k(-66.7%)

TABLE VI
COMPARISON OF TWO ESTIMATION SCHEME (V FOR VAS, N FOR NETWORK PROCESSING)

Scheme	Throughput (V)	Latency (V)	Throughput (N)	Latency (N)
NPES	14.71 FPS	67.98 ms	16.80 Mbps	59.52 ms
TDES	13.20 FPS	75.75 ms	16.95 Mbps	58.99 ms

TABLE VII
SWITCH TIME COMPARISON

Approach	Switch time
Dual-image FPGA	8.7 ms
Two single-image FPGA	9.6 ms

5) Utilized Ratio of FPGA: By comparing the FPGA utilization of TAPU, O-VA-FPGA, and D-FPGA, Fig. 21 shows the Gantt chart of FPGA working status. The yellow box in the figure indicates idle time, the red box indicates network packet processing, and the blue box indicates VAs. Ignoring the image switch time, which is relatively small (around 9 ms), Fig. 21(a) and (c) demonstrate that O-VA-FPGA and D-FPGA have a high ratio of idle FPGA, leading to inefficient utilization of resources. In contrast, TAPU has significantly less idle time, as shown in Fig. 21(b). This illustrates that TAPU can effectively utilize FPGA resources for both VAs and network processing.

As presented in Table IV, the FPGA utilized ratio in TAPU can reach up to 92.51%, which is 2.48 and 2.55 times higher than that of O-VA-FPGA and D-FPGA, respectively. Despite D-FPGA having the highest throughput due to over-provisioning, it has the lowest utilization. By comparison, TAPU has almost the same throughput as D-FPGA but without the need for over-provisioning, resulting in much higher utilization.

6) Application Development Effort: We investigate the potential of TAPU to facilitate the development of applications on dual-image FPGAs. To this end, we develop two applications, namely, VC and vehicles and pedestrians detection (VPD), as well as advanced NFs DES and ZIP on two dual-image FPGAs, namely, Intel Max10 and Xilinx Artix-7. We then compare the development effort in terms of the program length when programming with and without the skeleton provided by TAPU. The results of our study demonstrate the potential of TAPU to reduce the development effort for applications on dual-image FPGAs.

Table V presents the development effort of two applications on two different dual-image FPGAs. The results reveal two significant observations. First, the skeleton of TAPU can reduce a lot of development effort for all applications. For instance, the total lines of source code decrease by 41.2% and 54.5% for VC and VPD on Max10 when using skeleton.

Second, TAPU provides portability for application development. Specifically, with the skeleton, the code length of DES programmed on both Max10 and Artix-7 is consistent, measuring at 2.2 k lines, as shown in Table V. This demonstrates that the developer can reuse the code without modification for different FPGAs, thus enabling a higher degree of code portability.

7) Comparison Between Nonpreemption Estimation Scheme and Trace-Driven Estimation Scheme: We compare two estimation schemes NPES and TDES to demonstrate their performance in VAs and network processing. As shown in Table VI, we find that NPES performs better than TDES in VAs, and is inferior to TDES in network processing. This is because NPES assumes network traffic should not incur any preemption of VAs, which leads to network processing performance is not as good as TDES.

8) Runtime Overhead: We analyze the runtime overhead of TAPU from two aspects as follows.

Skeleton Overhead: The skeleton comprises several hardware functions that are designed to minimize development efforts, introducing a negligible delay when calling the function. Our experiments have revealed that the delay caused by calling the function is less than 1 ms, and the FPGA executes the task at the millisecond level, thus rendering the delay inconsequential.

FPGA Switch Overhead: The switching between dual-image FPGAs is unavoidable yet still quicker than utilizing two single-image FPGAs, as demonstrated in Table VII. This is because the image has been preloaded in dual-image FPGA and intermediary outcomes can be put away in RAM, while

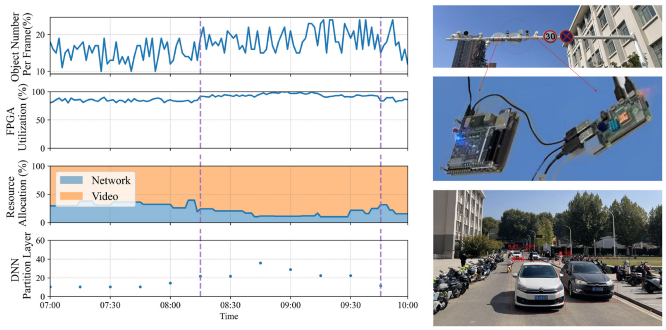

Fig. 22. End-to-end operations of TAPU in the campus traffic monitoring system. (Photograph has been informed and approved by the people in it and erased private information.)

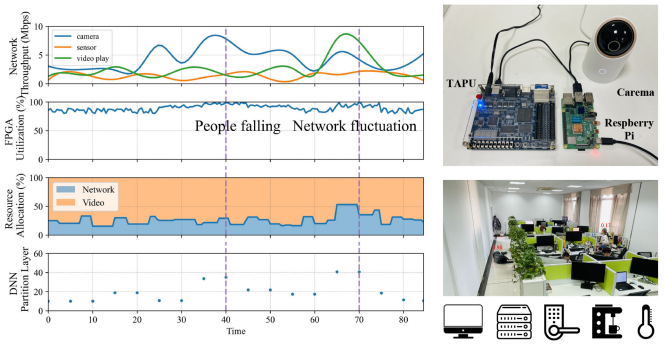

Fig. 23. End-to-end operations of TAPU in the office surveillance system. (Photograph has been informed and approved by the people in it and erased private information.)

two single FPGAs need to trade intermediary outcomes through I/O before continuing to the following stage.

C. Case Studies

To gain a better understanding of how TAPU performs in the real world, we have integrated it into two IoT gateways to enable different NFs and VAs applications. We have augmented a campus traffic monitor gateway with TAPU [22], for detecting vehicle speed and transmitting images of speeding vehicles to the server for storage, accountability, and punishment. Additionally, we have utilized TAPU to set up an office surveillance gateway [23], which is able to send the video of individuals who are in danger to the monitoring side of the health center and serve various applications using the network in the office. Furthermore, we have demonstrated the end-to-end operations of TAPU in both cases and all photographs has been approved by the people in it.

1) Campus Traffic Monitor System: We integrated TAPU into a campus traffic monitor gateway, as depicted in the right image of Fig. 22. This campus traffic monitor system deploys thirty Hikvision radar cameras at crucial roads and intersections within the 2 km^2 campus to detect speeding vehicles, which are recorded by a central server. It utilizes YOLOv2 to accomplish the license plate detection and vehicle speed measurement functions. The server is equipped with an Intel Core i9 11900K CPU and 120 GB of memory, running Ubuntu Server 18.04. Each camera is equipped with an ARM Cortex-A9 MPCore Soc and 8 GB memory, running on an embedded system, and connects to the server via a dedicated network to guarantee the system's security and stability. Since the network status is stable and no other applications are consuming network resources, the NPES is used to calculate the residual computation capacity for VAs. As all cameras share the resources of the server, the resource allocated to each camera is limited. Therefore, we employed the ATO algorithm of TAPU to support this application.

We collect the video of a camera between 7:00 AM and 10:00 AM as the data set for TAPU and show the end-to-end operation of TAPU in Fig. 22. The top plot shows the trend of object number per frame, which shows the fluctuation of workload. Between 8:15 and 9:15, due to the increase in vehicle and human traffic on weekdays, the object number per frame suddenly increases. The second plot shows the utilization status of the FPGA over a period of time. The utilization remains consistent due to the dedicated communication channel and increase as the workload gets heavier. The bottom two plots demonstrate the resource allocation in the FPGA and the partition layer change between the FPGA and the CPU, which is calculated by the ATO algorithm, and verify that TAPU successfully adjusts the partition layer during runtime, respectively. These changes are caused by dynamic changes in the number of vehicles at different times, which results in different numbers of DNN inference tasks and thus various resource allocations of the server to the cameras. This case demonstrates that TAPU can be effectively applied to large-scale, multiterminal IoT systems.

We estimate the overhead of TAPU further in this case. We find that the communication cost between TAPU and the IoT gateway is approximately 2–4 s, or 0.15% of computation time. These results demonstrate that TAPU has low overhead and can be effectively utilized to speed up the edge IoT gateway in real-world scenarios.

2) Office Surveillance System: We focus on applying TAPU to support an office surveillance system deployed in our laboratory. The system consists of Hikvision's AI-powered fall detection camera, an IoT gateway, and other edge devices running various applications connected to the gateway. The gateway receives the image from the camera and is responsible for the network traffic of the other IoT devices and applications. It has a pretrained model, Faster RCNN, to detect people within the monitored area. We tested the efficacy of TAPU in two scenarios: 1) *People Falling:* an individual enters the camera's field of view and simulates a fall and 2) *Network Fluctuation:* we used other applications to access the network and increase the load of the gateway processing network packets. As other applications in the edge devices also need to be connected to the Internet through the IoT gateway, this causes the network load of the gateway to vary significantly, thus necessitating a TDES.

As illustrated in Fig. 23, when people walk into the camera's field of view and fall down, the utilization of the TAPU increases to nearly 100%, due to the increase in VAs tasks. However, since the network situation remains unchanged, the time division of the FPGA image between the NF and the VAs does not fluctuate much. Consequently, due to the increased workload of VAs and the fixed estimated residual computation

capacity of the FPGA, some of the tasks can only be completed on the CPU, thus increasing the workload of the CPU in the DNN partition. Similarly, when the network load increases, the utilization of the FPGA also increases to nearly 100%. Since the TAPU prioritizes the NFs, the FPGA allocates more resources to the NFs image, thus reducing the resource obtained by the VAs image. This results in part of the VAs being done more on the CPU, leading to more layers of the DNN partition being calculated on the CPU.

In these two scenarios, we observed that the response time of TAPU to detect a fall decreased from 2 s to less than 1 s. Moreover, even when the network was congested, the time to recognize a fall did not exceed 2 s, thereby ensuring an effective and timely detection of falls. In conclusion, TAPU can be effectively deployed in smart fall detection systems to accelerate the detection process and to better handle different network conditions. We also estimated the overhead of TAPU in this case. We found that the communication cost between TAPU and the IoT gateway was approximately 1–2 ms, which was only 0.1% of the computation time. These results demonstrate that TAPU is lightweight and can be used to enhance the performance of edge IoT gateways.

VIII. Related Work

TAPU is an IoT gateway that utilizes a newly developed hardware, the multi-image FPGA enhancer, to offer accelerated video analysis and enhanced networking capabilities.

VAs Acceleration: VAs can be utilized for a broad array of applications in IoT gateways. It can be employed to detect motion, recognize objects, and monitor activities [24]. EC2Detect [25] is an edge–cloud collaborative real-time online video object detection method, utilizing a tracking-assisted object detection architecture based on edge–cloud collaboration with keyframe selection. Pathak et al. [26] proposed a distributed framework to orchestrate VAs across Edge and Cloud resources, exploring the tradeoffs between distributing various software components over a custom-built or complete system, and treating components on Edge and Cloud uniformly. ECHO [27] is an advanced big data platform that facilitates the transmission of IoT data streams between edge and cloud resources. It can deploy a variety of VAs applications to enable the implementation of smart city use cases. EdgeEye [28] is an edge-computing framework that enables the development of real-time intelligent VAs applications. It allows developers to convert models trained with deep learning frameworks into deployable components. DeepDecision [29] links front-end devices with more powerful back-end "helpers" (e.g., home servers) to enable deep learning to be executed either locally or remotely in the cloud/edge. Clio [30] presented a groundbreaking approach to dividing machine learning models between an IoT device and the cloud in a progressive way that is responsive to wireless dynamics.

VAs acceleration and hardware accelerators (such as TAPU) are orthogonal. Therefore, these video analysis methods, frameworks, and distributed video analysis solutions can improve the performance issues encountered by TAPU when processing video analysis tasks, thus supporting the development and implementation of IoT gateways.

NFs Acceleration: The requirement for network security and the need to reduce the amount of data transmission necessitate the complex running of NFs on the IoT gateway [31]. OmniMon [32] deftly orchestrates the cooperation between various entities across the entire network to carry out telemetry operations, making sure that the resource limitations of each entity are met without sacrificing absolute accuracy. A hierarchical SDN-based approach is proposed to expedite data management and balance the load not only between the IoT devices of a single domain but also between disparate network clusters [33]. For each service request requiring the integration of an IoT application and an NF virtualization into a location (e.g., a gateway or a small cloud), Xu et al. [34] presented an exact solution and an approximation algorithm with provable approximation ratios. To accelerate software packet processing, some studies exploited GPUs to achieve improved performance [35]. FlowShader [36] is a GPU acceleration framework designed to deliver both high generality and throughput even under highly skewed flow size distributions.

These NF acceleration methods can be combined with hardware accelerators (such as TAPU) to effectively enhance the NF of IoT gateways. For example, SDN-based hierarchical approaches can accelerate data management and load balancing. OmniMon's coordination approach can also ensure that the resource constraints of each entity are met.

FPGA Offloading: As flexible and customizable hardware, FPGA is increasingly being utilized as an accelerator for a variety of applications [37]. Many research efforts have explored how FPGA gas pedals can be leveraged to enhance the performance of applications. In particular, topics, such as how to reduce reconfiguration time [38], simplify the programming model of FPGAs [39], and optimize energy consumption, have been discussed in depth [40]. By utilizing FPGA gas pedals, it is possible to significantly enhance the efficiency of an application. Many have found that FPGA resources can be utilized to offload some CPU workloads and significantly decrease CPU resource consumption [41]. Zhang et al. [42] proposed offloading compactions to FPGAs to accelerate compactions and alleviate the CPU bottleneck when storing short key values. Enzian [43] linked a sizable FPGA with a server-class CPU in an asymmetric cache-coherent NUMA system. With the advent of applications in graph analytics, numerous accelerators have been proposed to efficiently support these workloads [44]. Fidas [45] is an FPGA-based intrusion detection offload system.

Relevant works focused on utilizing FPGA resources to offload CPU workloads, improve application efficiency and address issues, such as reconfiguration time and resource consumption. In contrast, TAPU utilizes multiple image FPGAs while considering both CPU and GPU computing resources to accelerate VAs and NF, addressing the challenges in current FPGA acceleration applications. To our knowledge, we are the first to configure and switch between multiple images of an FPGA with different functionalities.

IX. Conclusion

This article provides a comprehensive review of the current state of VAs and NFs in the IoT gateway, which require hardware accelerators to meet performance requirements. However, the dedicated accelerator for each function results in low utilization, and the IoT gateways usually have limitations in size and cost. To address this issue, we propose TAPU, a new multi-image FPGA accelerator that consists of both hardware and software designs to accelerate VAs and NFs, To maximally exploit the computation capacity of TAPU with minimal impact on the network processing, we developed algorithms for residual computation capacity estimation and efficient task offloading algorithms for VAs. We implement a prototype of TAPU and incorporate it into an IoT portal. The evaluation demonstrated that TAPU could reach up to 92% average utilization and enhance the throughput of VAs and NFs up to 1.49 times and 2.33 times, respectively. We further present two TAPU case studies based on network resource utilization. Finally, we predict that TAPU can be used on IoT gateways in many scenarios.

References

[1] O. Vermesan and J. Bacquet, *Next Generation Internet of Things: Distributed Intelligence at the Edge and Human Machine-to-Machine Cooperation*. Gistrup, Denmark: River Publ., 2019.

[2] "IoT 2022: Connected devices growing 18% to 14.4 billion globally." IoT Analytics. Accessed: Oct. 2022. [Online]. Available: https://www.iotforall.com/state-of-iot-2022

[3] A. Ghorayeb, R. Comber, and R. Gooberman-Hill, "Older adults' perspectives of smart home technology: Are we developing the technology that older people want?" *Int. J. Human Comput. Stud.*, vol. 147, Mar. 2021, Art. no. 102571.

[4] "Dell edge gateways for IoT." Accessed: May 2019. [Online]. Available: https://www.dell.com/en-uk/shop/gateways-embedded-computing/sf/edge-gateway

[5] "ADLINK IoT gateway." Accessed: May 2019. [Online]. Available: https://www.adlinktech.com/en/Industrial_IoT_and_Cloud_solutions_IoT_Gateway

[6] Z. Zhao, Z. Jiang, N. Ling, X. Shuai, and G. Xing, "ECRT: An edge computing system for real-time image-based object tracking," in *Proc. ACM SenSys*, Shenzhen, China, Nov. 2018, pp. 394–395.

[7] F. Loewenherz, V. Bahl, and Y. Wang, "Video analytics towards vision zero," *ITE J.*, vol. 87, no. 3, p. 25, 2017.

[8] J. Qiu et al., "Going deeper with embedded FPGA platform for convolutional neural network," in *Proc. ACM FPGA*, Monterey, CA, USA, Feb. 2016, pp. 26–35.

[9] X. Li, X. Wang, F. Liu, and H. Xu, "DHL: Enabling flexible software network functions with FPGA acceleration," in *Proc. IEEE ICDCS*, Vienna, Austria, Jul. 2018, pp. 906–910.

[10] S. Tanwir, D. Nayak, and H. Perros, "Modeling 3D video traffic using a Markov modulated gamma process," in *Proc. Int. Conf. Comput. Netw. Commun. (ICNC)*, Kauai, HI, USA, Feb. 2016, pp. 1–6.

[11] H. Park, T. Kim, and S. Kim, "Network traffic analysis and modeling for games," in *Proc. 1st Int. Workshop Internet Netw. Econ. (WINE)*, Hong Kong, Dec. 2005, pp. 1056–1065.

[12] M. Ebrahimi and M. Daneshtalab, "EbDa: A new theory on design and verification of deadlock-free interconnection networks," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 703–715, 2017.

[13] D. Lee, S. Zhou, X. Zhong, Z. Niu, X. Zhou, and H. Zhang, "Spatial modeling of the traffic density in cellular networks," *IEEE Wireless Commun.*, vol. 21, no. 1, pp. 80–88, Feb. 2014.

[14] W. Feng et al., "Study on multi-network traffic modeling in distribution communication network access service," in *Proc. IEEE ICACT*, Mumbai, India, Feb. 2018, pp. 720–723.

[15] L. Tedesco, A. Mello, L. Giacomet, N. Calazans, and F. Moraes, "Application driven traffic modeling for NoCs," in *Proc. ACM SBCCI*, Belo Horizonte, Brazil, Sep. 2006, pp. 62–67.

[16] A. Sang and S.-Q. Li, "A predictability analysis of network traffic," *Comput. Netw.*, vol. 39, no. 4, pp. 329–345, 2002.

[17] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-ResNet and the impact of residual connections on learning," in *Proc. AAAI*, San Francisco, CA, USA, Feb. 2017, pp. 4278–4284.

[18] B. V. Cherkassky and A. V. Goldberg, "On implementing push–relabel method for the maximum flow problem," in *Proc. Springer IPCO*, 1995, pp. 157–171.

[19] "Intel MAX 10 FPGA development kit." Accessed: Jun. 2018. [Online]. Available: https://www.altera.com/products/boards_and_kits/dev-kits/altera/max-10-fpga-development-kit.html

[20] A. Martin, D. Jamsek, and K. Agarawal, "FPGA-based application acceleration: Case study with gzip compression/decompression streaming engine," in *Proc. ICCAD Special Session C*, vol. 7, 2013, p. 2013.

[21] S. Barker, A. Mishra, D. Irwin, E. Cecchet, P. Shenoy, and J. Albrecht, "Smart*: An open data set and tools for enabling research in sustainable homes," in *Proc. SustKDD*, vol. 111, 2012, p. 108.

[22] J. Barthélemy, N. Verstaevel, H. Forehead, and P. Perez, "Edge-computing video analytics for real-time traffic monitoring in a smart city," *Sensors*, vol. 19, no. 9, p. 2048, 2019.

[23] Y. Liu, L. Kong, G. Chen, F. Xu, and Z. Wang, "Light-weight AI and IoT collaboration for surveillance video pre-processing," *J. Syst. Archit.*, vol. 114, Mar. 2021, Art. no. 101934.

[24] M. Hanyao, Y. Jin, Z. Qian, S. Zhang, and S. Lu, "Edge-assisted online on-device object detection for real-time video analytics," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, 2021, pp. 1–10.

[25] S. Guo, C. Zhao, G. Wang, J. Yang, and S. Yang, "EC^{2}detect: Real-time online video object detection in edge-cloud collaborative IoT," *IEEE Internet Things J.*, vol. 9, no. 20, pp. 20382–20392, Oct. 2022.

[26] T. Pathak, V. Patel, S. Kanani, S. Arya, P. Patel, and M. I. Ali, "A distributed framework to orchestrate video analytics across edge and cloud: A use case of smart doorbell," in *Proc. 10th Int. Conf. Internet Things*, 2020, pp. 1–8.

[27] A. Khochare, P. Ravindra, S. P. Reddy, and Y. Simmhan, "Distributed video analytics across edge and cloud using," in *Proc. Serv. Oriented Comput. (ICSOC) Workshops*, Málaga, Spain, 2018, pp. 402–407.

[28] P. Liu, B. Qi, and S. Banerjee, "EdgeEye: An edge service framework for real-time intelligent video analytics," in *Proc. 1st Int. Workshop Edge Syst. Anal. Netw.*, 2018, pp. 1–6.

[29] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen, "DeepDecision: A mobile deep learning framework for edge video analytics," in *Proc. IEEE INFOCOM*, Honolulu, HI, USA, Apr. 2018, pp. 1421–1429.

[30] J. Huang, C. Samplawski, D. Ganesan, B. Marlin, and H. Kwon, "CLIO: Enabling automatic compilation of deep learning pipelines across IoT and cloud," in *Proc. 26th Annu. Int. Conf. Mobile Comput. Netw.*, 2020, pp. 1–12.

[31] L. Linguaglossa et al., "Survey of performance acceleration techniques for network function virtualization," *Proc. IEEE*, vol. 107, no. 4, pp. 746–764, Apr. 2019.

[32] Q. Huang, H. Sun, P. P. C. Lee, W. Bai, F. Zhu, and Y. Bao, "OmniMon: Re-architecting network telemetry with resource efficiency and full accuracy," in *Proc. ACM SIGCOMM*, 2020, pp. 404–421.

[33] Z. Eghbali and M. Z. Lighvan, "A hierarchical approach for accelerating IoT data management process based on SDN principles," *J. Netw. Comput. Appl.*, vol. 181, May 2021, Art. no. 103027.

[34] Z. Xu, W. Gong, Q. Xia, W. Liang, O. F. Rana, and G. Wu, "NFV-enabled IoT service provisioning in mobile edge clouds," *IEEE Trans. Mobile Comput.*, vol. 20, no. 5, pp. 1892–1906, May 2021.

[35] K. Xie et al., "Accurate and fast recovery of network monitoring data with GPU-accelerated tensor completion," *IEEE/ACM Trans. Netw.*, vol. 28, no. 4, pp. 1601–1614, Aug. 2020.

[36] X. Yi et al., "FlowShader: A generalized framework for GPU-accelerated VNF flow processing," in *Proc. IEEE 27th Int. Conf. Netw. Protocols (ICNP)*, 2019, pp. 1–12.

[37] A. Imteaj, U. Thakker, S. Wang, J. Li, and M. H. Amini, "A survey on federated learning for resource-constrained IoT devices," *IEEE Internet Things J.*, vol. 9, no. 1, pp. 1–24, Jan. 2022.

[38] A. Kamaleldin et al., "Design guidelines for the high-speed dynamic partial reconfiguration based software defined radio implementations on Xilinx Zynq FPGA," in *Proc. IEEE ISCAS*, Baltimore, MD, USA, May 2017, pp. 1–4.

[39] S. Wang and Y. Liang, "A framework for iterative stencil algorithm synthesis on FPGAs from OpenCL programming model," in *Proc. ACM/SIGDA FPGA*, Monterey, CA, USA, Apr. 2017, pp. 285–286.

[40] S. Jiang et al., "Accelerating mobile applications at the network edge with software-programmable FPGAs," in *Proc. IEEE INFOCOM*, Honolulu, HI, USA, Apr. 2018, pp. 55–62.
[41] D. Gizopoulos et al., "Modern hardware margins: CPUs, GPUs, FPGAs recent system-level studies," in *Proc. IEEE IOLTS*, Rhodes, Greece, 2019, pp. 129–134.
[42] T. Zhang et al., "FPGA-accelerated compactions for LSM-based key-value store," in *Proc. FAST*, 2020, pp. 225–237.
[43] D. Cock et al., "Enzian: An open, general, CPU/FPGA platform for systems software research," in *Proc. ACM ASPLOS*, Amsterdam, The Netherlands, Feb. 2022, pp. 434–451.
[44] R. Ramezani, "A prefetch-aware scheduling for FPGA-based multi-task graph systems," *J. Supercomput.*, vol. 76, no. 9, pp. 7140–7160, 2020.
[45] J. Chen et al., "Fidas: Fortifying the cloud via comprehensive FPGA-based offloading for intrusion detection: Industrial product," in *Proc. 49th Annu. Int. Symp. Comput. Archit.*, 2022, pp. 1029–1041.

Huanghuang Liang received the B.S. degree in automation engineering from Anhui University of Technology, Ma'anshan, China, in 2016, and the M.S. degree in automation engineering from the University of Electronic Science and Technology of China, Chengdu, China, in 2019. He is currently pursuing the Ph.D. degree in computer science with Wuhan University, Hubei, China.

His research interests include cloud computing and big data systems.

Qianlong Sang received the B.S. degree in cyber science and engineering from Wuhan University, Hubei, China, in 2022, where he is currently pursuing the Ph.D. degree in computer science.

His research interests include edge computing and big data systems.

Chuang Hu (Member, IEEE) received the B.S. and M.S. degrees in computer science from Wuhan University, Hubei, China, in 2013 and 2016, respectively, and the Ph.D. degree from The Hong Kong Polytechnic University, Hong Kong, in 2019.

He is currently an Associate Researcher with the School of Computer Science, Wuhan University. His research interests include edge learning, federated learning/analytics, and distributed computing.

Yili Gong (Member, IEEE) received the B.S. degree in computer science from Wuhan University, Hubei, China, in 1998, and the Ph.D. degree in computer architecture from the Institute of Computing, Chinese Academy of Sciences, Beijing, China, in 2006.

She is currently an Associate Professor with the School of Computer Science, Wuhan University. Her interests include intelligent operations and maintenance in HPC environments, distributed file systems, and blockchain systems.

Dazhao Cheng (Senior Member, IEEE) received the B.S. degree in electrical engineering from the Hefei University of Technology in 2006, the M.S. degree in electrical engineering from the University of Science and Technology of China, Chengdu, China, in 2009, and the Ph.D. degree from the University of Colorado, Colorado Springs, CO, USA, in 2016.

He was an Assistant Professor with the University of North Carolina at Charlotte, Charlotte, NC, USA, from 2016 to 2020. He is currently a Professor with the School of Computer Science, Wuhan University, Hubei, China. His research interests include big data and cloud computing.

Xiaobo Zhou (Senior Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science from Nanjing University, Nanjing, China, in 1994, 1997, and 2000, respectively.

He was a Professor with the Department of Computer Science, University of Colorado, Colorado Springs, CO, USA. He is currently a Distinguished Professor with the State Key Laboratory of Internet of Things for Smart City and the Department of Computer and Information Sciences, University of Macau, Macau, China. His research interests include distributed systems, cloud computing and datacenters, data parallel and distributed processing, and autonomic.

Prof. Zhou was the recipient of the NSF CAREER Award in 2009.

Yu Wang (Fellow, IEEE) received the B.S. and M.S. degrees in computer science from Tsinghua University, Beijing, China, in 1998 and 2000, respectively, and the Ph.D. degree in computer science from Illinois Institute of Technology, Chicago, IL, USA, in 2004.

He is currently a Professor with the Department of Computer and Information Sciences, Temple University, Philadelphia,, PA, USA. His research interests include wireless networks, smart sensing, and mobile computing.

Prof. Wang is a recipient of the Ralph E. Powe Junior Faculty Enhancement Awards from Oak Ridge Associated Universities in 2006, the Outstanding Faculty Research Award from the College of Computing and Informatics at the University of North Carolina at Charlotte in 2008, and the ACM Distinguished Member in 2020. He is also a Senior Member of ACM.