UNIVERSITY OF CALIFORNIA,
IRVINE

System Software Support for FPGA-based Multi-Accelerator Architectures in Edge Computing Systems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Hsin-Yu Ting

2023

DEDICATION

To my wife, Allison, and my family.
Your love and support have been the foundation of my achievements.

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor, Professor Eli Bozorgzadeh, for her invaluable insights and dedicated mentorship throughout my doctoral journey.

I would like to thank Professor Alex Nicolau and Professor Ian G. Harris for their time, support, and feedback in serving as members of my dissertation committee.

It has been a great pleasure collaborating with Professor Ardalan Amiri Sani, Dr. Ahmad Razavi, Tootiya Giyahchi, Mihnea Chirila, and Leming Cheng. I extend my sincere thanks for their significant contributions to our endeavors.

I thank the Institute of Electrical and Electronics Engineers (IEEE) and the Association for Computing Machinery (ACM) for giving me permission to include my previously published papers in this dissertation.

Finally, the endeavor would not have been possible without the support of my wife and my family. I would especially like to thank my wife, Allison. Her encouragement has been the driving force behind my accomplishments, and I'm grateful for her presence throughout this journey.

VITA

Hsin-Yu Ting

EDUCATION

Doctor of Philosophy in Computer Science **2023**
University of California, Irvine *Irvine, California*

Master of Science in Computer Science **2013**
National Tsing Hua University *HsinChu, Taiwan*

Bachelor of Science in Computer Science **2011**
National Yang Ming Chiao Tung University *HsinChu, Taiwan*

RESEARCH EXPERIENCE

Graduate Research Assistant **2016–2023**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant **2016–2023**
University of California, Irvine *Irvine, California*

PUBLICATIONS

S. A. Razavi, H.-Y. Ting, T. Giyahchi, and E. Bozorgzadeh. On exploiting patterns for robust fpga-based multi-accelerator edge computing systems. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 116–119. IEEE, 2022

M. Chirila, P. D'Alberto, H.-Y. Ting, A. Veidenbaum, and A. Nicolau. A heterogeneous solution to the all-pairs shortest path problem using fpgas. In *2022 23rd International Symposium on Quality Electronic Design (ISQED)*, pages 108–113. IEEE, 2022

H.-Y. Ting, T. Giyahchi, A. A. Sani, and E. Bozorgzadeh. Dynamic sharing in multi-accelerators of neural networks on an fpga edge device. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 197–204. IEEE, 2020

H.-Y. Ting, A. A. Sani, and E. Bozorgzadeh. System services for reconfigurable hardware acceleration in mobile devices. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE, 2018

ABSTRACT OF THE DISSERTATION

System Software Support for FPGA-based Multi-Accelerator Architectures in Edge Computing Systems

By

Hsin-Yu Ting

Doctor of Philosophy in Computer Science

University of California, Irvine, 2023

Professor Eli Bozorgzadeh, Chair

Edge computing plays a key role in providing low latency and high availability services for emerging Internet of Things (IoT) applications. Recently, IoT devices are harnessing Deep Neural Networks (DNNs) to empower intelligence capability. With the increasing computing demands on IoT DNN applications, edge computing systems have evolved and adapted to employ hardware accelerators to enhance the processing power alongside the multi-core processors. Among hardware accelerators, Field Programmable Gate Arrays (FPGAs) have gained increased attention due to their reconfigurability, high performance, and power efficiency. The FPGA resource allows multiple workloads spatially co-locating and concurrently executing on the shared FPGA resource. With the performance variability due to application requirements and contention for the limited shared computing resources, the edge system poses significant challenges for efficiently scheduling and allocating resources for application tasks. In the absence of proper management, the system could turn to sub-optimal resource partitioning and utilization, and increase latency for completing applications. Therefore, modern edge systems require a flexible and efficient mechanism to dynamically partition shared resources and schedule tasks based on application requirements and available resources. However, in various IoT monitoring systems, sensing occurs at a fixed rate, and, hence, the data is sent to the edge for acceleration periodically. When multiple IoT devices continuously send acceleration requests to the edge, characteristics of applications in patterns can be observed. Such regularity in patterns provides optimization opportunities for the system to support multi-tenancy in sharing computing resources.

In the first part of the dissertation, I discuss the emerging edge computing systems that facilitated IoT end devices' compute-intensive tasks, i.e. DNNs, to be offloaded to the edge. I address the efforts to the integration with FPGAs and the deployment of DNNs on the resource-constrained edge. While efficiently managing task scheduling and resource allocation among various concurrent DNN applications co-locating on a multi-accelerator edge system becomes challenging, I discuss the current efforts made in the related resource management approaches for the multi-tenant FPGA-based edge system and their limitations.

Given that DNN applications often have similar or shared types of requirements, e.g. dataset types and accuracy, I focus on developing a DNN-Accelerator sharing system at the FPGA edge device, that serves various DNN applications from multiple end devices simultaneously. The proposed *SharedDNN/PlanAhead* policy exploits the regularity among requests for various DNN accelerators and determines which accelerator to allocate for each request and in what order to respond to the requests that achieve maximum responsiveness for a queue of acceleration requests. My proposed framework exploits a priori known pattern of input task arrivals and matches the suitable accelerators to the tasks according to the utilization of shared FPGA resources and application's requirements.

When multiple end devices are consistently sending tasks to the edge, there exist patterns of applications in the task queue on the edge. I, then, present a systematic approach that exploits the characteristics of applications in patterns and employs a mixed offline/online multi-queue scheduling method to optimize responsiveness by reducing response time and minimizing task drops for consistent IoT DNN workloads. The proposed framework not only exploits the regularity of the historical data and extracts patterns but also provides an adaptive online scheduler to mitigate the effect of noises and fluctuation due to network delays and system workload congestion.

Lastly, considering that IoT applications can be event-driven, characterized by varying task rates changing over time, the input workload experiences dynamic and uncertain changes. To dynamically adapt to uncertainty and changes in input workloads, I demonstrate a learning-based multi-accelerator management framework that asynchronously learns the scheduling and allocation policy and dynamically partitions shared resources to maximize system performance through interaction with the edge.

In this dissertation, I investigate the FPGA-based multi-accelerator management system software to schedule and allocate tasks onto FPGA edge systems in the presence of various IoT DNN workloads. The experimental results show significant improvements in response time and task drops by exploiting the regularity of input workloads and deploying the mixed offline/online-phase system software. In addition, when the workload experiences dynamic and uncertain changes where the regularity of workloads becomes unpredictable and hard to extract and generalize, I present a learning-based FPGA-based multi-accelerator management framework that allows the system to capture these dynamics and find an adaptive scheduling policy to accommodate the uncertainty. The experimental results show improved average throughput and task drop rates compared to other state-of-the-art heuristic and learning-based approaches.

Chapter 1

Introduction

In recent years, the Internet of Things (IoT) techniques have become widely adopted. Internet of Things (IoT) refers to a network of devices, embedded with sensors and software and connected to the internet, that, hence, can communicate and coordinate between them and their users. This integration of IoT technologies spans various sectors, encompassing smart homes, healthcare, industrial automation, smart cities, and more (See Figure 1.1). These smart connected devices have the capability to collect, share, and analyze information and may take relevant actions based on the insights accordingly. To achieve this, IoT devices utilize sensors and processors to gather and analyze data acquired from their environments. The collected data will be shared by sending to either a gateway or other IoT devices. Subsequently, the data can be analyzed locally or sent to the cloud for further processing [62, 114]. For instance, smart surveillance cameras, integrated into the parking structures and streets, for smart cities can utilize IoT capabilities to monitor public spaces. These cameras can not only capture real-time footage but also analyze the data locally to identify potential security threats or unusual activities. The collected data can also be transmitted to a centralized cloud infrastructure for more in-depth analysis and long-term storage.

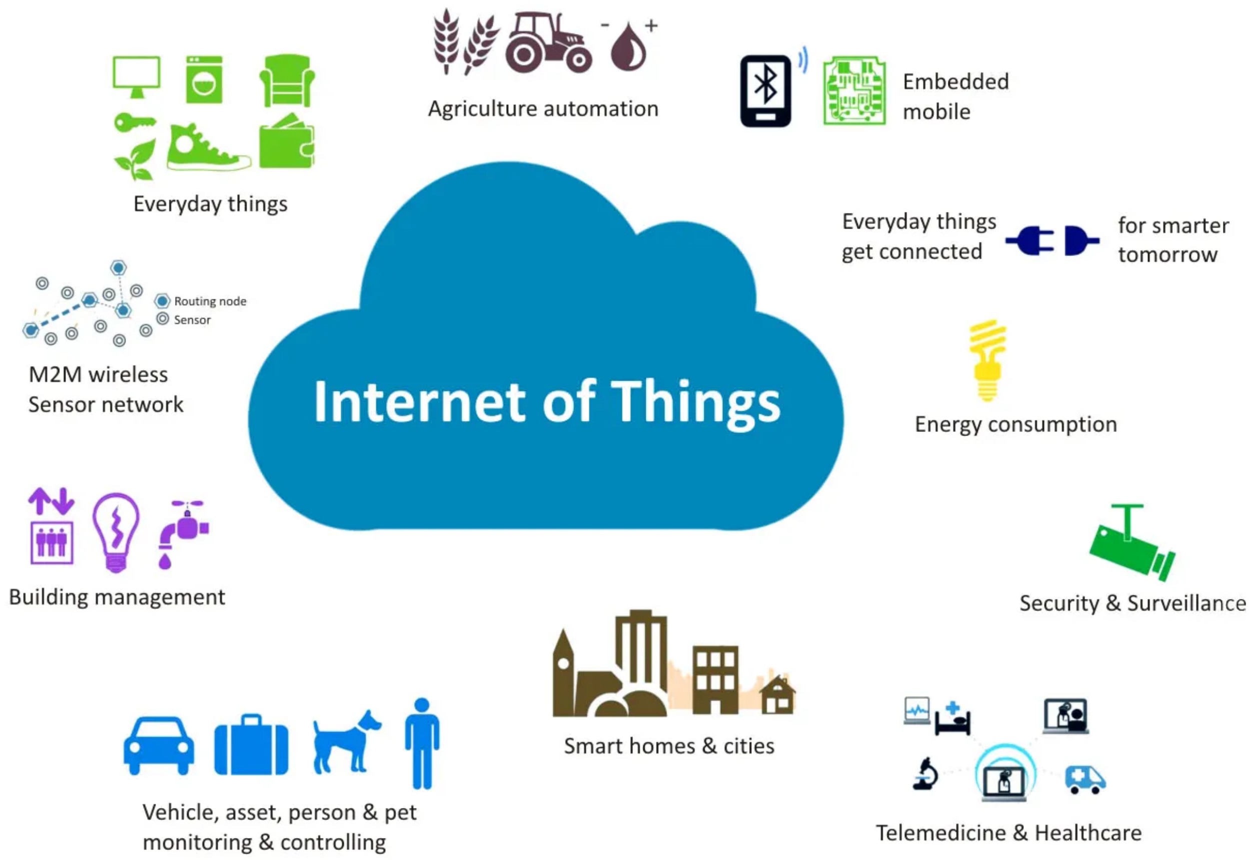

Figure 1.1: Applications of IoT in diverse fields (Source from [118])

Moreover, according to recent statistics reported by Statista [92], the number of IoT devices in the world will reach over 15 billion by 2023, which has surpassed the current number of people on Earth. It is forecast to be close to 30 billion by 2030 and would be a nearly double increase from the IoT installed base in 2023. Another report on Statista [43] shows the forecast of more than 75 billion IoT-connected devices in use by 2025. These numbers are forecast differently. Getting the statistics on the actual numbers is difficult, but from all these varying stats we can at least establish facts that the era of IoT technology is becoming and the market is growing. While IoT devices heavily rely on cloud computing for processing data, the surge in IoT devices presents challenges in managing and scaling centralized cloud infrastructures. On the other hand, the surge in IoT devices is generating an unprecedented amount of data, expected to reach 175 zettabytes (ZB) by 2025, according to a report by IDC [40]. While this data holds immense potential for insights and innovation, the volume of data poses significant challenges in terms of management and analysis. This data surge also places a substantial burden on traditional cloud computing infrastructures where centralized cloud servers deal with the vast volume and variability of data, leading to increased latency and potential bottlenecks. In addition, the data generated by IoT devices can be sensitive and personal, raising concerns about privacy and security. Therefore, there is a pressing need for more efficient approaches to alleviate the strain on centralized servers and process and analyze such vast and sensitive data.

1.1 Edge Computing in IoT Systems

Over the years, various concepts, such as VM-based cloudlets [115], and fog/edge computing [35, 113, 32, 1], have been proposed from different perspectives to mitigate reliance on remote cloud infrastructure. Since the VM-based cloudlets [115] was proposed, there have been great efforts made to explore CPU-based edge offloading across various types of applications.

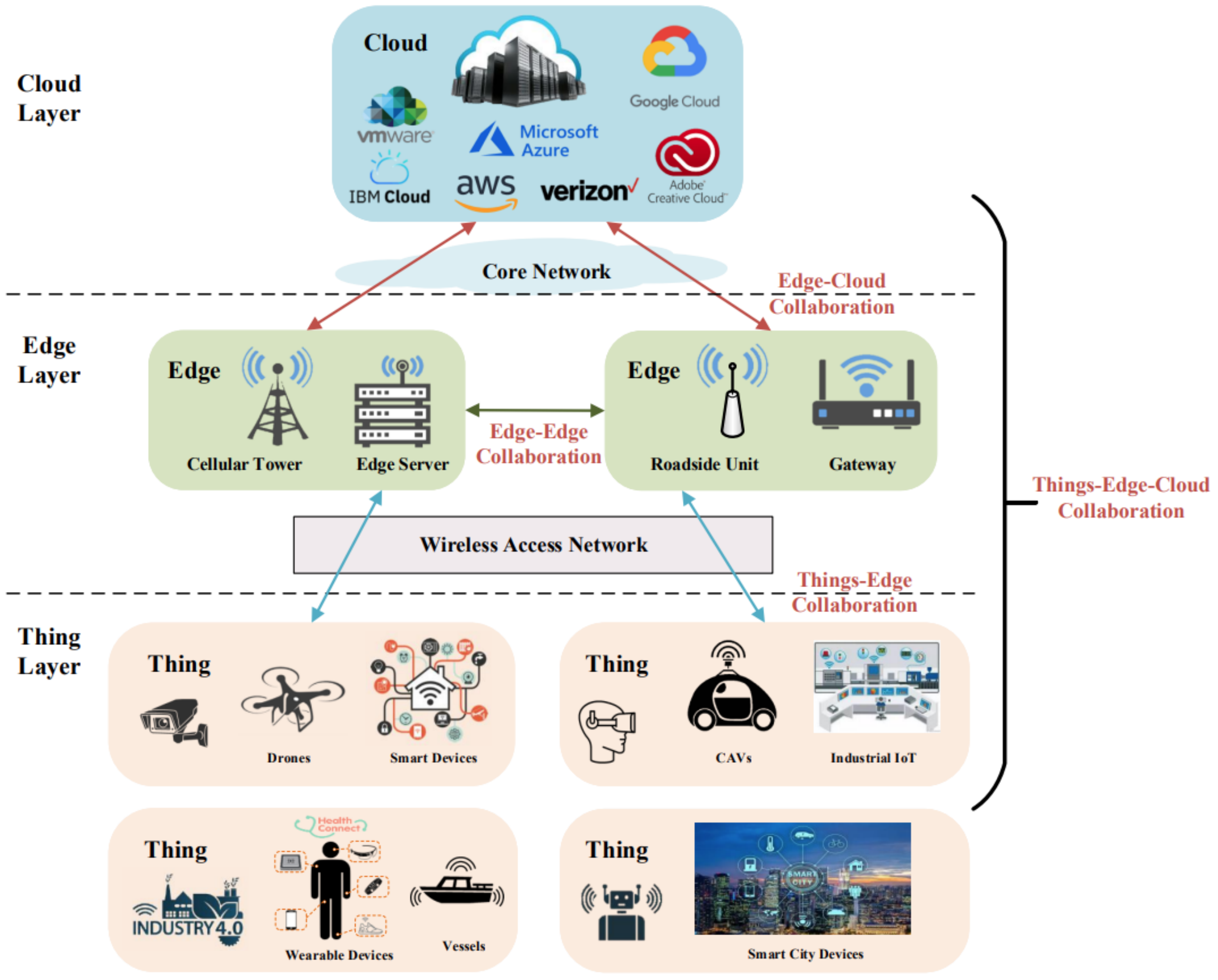

Figure 1.2: The edge nodes collaborated between themselves, end devices, and cloud nodes (Source from [74])

Edge computing has emerged as a promising solution to address this challenge. The technique involves decentralized processing, which is a distributed computing paradigm that brings analysis and data storage closer to the data source, reducing the amount of data that needs to be transmitted to the cloud [99, 147, 69]. As reference in [74], Figure 1.2 illustrates how edge computing can coordinate between the IoT end, edge devices, and the cloud.

In edge computing, an edge node is a generic term that refers to any device, server, or gateway that performs edge computing. According to IBM's definition [38], an edge device may simply have ARM or x86 class CPUs with 1 or 2 cores, and 128 MB of memory for special-purpose work. Or it can be constructed with an industrial PC with 8, 16, or more cores of compute capacity, and 16 GB of memory, which is typically used to run compute-intensive application workloads and share services. For a use case example, consider a smart warehouse security system that relies on cloud-based processing for analyzing video feeds from multiple cameras. In a traditional setup, each camera captures footage and sends data to the cloud 24 hours per day for analysis. The cloud servers process the video data, perform object detection, and generate alerts for any suspicious activities. Using edge computing, the data are analyzed directly at the local edge. The edge devices may only send relevant information, such as detected anomalies or objects, to the cloud. This significantly reduces the amount of data that needs to be transmitted to the cloud, which can help alleviate the burden on the cloud and reduce network congestion. This approach can also help improve response times and reduce latency, which can be critical in such security monitoring applications.

1.2 Deep Neural Network Applications in Edge Computing

While edge computing enables real-time analysis of data, the IoT applications that run Deep neural networks (DNNs) approaches benefit significantly from edge computing given its capability of analyzing data in a timely manner. Deep neural networks, which are a subset of machine learning algorithms, have been widely used since DNNs are capable of processing big data and identifying complex patterns, making them ideal for applications that require predictive analysis and decision-making in real-time. The fusion of IoT technologies with the power of DNNs has led to great advancements in various domains [112]. For instance, in smart healthcare, DNNs can be used to detect and diagnose early signs of skin cancer and Alzheimer's disease [126]. In smart cities, the intelligent traffic monitoring system uses DNNs to monitor traffic flows [75, 140] and detect objects, such as pedestrians, on-road vehicles, and unattended objects [77], or road accidents [17, 123]. Various methodologies of using DNNs are studied to detect anomalies in intelligent video surveillance [96].

In DNN applications, the evolution of computer vision tasks has come to big accomplishment, particularly following the success of a CNN-based model, called AlexNet [61], for image classification. Subsequent advancements are like VGG [122], GoogLeNet [124], ResNet [30], MobileNet [34], Inception [125], as well as other types of models tailored for object detection, such as YOLO [108, 109], SSD [71]. These DNN models, resulting in a variety of accuracy and compute complexity (which affects performance and resource utilization), have created a huge design space. Figure 1.3 (Source from [8]) shows the representative neural networks in recent years regarding their inference accuracy, compute complexity (i.e. amount of operations required for a single forward pass), and the number of network parameters.

Also, there is a variety of neural network architectures proposed, including Convolutional Neural Network (CNN) [64], Recurrent Neural Network (RNN) [121], Long Short Term

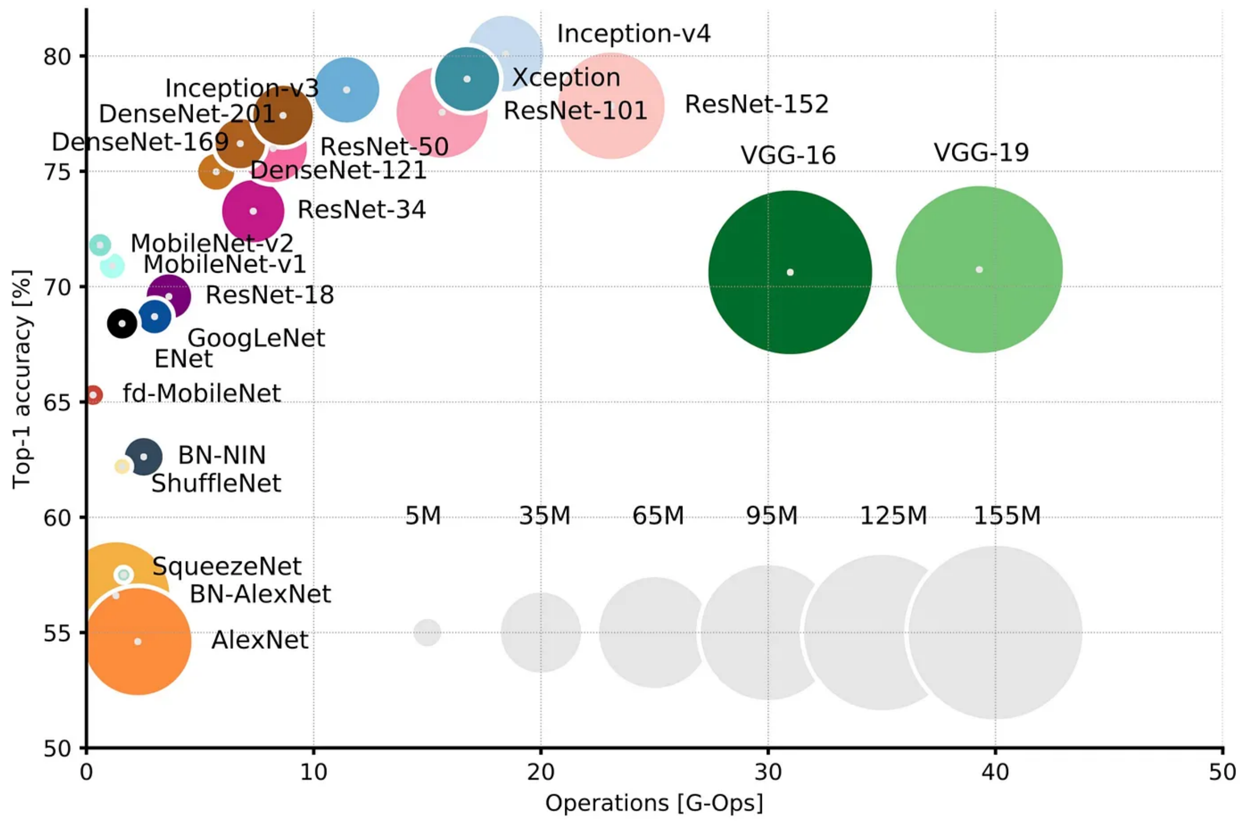

Figure 1.3: Variety of CNN-based models. (The size of the blobs is proportional to the number of network parameters) (Source from [8])

Memory (LSTM) [121], etc. Each of them has demonstrated significant advancements in their respective application domains. For example, CNNs excel in image processing tasks, RNNs prove effective in sequential data analysis, and LSTMs address challenges in modeling long-term dependencies. The variability of DNNs allows them to surpass in diverse application scenarios, each benefiting from the strengths of specific neural network architectures.

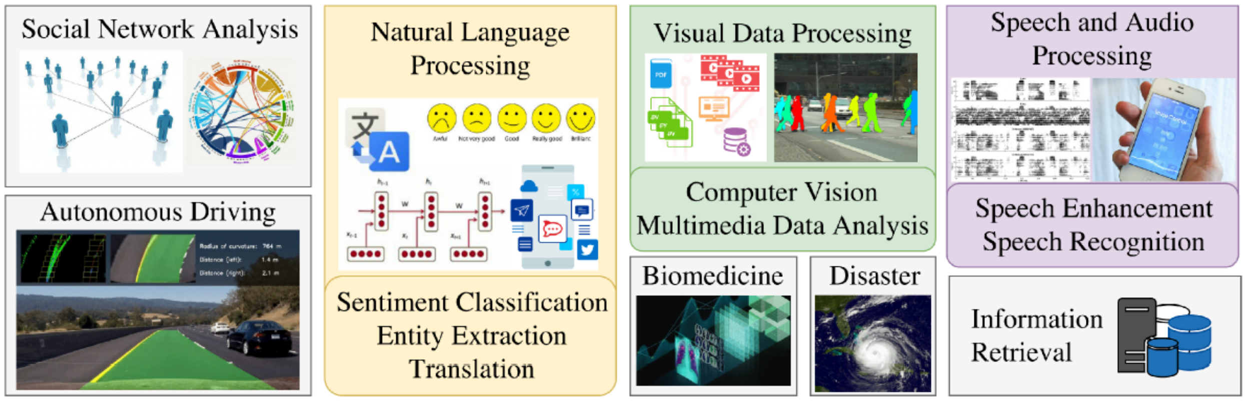

Figure 1.4: Some of the Popular DNN applications (Source from [98])

These DNN techniques are applied in IoT systems and have demonstrated exceptional accuracy across many other domains, including image analysis, speech analysis, recommendation systems, medical data analysis, etc [138, 69, 117, 98]. As referred in [98], some of the main deep learning applications are visualized in Figure 1.4.

In recent years, the evolution of DNNs has led a trend towards increasingly larger models, driven by the pursuit of enhanced accuracy and the ability to handle more complex tasks. As the demand for pattern extraction, natural language processing, and sophisticated decision-making grows, researchers are developing larger DNN architectures with higher numbers of parameters to capture more features within data and provide complex data analysis.

While DNN techniques are known powerful and growing more complex in processing big data, the computational demands and memory overhead associated become concerns for resource-constrained devices, which leads to efforts in investigating lightweight modelings. Hyperdimensional Computing (HDC) is an emerging computational model, that utilizes high-dimensional patterns for learning tasks, differing from traditional deep neural networks that rely on layered architectures. The technique exhibits advantages such as energy efficiency and smaller model size; however, it still has limitations and currently achieves sub-par capability in complex applications compared to neural networks [76]. We agree that HDC has the potential to address the challenge of learning tasks on limited computation resources, but it's still a relatively new field and has not yet been widely adopted. DNNs are still dominant in edge computing. Therefore, in this dissertation, we are focusing on the variants of optimized Deep Neural Network models. To optimize DNN models, various techniques such as model compression, pruning, quantization, generalization, and weight-sharing have been explored [56]. These approaches aim to enhance the efficiency of deep learning models and may sacrifice accuracy, making them more suitable for deployment on devices with limited computational resources and memory capacity.

The combination of DNN model optimization techniques causes a larger and more expansive design space, depending on other choices in architectures, hyperparameter tuning, and training methodologies. This huge design space provides opportunities for a wide range of DNN models to accommodate any specific application requirements. A designer can decide DNN models according to the metrics of their application domains, architectural types, and configurations, and adjust hyperparameters, such as learning rates, layer size, and activation functions, to further optimize the model. However, such a wide range of possibilities introduces challenges for the designer to find an optimal fit for DNN models in IoT systems.

In addition to the data explosion, increased compute intensity, and expansive design spaces, recent IoT applications also place a growing demand for near-real-time processing.

Many applications, such as those in autonomous vehicles, industrial automation, and immersive experiences (e.g. augmented reality), require instant insights for timely decision-making. This near-real-time demand enhances the challenges associated with transmitting vast amounts of data to centralized cloud servers, as it introduces latency that can impact the effectiveness of these applications. The demand on the IoT network and the unreliable prolonged latency may not consistently meet the latency requirements for IoT applications [5].

1.3 Hardware Acceleration of DNN applications on the Edge

Rrunning compute-intensive DNN applications, even with reduced models, on a general-purpose edge with a multi-core CPU may still fail to meet performance requirements. To increase the computation capability and accommodate the real-time demand of DNN applications, modern edges (also, clouds in data centers) are often extended to incorporate accelerators, such as Field Programmable Gate Arrays (FPGAs) [47, 48, 158], Graphics Processing Units (GPUs) [150, 145], and Application-Specific Integrated Circuits (ASICs) [25], to increase the computation capability of resource-constrained edge nodes and improve the processing power aligned with the multi-cores [144]. Among these accelerators, ASICs are designed to perform specific logic functions and can be optimized for that function. This makes them faster and more efficient. However, they cannot perform any other task. This makes them less versatile than GPUs and FPGAs. GPUs employ a Single-Instruction, Multiple-Data (SIMD) architecture, offering high performance for highly parallel tasks and floating-point operations. However, due to the nature of SIMD architecture, GPUs lack heterogeneity in processing tasks, as all processing elements execute the same instruction on different data.

This limitation prevents fine-tuning for individual concurrent tasks and may result in high power consumption. On the other hand, FPGAs provide high customization, allowing for the selection of hardware parallelism and pipelining to achieve lower latency and better energy efficiency. Despite that deploying applications on FPGAs demands a deeper understanding of the hardware design process and requires expertise in FPGA programming, there have been works and solutions to facilitate the application deployment. For instance, the advancement of high-level synthesis (HLS) tools and development toolkits, like Xilinx Vitis HLS [2], Intel OneAPI [42], LegUp [7], has been eased the development of FPGA accelerators.

As Microsoft deploys FPGAs in its Bing search engine and Azure cloud [102] and Intel's significant acquisition of Altera for \$16.7 billion [100], the integration of FPGAs into edges and datacenters is recognized as a highly promising strategy for ensuring ongoing growth in future computing infrastructure. The flexibility of the hardware architecture enables the customization and reconfiguration of all on-chip logic blocks, which allows the adaption of hardware to diverse workloads.

In most IoT systems, tasks are heterogeneous and come from a variety of data sources, which results in high diversity but low latency requirement cases. The integration of FPGAs also facilitates customized computation to meet the specific needs of diverse applications. This is especially important given the escalating demand for computing power and the diverse requirements of data processing within the limited-resource edge nodes. Note that the resources on the edge node could be still limited compared to cloud nodes in data centers. Besides, the edge computing workloads are prone to be unbalanced and dynamically evolving [142, 41, 78] where tasks exhibit variability and may require customized acceleration. FPGAs offer a distinctive advantage through the ability to dynamically reconfigure individual accelerators and the entire FPGA fabrics in real-time where, in contrast to the relatively fixed architecture of GPUs, GPUs are known for their capabilities in highly parallel processing. The dynamic adaptability of FPGAs makes it a well-suited solution that not only accommodates the diverse computation tasks inherent in edge computing workloads but also enhances the overall efficiency and responsiveness of edge computing systems. In IoT systems, Computer Vision (CV) applications stand as a prime example of this dynamics. In IoT systems, devices equipped with cameras or sensors capture and offload visual data to the edge, enabling tasks such as object detection, facial recognition, and further object analysis. The workload dynamically fluctuates based on the environment, events, or the number of connected devices. Furthermore, the end device may occasionally preprocess data locally before transmission to the edge, which adds an additional layer of complexity and dynamism to the edge computing environment.

In this dissertation, we are considering the edge device, incorporating the multi-core and FPGA-based accelerators, that serve multiple end devices and share with task offloading services for hardware acceleration on computer vision applications using DNNs, as our target platform.

1.3.1 Deep Neural Network Models and Accelerators on FPGAs

In IoT systems, where the edge node is in closer proximity, DNN applications are executed efficiently. For instance, a study in [51] implements a digital edge computing layer over a Zynq board, serving as an edge, to achieve efficient image processing.

Modern DNNs involve millions of floating-point parameters and billions of floating-point operations for one inference on the image. However, when deploying the DNN models to hardware acceleration, the design and implementation of hardware for algorithms, as well as the utilization of lightweight algorithms, are equally crucial for the practical operation on resource-constrained edge devices.

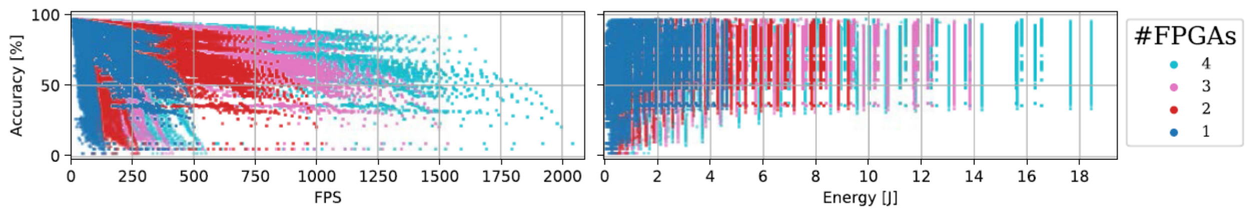

Figure 1.5: Design Space created when combining CNN model pruning and hardware parameters (Source from [59])

Since FPGAs provide high flexibility in customized hardware acceleration for neural networks, there have been efforts to design low-complexity DNN models that are dedicated to the custom accelerator architectures in various aspects, such as lower-precision arithmetic with quantization and small memory footprint [157, 110, 132, 97, 91], and model compression/pruning [31, 6, 59], where low latency and energy efficiency are achieved on FPGA-based accelerators. In Figure 1.5, AdaServ [59] shows the design space that explores the CNN model pruning and hardware parameters with a trade-off in accuracy versus throughput (i.e. FPS) or energy. For highly low-cost DNN applications, the bit quantization strategy can reduce computation to binarization, that is 1-bit in weight parameters as Binarized Neural Network (BNN). [15, 105, 157] has been shown to achieve high throughput (and minimized resource utilization) and still keep considerable accuracy. Besides, a variety of FPGA-based hardware architectures [132, 155, 90, 67, 23, 149, 20, 49] are also investigated to provide scalability toward different degrees of performance requirement, resource utilization, and the depth of the neural network. CHaiDNN [9] provides the DNN accelerator library that partitions DNN workloads into software and hardware layers and runs them in parallel to improve inference throughput.

Other than fine-tuning the customized architecture, for exploring huge design space, researchers have developed to systematically and efficiently explore the tradeoff between accuracy and efficiency for hardware-efficient DNN models [28, 66, 50, 143].

While research in the field is still rapidly growing, the current emphasis on DNN in FPGA acceleration primarily revolves around optimizing and deploying a single DNN accelerator on the FPGA. This focus, while valuable for the specific application itself, may not be sufficient in the context of edge computing scenarios where multiple tasks are currently requested, leading to competition for shared computing resources. The complexity arises with simultaneous task execution and resource contention, and, hence, developing an efficient FPGA-based Multi-Accelerator management framework for the dynamic demands becomes crucial for such FPGA-based edge computing environments.

1.4 System Software Support in Edge Computing

As discussed above, the edge computing technique not only alleviates the burden on the centralized cloud infrastructure and reduces the cost of data transmission to the cloud but also enhances real-time processing capabilities when it's crucial for applications requiring low latency. However, while edge computing offers significant advantages, it also comes with its own set of limitations and challenges. One key challenge is the limited processing power and storage capacity of edge devices compared to cloud servers. This constraint can impact the complexity and scale of computations that can be performed at the edge. Additionally, it poses another challenge in the increased complexity of managing layers of heterogeneous computing nodes, networks, and storage infrastructures collaboratively.

For the aforementioned challenges, to still meet the low latency requirements, researchers have investigated computation partitioning strategies among end devices, the edge nodes, and the data center cloud [150, 52, 11, 57, 146]. The entire neural network (NN) can be partitioned into parts or layers, with partial models deployed at various locations that could be shared vertically between cloud-edge-end or among edges. Intermediate results from one partial model are then forwarded to the subsequent higher layers of the overall model. Neurosurgeon [52] is one work that evaluates each design point based on end-to-end latency or mobile energy consumption. Then, it intelligently partitions DNN workloads layer-wise into segments and decides whether each segment should be deployed on the end, the edge, or the cloud while achieving the best latency and energy consumption of end devices.

In addition to task partitioning approaches in DNN applications, efficient task scheduling and resource allocation on the edge node itself are also critical for optimizing system performance and ensuring resource efficiency on the limited computational resources. Through task scheduling and resource management strategies, the edge device can prioritize DNN tasks based on their criticality, allocate resources dynamically, and adapt to changing workloads. For example, DjiNN [29] investigates the DNN-as-a-Service support for running multiple DNN services on a single-GPU server that achieves 100x throughput gain compared to the CPU-only baseline, and, in addition, explores the scalability on multiple GPUs. Clipper [16] provides a low-latency online prediction serving system that features caching, adaptive batching, and model abstraction for DNN applications on a single GPU server. In addition to GPU-based edge devices, our initial work [129] proposes the service infrastructures of obtaining DNN-as-a-Service and allows multiple applications to access accelerators simultaneously on an FPGA-based mobile edge prototype. AmorphOS [55] provides the system support for FPGA acceleration in different modes to improve performance and efficiency.

Other related works, like MCDNN [27], propose approximation-based DNN execution systems on GPU-accelerated devices to determine selected DNN models in compromised accuracy and performance/resource and initiate execution either remotely or locally.

In this dissertation, we are investigating the system software that provides efficient task scheduling and resource allocation on a single edge node, that obtains multi-core and FPGA-based accelerators. The edge node communicates and coordinates with multiple end devices, and the system software on the edge aims to maximize the resource efficiency of the shared DNN offloading services. The focus is on efficiently managing and processing the offloaded DNN applications from end devices with accelerators on the edge device itself, which doesn't partition tasks and rely on remote computing in the cloud. The system software not only dynamically allocates resources based on application demands and accelerator availability but also schedules tasks regarding factors like task dependencies and real-time workloads.

1.4.1 Multi-tenancy Support on the FPGA-based Multi-Accelerator Edge

Deploying DNNs onto FPGA hardware acceleration is proven to be a promising approach to improve the performance of DNN inference on edge devices. Existing works have shown promising solutions for deploying DNNs onto FPGA hardware acceleration for enhancing computational efficiency in real-time applications [3]. These efforts primarily concentrate on deploying a single DNN accelerator on a dedicated FPGA device.

In edge computing, the computing resources are shared among multiple end devices, requesting all sorts of accelerators to deploy for acceleration. In software processes, time-multiplexing is a common technique for sharing CPU resources, where the CPU time is divided into small time intervals known as time slices or quantum. Each user or process is allocated a time slice during which it can execute its tasks, and the operating system rapidly switches contexts between processes. However, applying a similar approach to share FPGA resources in a temporal manner, such as switching FPGA accelerators between applications, incurs overheads in setting up the FPGA fabrics for subsequent hardware acceleration. Consequently, the benefits of hardware acceleration are compromised by repetitive overhead in changing accelerators. Similarly, in hardware accelerators, sharing FPGA resources on a First-Come, First-Serve (FCFS) basis for task scheduling may involve significant overheads in switching FPGA accelerators between applications as well. In addition to sharing resources in a temporal aspect, it's also common to co-locate different tenants' workloads on the same server (e.g. hyper-threading in multi-core CPUs) to increase utilization. However, sharing an FPGA is more complex because applications will allocate physical spaces for accelerators on the chip, tenants can have diverse requirements, and the demand can also fluctuate over time. As a result, the FPGA edge can lead to contention and be fragmented in FPGA resources, impacting overall performance, when there are concurrent executions of multiple accelerators and diverse tasks on a shared FPGA edge. And the incurred transition overhead for the re-configuration and re-allocation of accelerators increases dramatically. Some works [70, 111] focus on directly reducing the overhead in reconfiguration operations to reduce the transition overheads of reconfiguring and re-allocating accelerators.

To tackle the contention in FPGA resources and support multi-tenant DNN acceleration, related works investigate time-multiplexing the DNN accelerator across multiple tasks for temporal multi-tenancy supports. István et al. [45] apply a single-pipeline design principle and state-machine-based logic to a multi-tenant approach for sharing an FPGA in the cloud among tenants. However, they are limited to the same type of application with different workloads and quality of service requirements. PREMA [14] introduces a preemptive execution scheduling algorithm for DNNs on a monolithic accelerator, which employs time-sharing techniques to facilitate multi-tenancy. AI-MT [4] focuses on re-structuring the layer's operations orchestrated with the proposed architecture that enhances the resource efficiency of the accelerator to achieve multi-tenancy. Other related works look into spatial multi-tenancy support where compute resources or memory resources are spatially partitioned across applications, e.g. Planaria [24], GAMMA [53], Mbongue, et al [82], etc.

Existing works in this domain tend to concentrate on either temporal or spatial multi-tenancy support, presenting a limitation in achieving comprehensive global optimization.

In this dissertation, our targeted edge computing system is on the FPGA edge that has multi-core CPUs and multiple FPGA-based accelerators. During runtime, the FPGA edge system can reconfigure individual accelerators independently and re-partition the entire FPGA resource into various numbers and sizes of DNN accelerators accordingly. The focus is on scheduling tasks, which allows out-of-order execution, aligned with accelerator allocation from a library of DNN accelerators where the tasks are in diverse workloads and various demands. (See Figure 1.6)

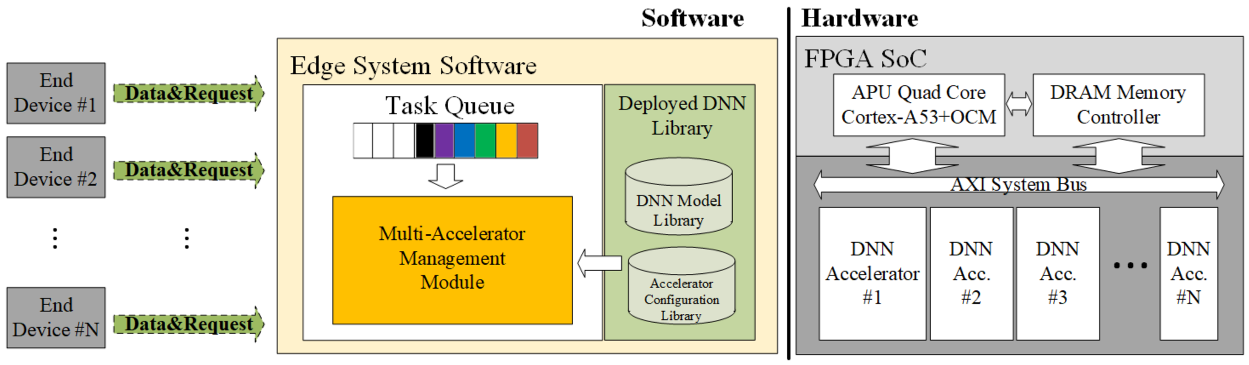

Figure 1.6: The Theme of This Dissertation

1.5 Overview and Contributions of this dissertation

As discussed, multi-tenancy support on FPGAs is an emerging topic that has captured researchers' attention. This approach focuses on sharing FPGA resources among multiple applications simultaneously. From the spatial multi-tenancy perspective, partitioning FPGA resources into multiple accelerator slots allows various applications to concurrently utilize FPGAs. These efforts improve the resource efficiency of FPGA and the system throughput to process multiple tasks to some extent. However, they encounter limitations in effectively sharing the same accelerators and managing various accelerator re-allocations among multiple applications or tenants. On the other hand, the temporal multi-tenancy support requires the capability of partitioning tasks into pipelined workloads orchestrated with the customized hardware architecture, and tasks are preemptive and multiplexing according to the time slice. In addition, existing multi-tenant platforms bind workloads to specific accelerators or sub-arrays [24, 45, 65], which lacks the flexibility in exploiting the regularity and dynamically allocating accelerators for diverse application requirements. This leads to high transition overheads when migrating workloads between applications during accelerator resource reallocation. Existing FPGA-based Multi-Accelerator system software lacks joint consideration and optimization in both spatial and temporal multi-tenancy support on FPGAs for dynamic workloads.

Moreover, current DNN serving systems process tasks from user-specified DNN models within pre-allocated computing resources, which result in either unnecessary constraints or wasted computing power. And the lack of dynamically sharing computing resources can even lower resource utilization and efficiency. There are works [10, 58, 101] formulate the mapping search as an optimization problem that models applications into a graph of dependent tasks for an FPGA-based multi-accelerator system. The accelerator allocation is not determined by the task itself from the user, which becomes flexible to map in compatible accelerators from the system perspective; however, they are not considering the share of accelerators between tasks, which could lead to unbalanced utilization of accelerators, and lack the consideration for the migration of workloads.

In edge computing, the edge computing workloads are diverse and vary over time. While processing various DNN workloads on the edge, the performance variability due to the design of DNN models and contention for the limited shared computing resources poses significant challenges to the edge system for efficiently scheduling and allocating resources on application tasks. In the absence of proper management, the system could turn to sub-optimal resource partitioning and utilization, and increase latency for completing applications. Thus, edge systems require a flexible and efficient mechanism to dynamically partition shared resources and schedule tasks based on diverse application requirements and available resources. However, in IoT systems, e.g. smart surveillance systems, data are often sampled periodically to monitor environments in a certain rate. These types of input become more predictable and can have some patterns for us to exploit and enhance the system support. while there are some patterns that can be exploited, there is always some uncertainty and unexpected load that has to be considered, such as the event-based data or noises under network congestion. Therefore, it's worth investigating approaches to exploit input workload characteristics to help the system in task scheduling and accelerator allocation. And, so far, there have been quite some efforts studied in resource management on general-purpose edge. However, they are not designed with compute resource scarcity or custom acceleration in mind. In addition, due to the lack of consideration for the heterogeneity and restriction in allocating computing resources, they are not suited for multi-accelerator FPGA edge systems. For efforts on the FPGA edge, most are online heuristic methods that find approximately optimal decisions and fail to exploit the input workload characteristics.

The focus of this dissertation is on sharing FPGA-based accelerators and dynamically allocating FPGA resources among user applications on an FPGA-based Multi-Accelerator Edge. It is important to note that the system support is not restricted to FPGA accelerators. The versatility of the proposed approach lies in its capability to model various accelerator characteristics, e.g. CPUs and GPUs, ensuring efficient resource allocation and sharing across diverse computing platforms.

Our target application is computer vision applications in IoT systems that utilize DNN models. The FPGA edge serves various concurrent DNN applications requesting hardware acceleration from diverse end devices. All devices send their tasks for acceleration simultaneously, forming queues of requests that require real-time responsiveness. Our application models are heterogeneous and do not bind the specific physical accelerator to the edge. Due to the limited number of resources and time-consuming process of accelerator re-allocation, sharing FPGA accelerators among different applications is crucial. Also, scheduling multi-tenant executions and flexibly orchestrating available accelerators based on applications' requirements becomes significantly important since resources must be dynamically reallocated to meet the distinct demands of concurrent workloads. Therefore, we need system software support on the FPGA edge that exploits input workload characteristics and jointly addresses both task scheduling and accelerator allocation for the concurrent DNN execution. The system software aims to enhance accelerator sharing and optimize overall FPGA resource utilization to accommodate diverse input workloads originating from multiple end devices.

Given those challenges, in Chapter 2, I first propose the DNN accelerator sharing service on the edge system. The framework uses the offline phase to determine the optimal policy on task scheduling and each task's corresponding allocated accelerator (referred to as $SharedDNN/PlanAhead$). At runtime, our results show an overall 2.20x performance gain at best and utilization improvement by reducing up to 27% of DNN library usage while staying within the requests' requirements and resource constraints. The DNN accelerator sharing service, $SharedDNN/PlanAhead$, shows the benefit of temporal and spatial sharing for the potential of optimization.

When the deployment environment is not sufficiently well-controlled, noises and uncertainties can happen due to network delays and system workload congestion. Therefore, we can't assume a fixed pattern since an exact task order and arrival time can't be forced. However, for the computer vision applications in IoT systems, the data are consistently sampled and tasks are sent periodically from end devices. When there are multiple end devices concurrently sending tasks to the edge, there exist patterns of applications, formed in the task queue on the Edge. In this case, I investigate data-driven approaches that exploit input workload characteristics to further enhance the system. Therefore, in Chapter 3, I present a two-phase systematic approach that not only exploits the characteristics of applications in patterns and balances the multi-tenant execution but also employs a mixed offline/online multi-queue scheduling method to optimize responsiveness by reducing response time and minimizing task drops for consistent IoT DNN workloads. We proposed a clustering-based pattern extraction approach that extracts characteristics for consistent input workloads and allows the variants in workloads as well. For the online phase, we propose an adaptive online scheduler to accommodate the network variation and mitigate the deviation from our planned schedule from the offline phase. With this two-phase approach, the edge system can significantly improve the responsiveness and robustness in serving multiple end devices where the task drops are improved to zero.

Other than the stable monitoring system, there are more challenges. Considering IoT applications mostly operate in an event-driven paradigm, characterized by varying task rates changing over time, the input workload experiences dynamic and uncertain changes. Then, the patterns of tasks become unpredictable and hard to extract and generalize in sequences or clusters. To effectively capture these dynamics and find an adaptive scheduling policy, in Chapter 4, I present a learning-based multi-accelerator management framework. This framework asynchronously learns the scheduling and allocation policy, dynamically partitioning shared resources to optimize system performance through continuous interaction with the edge. The experimental results show that our proposed framework outperforms related learning-based approaches and heuristic scheduling schemes. It surpasses in terms of average throughput and task drop rates, showcasing its efficiency in adapting to the dynamic workload changes inherent in IoT applications.

Chapter 2

Dynamic Sharing in Multi-accelerators of Neural Networks on an FPGA Edge Device

2.1 Introduction

Edge computing provides efficient data processing by minimizing the amount of long-distance communication between IoT devices and the cloud server. Therefore, edge nodes can reduce bandwidth occupation, latency, and energy consumption [147]. Hence, edge devices have become a hub for real-time DNN applications and many application-specialized DNN accelerators have been developed on the edge. In this context, FPGAs are attractive accelerators due to their low power consumption and reconfigurability, which makes them shareable as edge accelerators. Since the edge is a shared commodity, there are various applications requesting access to the edge accelerators.

Figure 2.1 shows an overview of such an edge device: The FPGA edge can deploy different

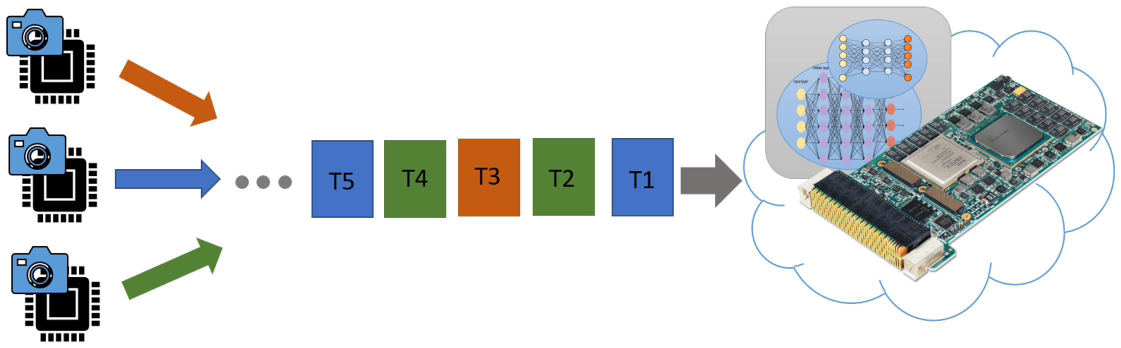

Figure 2.1: Overview: FPGA Edge Serving Multiple IoT Applications

DNN accelerators and receive tasks from different end devices. All devices send their requests for acceleration simultaneously and form a queue of requests demanding real-time responsiveness. Given limited computing resources at the edge, sophisticated strategies for edge devices are required to prioritize the tasks and to allocate the best acceleration options that meet the task requirements, and yet, maximize the edge responsiveness.

We develop a DNN accelerator sharing system on the FPGA edge device such that multiple users can access various DNN accelerators during runtime. We present an optimal offline policy for accelerator allocation to each request. In this policy, the system software determines which accelerator to use for each task given the user DNN model parameters. In addition, we assume that requests to access accelerators are received at the edge with a periodic pattern that knows a priori (using simple less-congested network models or real-time wireless networks). We present a graph representation G to represent the tasks and accelerators and show that the optimal policy is obtained by solving the min-cost k-disjoint paths to this graph in an offline pre-deployment phase. To solve it, we present a mixed-integer linear programming formulation of this problem. During deployment, while each end device periodically sends requests to the edge , the runtime task manager maps the tasks to the accelerators following the proposed offline policy (referred to as $SharedDNN/PlanAhead$). Our proposed framework is applied on a set of end devices sending requests to access FPGA-based DNN accelerators for vision applications such as object detection and classifications. The requests are all sent wirelessly at fixed periodic rates and each request includes user DNN model parameters. The framework develops an acceleration service both at offline pre-deployment and runtime deployment phases on the FPGA edge. Hence, using the offline phase of our framework, we determine the optimal policy on accelerator allocation and execution of the tasks on those accelerators. At runtime, our results show that at best the overall speedup can achieve 2.20x performance gain and utilization improvement by reducing up to 27% of DNN library usage.

In this chapter, we are targeting an edge equipped with a library of FPGA-based binarized DNN accelerators. Applications send their acceleration requests and our proposed DNN acceleration service efficiently determines which accelerators and slots to select for each request and in which order the requests are responded to. To the best of our knowledge, our proposed DNN accelerator sharing service is the first effort in edge-based DNN-accelerator sharing service where we exploit FPGA reconfigurability and consider the trade-off between performance and accuracy of DNN models to share the accelerators on an FPGA edge to respond to a queue of incoming requests to access the accelerators from multiple applications.

2.2 Edge FPGA-based DNN Accelerator Sharing System

Many IoT applications monitor the environment by regularly sensing and processing the data in real-time. For example, in vision applications, various end devices capture the images from the environment and apply vision algorithms such as object detection/classifications or object tracking. While multiple end devices send their data to the edge to run vision algorithms at a fixed periodic rate, the edge receives a stream of requests from multiple end devices to access the edge computational resources at the same time. Given that each end device submits their request at a fixed rate, under stable and simple wireless communication, we observed periodic patterns in the arrival of the requests at the edge. (Figure 2.2, for example, if a DNN accelerator is requested by an end device for object classification at 3 fps and another device requests access to a DNN accelerator at 2 fps, and another one at 5 fps, we observe patterns in the arrival of the tasks at the edge). In this chapter, given a simple network model, we assume this pattern is extracted empirically. The pattern of requests to access FPGA accelerators is then used at the pre-deployment phase to dynamically allocate accelerators to each request and re-order the execution, if necessary, to enhance the edge responsiveness to the queue of requests to access edge FPGA accelerators. This work develops an acceleration service both at the pre-deployment and deployment phases of IoT systems to access multiple DNN accelerators on the FPGA edge.

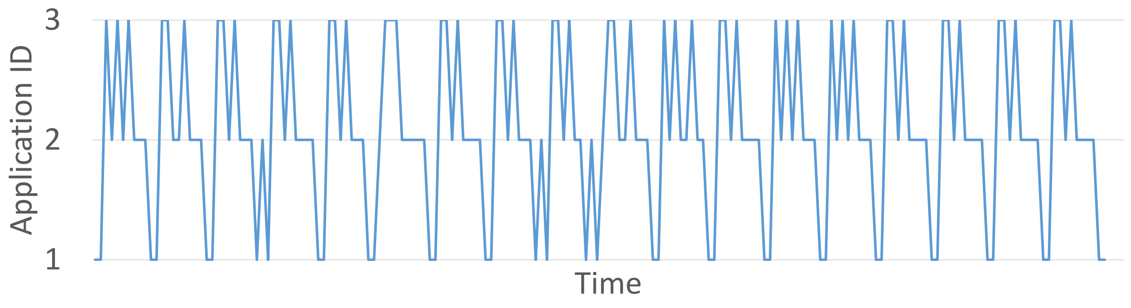

Figure 2.2: Example Pattern of Receiving Tasks at The Edge from Applications #1-3. (#1 at 2 fps, #2 at 5 fps, and #3 at 3 fps)

DNN algorithms with significantly reduced computational loads have shown to be promising solutions in the real-time response of vision algorithms in IoT systems with acceptable accuracy [28]. As mentioned in related work, there are several state-of-the-art FPGA designs such as binarized DNN algorithms on FPGA, making them viable candidates as accelerators on FPGA edge devices. As resource-efficient accelerators, multiple DNN accelerators can fit on an FPGA device, hence, enabling parallelism. While reconfiguring an FPGA-based DNN accelerator requires only a few milliseconds, it allows the system to switch accelerators between applications at runtime.

Many users request their specific FPGA-based DNN accelerator to perform inference for vision applications, e.g. object detection, based on various factors, such as performance, inference accuracy, target dataset, and DNN model of the accelerators. To perform DNN acceleration, such DNN models have to be trained in advance, which determine inference accuracy and the target dataset for inference. The DNN model's network topology will influence compute complexity as well as inference speed. The same network topology can also apply to other DNN models and train for other target datasets, which results in various inference accuracies. Then, the DNN topology can be implemented in different architectures for hardware acceleration, which also affects the performance. Hence, different DNN models' hardware acceleration may share the same DNN accelerator but acquire different DNN model parameters. However, this can still lead to requests to access a large diverse set of accelerators by various applications and users at the edge while FPGA resources are shared to access those accelerators. In order to have more effective utilization of FPGA resources, we propose the system software to allocate the accelerators to each task instead of users. Each user request, instead, contains the target *dataset* and *accuracy* requirement along with the DNN model, not a specific DNN accelerator. While the acceleration request is received on the edge device, it is sent to the common task queue. Since the patterns of incoming requests are known a priori, the expected requests in the queue along with their DNN model parameters are deployed during the offline phase to find a matching accelerator for each task while effectively enhancing responsiveness to all requests. In order to achieve this, it aims to avoid too many FPGA acceleration reconfiguration and accelerator swapping.

In Figure 2.3, the edge sharing system is applied to the FPGA edge, containing embedded processors and multiple accelerator slots. As mentioned earlier, we assume that input task arrivals in a finite time horizon T come with a priori known expected pattern. Hence, the

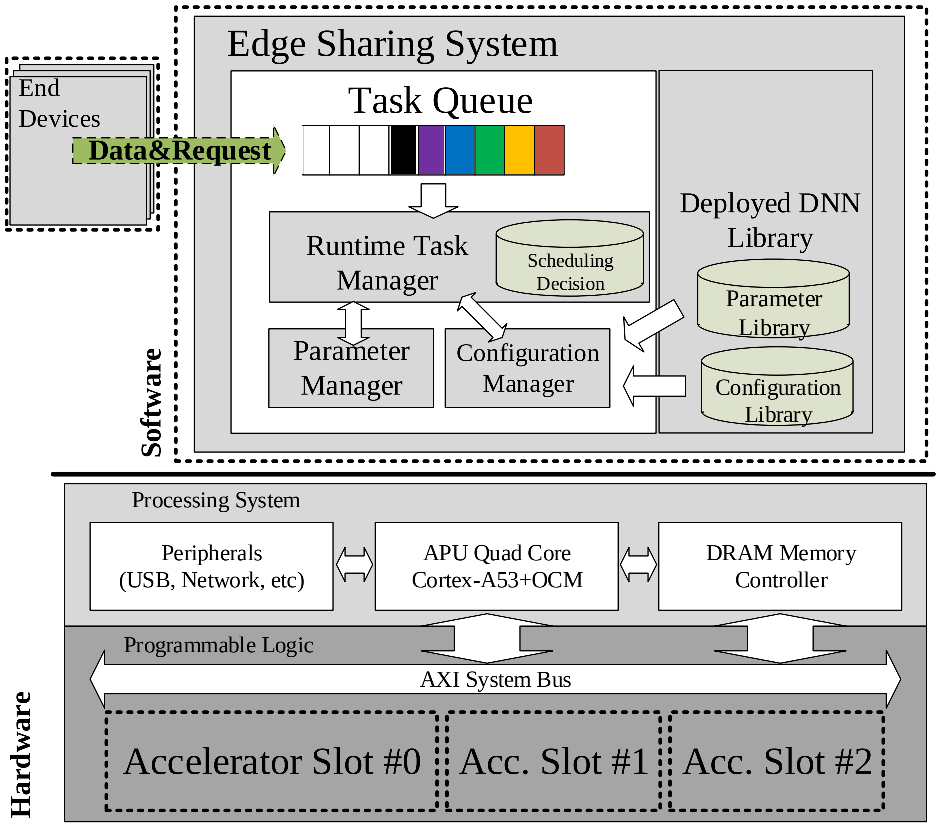

Figure 2.3: Overview: Edge FPGA-based DNN Accelerator Sharing System

offline pre-deployment phase will determine the expected order to run the incoming requests to access the accelerators. In addition, it determines which accelerator and which accelerator slot is used for each task request. Given that each request from an application is considered independent from the next requests, the consecutive requests from the same user or end device to access DNN accelerators may be mapped to different accelerators and slots. Also, the requests can respond totally out of order of their arrivals in the queue. That is why we call this phase "Static Out-of-order DNN-accelerator Sharing systems" or simply referred to as $SharedDNN/PlanAhead$.

During deployment, while each end device periodically sends requests to the edge to access FPGA DNN accelerators, the runtime task manager maps the tasks to the accelerators following the offline policy determined by the $SharedDNN/PlanAhead$ phase. The configuration bitstreams of each utilized DNN model are stored in a configuration library, and the configuration manager reconfigures the individual accelerator slots in the FPGA accordingly, driven by the runtime task manager. The weight/threshold parameters of each DNN model are stored in a parameter library, and the parameter manager loads to the corresponding accelerator slots, driven by the runtime task manager as well.

2.3 SharedDNN/PlanAhead: Offline Out-of-Order DNN-Accelerator Sharing Policy

The problem of Shared-DNN static scheduling is stated as follows: We are given a queue of requests to access DNN accelerators on the edge FPGA device from multiple end users or IoT end devices. Each request includes a set of requirements of DNN models such as *accuracy*, *datasets*, etc. The FPGA edge has a rich collection of binarized DNN configurations to select from, to implement, and to run the accelerator in response to each request while meeting the user DNN model requirements. We assume that multiple accelerators can be potentially placed and implemented on the FPGA and hence, multiple accelerators can be deployed in parallel. If the accelerator requested by the user does not exist on the FPGA accelerator slots, the FPGA invokes a runtime partial reconfiguration to allocate a new accelerator. Given the queue of requests in a given order, the problem is how to allocate DNN accelerators for each request and in what order to execute the accelerators such that the total execution time of all accelerators corresponding to all the requests are minimized.

Assume there are $\underline{k}$ accelerator slots on FPGA. Each slot can be reconfigured to implement a subset of DNN accelerators from the libraries if the resources in the slot are sufficient for the implementation of those accelerators (one at a time). We are also given a queue of tasks (or requests) $T = T_1, T_2, ..., T_n$ and would like to map each task T_i to a DNN accelerator Acc_j such that the total elapsed time to finish execution of all requested accelerations is minimized. The tasks are allowed to be re-ordered (or partially) and do not necessarily follow their temporal ordering in the ready queue. We first present a graph representation and map the problem to find the min-cost k disjoint paths problem in this graph. We present a mixed-integer linear programming (MILP) formulation of the problem to be solved by mathematical programming solvers such as IBM CPLEX® Optimizer [39].

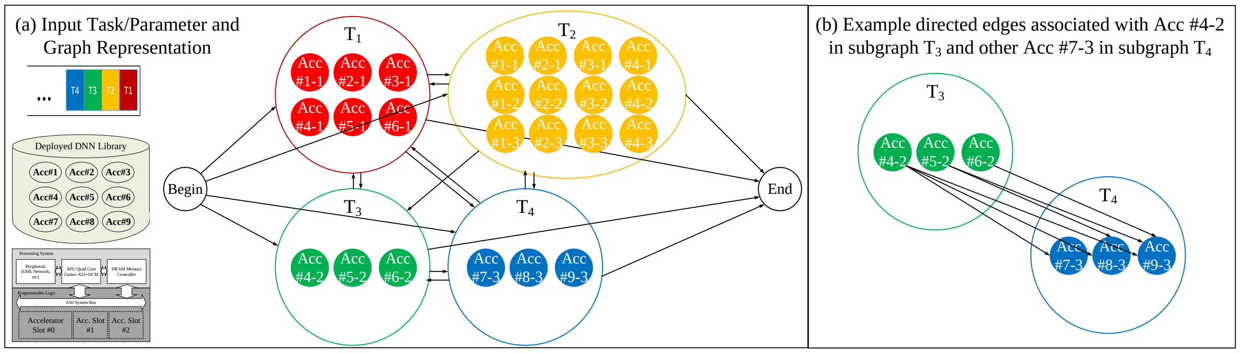

Figure 2.4: Graph Representation of Tasks And Accelerators

2.3.1 Graph Representation of Tasks and Accelerators

We introduce graph G to represent all the tasks in the request queue within time interval T. This graph requires not only to reflect the mapping and allocation of each task to an accelerator but also to provide an ordering of execution of the tasks on the designated accelerators. Graph $G = (V, E)$ is composed of a set of nodes V and an edge set E. Each node represents a task being mapped to an accelerator. Given each user's requested DNN model parameters such as dataset, accuracy, etc., various DNN accelerators are qualified to execute this task. Hence, in graph G, we introduce subgraph T_i associated with each task i in the queue. The nodes in such a subgraph represent all the potential eligible accelerators to execute task T_i.

Figure 2.4(a) shows an example of graph G for a given set of tasks $T = \{T_1, T_2, T_3, T_4\}$, DNN library, and the hardware configuration setting (see Section 2.4.1). Let's assume a node labeled as "Acc. #i-j" represents a DNN accelerator with ID i using dataset j. For example, in Figure 2.4(a), accelerator #1 using dataset #1, named Acc. #1-1, which satisfies the requirements of task T1, will be included in subgraph T_1. Since accelerator #1 using dataset #1 can also meet task T2's requirement, it's also included in the subgraph T_2. Hence, Graph G is composed of N such subgraphs (T_i) representing all the tasks in the queue.

In this graph, directed edges are introduced between the nodes across subgraphs T_i, $i = 1, 2..., N$. The edges between the subgraphs are defined as follows: A directed edge between any node from one subgraph to another represents the ordering of execution of the two tasks in one accelerator slot. For example, an edge from the node labeled as "Acc. #1-1" (from subgraph T_1) to a node labeled as "Acc. #1-1" (from subgraph T_2) means that task 2 executes on a DNN accelerator with ID 1 and with dataset 1 after the execution of task 1 on the same accelerator with the same dataset in the same accelerator slot. Given that both nodes have the same accelerator and dataset, there is no need for partial reconfiguration and parameter reload. Besides, there might be no directed edges between any two nodes from one subgraph to another if the corresponding accelerator slot can't fit either one of the accelerators, e.g., in Figure 2.4(b), no directed edges between Acc. #5-2 in subgraph T_3 and Acc. #7-3 in subgraph T_4.

The cost on the directed edge reflects the transition delay overhead between the two task execution, including the partial reconfiguration overhead (to replace with a new accelerator) and/or time overhead to re-load DNN parameters. For example, in Figure 2.4(b), the cost on the edge from Acc. #4-2 in subgraph T_3 to Acc. #7-3 in subgraph T_4 includes both the partial reconfiguration overhead and the time overhead since the FPGA slot is reconfigured to implement accelerator ID 7 after execution on accelerator ID 4 is finished. On the other hand, the transition overhead on the edge from Acc. #1-1 in subgraph T_1 to Acc. #1-2 in subgraph T_2 includes only the time overhead to reload the new parameters for DNN. Since an edge imposes the ordering between the two tasks, the subgraph associated with each task T_i is an independent set in the graph.

Directed path in graph G: A directed path in graph G that covers a subset of subgraphs represents the consecutive execution of those tasks on the accelerators corresponding to their nodes along the path. Covering a subgraph means to cover at least one node in each subgraph. The execution time for each task can be considered as the cost of the node and the reconfiguration time overhead is the cost of the edges along the paths. Hence, the total cost of the path is the total elapsed time to execute those tasks and the transition time overhead to reconfigure the accelerators when necessary. Each task is only executed on one accelerator and hence, exactly one node from each subgraph associated with each task has to belong to a directed path. This is referred to as uniqueness in path selection.

Since the directed path refers to the sequential execution of the tasks along the path, it refers to the ordering of runtime accelerator implementation and task execution in only one FPGA accelerator slot. If there is only one slot (one accelerator at a time), then the problem of allocating each task to an accelerator is to find a directed path that covers exactly one node from each subgraph.

When there is more than one slot for accelerators on FPGAs and more than one accelerator can run in parallel, the tasks do not need to all belong to one directed path. In this case, each directed path represents one slot. Each directed path represents the sequence of tasks execution on the accelerators placed in that slot. Hence, for 3 slots on FPGAs, we need to find 3 directed paths in graph G such that all the subgraphs associated with each task is covered. In addition, we have to make sure only one node from each subgraph is selected.

Finding k disjoints path in this graph solves the problem and allocates an accelerator to each task. In Figure 2.4(a), the FPGA edge contains two accelerator slots (L and M). Therefore, we need to find two disjoint paths and cover all the subgraphs to provide a feasible ordering for all the tasks. If we find two disjoint paths with minimum total cost (minimizing the maximum cost of the paths), we have identified the optimal solution.

Since the directed paths do not keep the initial ordering of the tasks in the queue, we call it out-of-order responsiveness to user requests to access accelerators on an FPGA edge. If the partial temporal ordering of the tasks in the queue needs to be preserved, we remove the direct edges between the nodes of those subgraphs that result in violating the ordering imposed by the user or application policy.

In the next section, we present a mathematical problem formulation of this problem using mixed-integer linear programming.

2.3.2 Mixed Integer Linear Programming Formulation

We present a mixed integer linear programming (MILP) formulation of the problem. We have a set of N tasks, representing N=$\{1, 2, ..., N\}$ and a set of accelerator slots K=$\{1, 2, ..., K\}$.

Table 2.1: Definitions of Input Tasks/Parameters and Decision Variables for MILP Formulation

Input Task/Parameter	
ds_i	Target dataset for inference of task i
ac_i	Accuracy requirement for inference of task i
ta_i, tp_i	Arrival time and relative period of task i
$temp_{i,j}$	Indicate if task i and j are sensitive to the temporal order
dnn_i	Selected accelerator from DNN library satisfying the requirement of task i
$dnn2ds_{u,p}$	Inference accuracy on DNN u using dataset p
$dnn2slot_{u,k}$	Indicate if DNN u can fit in Slot k
$dpd(ds_i, p)$	Indicate if target dataset of task i is dependant to dataset p
$opr_{i:u,jv,k}$	Reconfiguration overhead from DNN u running task i to DNN v running task j in Slot k
$orel_{i:p,j:q,k}$	Parameter reload overhead from task i using dataset p to task j using dataset q in Slot k
$te_{i:u,k}$	Execution time on DNN u running task i in Slot k
Decision Variable	
$cons_{i:u:p,j:v:q,k}$	Indicates if DNN u running task i using dataset p and DNN v running task j using dataset q in Slot k are executed in a **consecutive** order.
$imp_{i:u:p,k}$	Indicates if DNN u running task i using dataset p is **implemented** in Slot k
$tss_{i:u:p,k}$	Scheduled start time of DNN u running task i using dataset p in Slot k

Table 2.1 lists the input tasks, task parameters, and decision variables for the proposed MILP formulation.

1. **Uniqueness and Disjoint Path Constraint**: Each task is executed exactly once and allocated to an accelerator from the DNN library.

$$\sum_{j \in N} \sum_{v \in dnn_j} \sum_{q \in ds_j} cons_{i:u:p,j:v:q,k} = imp_{i:u:p,k}, \quad \forall i \in N, \forall u \in dnn_i, \forall p \in ds_i, \forall k \in K \tag{2.1}$$

$$\sum_{u \in dnn_i} \sum_{p \in ds_i} \sum_{k \in K} imp_{i:u:p,k} = 1, \forall i \in N \tag{2.2}$$

2. **Flow Constraint**: If on path k, there is an edge to the node of the DNN w running task i using dataset o, then there is an edge starting from this node to another node on the same path k.

$$\sum_{i \in N} \sum_{u \in dnn_i} \sum_{p \in ds_i} cons_{i:u:p,h:w:o,k} = \sum_{j \in N} \sum_{v \in dnn_j} \sum_{q \in ds_j} cons_{h:w:o,j:v:q,k}, \quad \forall h \in N, \forall w \in dnn_h, \forall o \in ds_h, \forall k \in K \tag{2.3}$$

3. **Timing Constraint**: When there are two consecutive nodes on the same path k, the second node will not start until the accomplishment of the first one. And each scheduled node will never start before its arrival.

$$cons_{i:u:p,j:v:q,k} == 1 \implies tss_{i:u:p,k} + opr_{i:u,j:v,k} + orel_{i:p,j:q,k} + te_{i:u,k} \leq tss_{j:v:q,k}, \quad \forall i,j \in N, \forall u,v \in dnn_{\{i,j\}}, \forall p,q \in ds_{\{i,j\}}, \forall k \in K \tag{2.4}$$

$$imp_{i:u:p,k} == 1 \implies ta_i \leq tss_{i:u:p,k}, \\ \forall i \in N, \forall u \in dnn_i, \forall p \in ds_i, \forall k \in K \tag{2.5}$$

4. **Accelerator Slot Constraint**: The FPGA fabric each accelerator slot can vary, which might not be able to fit large DNN accelerators. This constraint makes sure such accelerators will not be allocated to the slot.

$$dnn2slot(u,k) == 0 \implies imp_{i:u:p,k} = false, \\ \forall i \in N, \forall u \in dnn_i, \forall p \in ds_i, k \in K \tag{2.6}$$

5. **Dependent Dataset Constraint**: While some targeted dataset is related or a subset of other datasets due to the less need of classes for IoT devices. This constraint will make sure the migration of the target dataset to other dependent datasets can still meet the accuracy requirement.

$$dpd(ds_i, p) == 0 \ OR \ (dnn2ds_{u,p} < ac_i) \implies cons_{i:u:p,j:v:q,k} = 0, \\ \forall i,j \in N, \forall u,v \in dnn_{\{i,j\}}, \forall p,q \in ds_{\{i,j\}}, k \in K \tag{2.7}$$

6. **Temporal Ordering Constraint**: Some tasks are sensitive to temporal ordering, such as human actions for applications. Tasks with such constraints will be processed in chronological order.

$$temp_{i,j} == 1 \implies tss_{i:u:p:k} \leq tss_{j:v:q:k}, \\ \forall i,j \in N, \forall u,v \in dnn_{\{i,j\}}, \forall p,q \in ds_{\{i,j\}}, k \in K \tag{2.8}$$

7. **Objective**: To achieve maximum responsiveness, the objective function is to minimize the maximum elapsed time to the end node while all the tasks are scheduled on the given accelerator slot configuration. The formula is in the following: $min(max(tss_{end,,k}, \forall k \in K))$

2.4 Evaluation

2.4.1 Platform Setup

The edge FPGA-based DNN accelerator sharing system is deployed on a Xilinx Ultrascale+ MPSoC ZCU104 board. We partitioned the FPGA fabric into multiple various-size accelerator slots. Each accelerator slot can be occupied by different DNN accelerators. The bigger accelerator slots, obtaining more amount of FPGA fabric, can fit bigger and then more powerful DNN accelerators. Considering the resource utilization of our deployed DNN library, the FPGA edge is allowed to have one of the three configuration settings:

- One large and one medium accelerator slots (LM)
- One large and two small accelerator slots (LSS)
- One small and two medium accelerator slots (MMS)

The time to reconfigure individual L/M/S slots varies in 120/105/90 milliseconds related to the amount of FPGA fabric. Our edge-sharing system then provides multiple accelerators serving vision applications simultaneously and applies different DNN accelerators to each slot by downloading partial bitstream files as well as partial reconfiguration. The edge-sharing system can dynamically share the FPGA while the system is still operational among multiple tasks without interruption.

Based on our FPGA edge's computing resources and configuration settings, Table 2.2 shows nine accelerators in our DNN library before deployment for our experiments. The listed accelerators are modified versions of Binarized Neural Network accelerators, e.g. [132], with different optimizing factors so they can be accommodated properly given the size limitations of slots and also to meet the required performance, e.g. accelerator 1 to 3 are obtaining the same DNN topology but different hardware architectures, same as accelerator 4 to 6, and 7 to 9. Since Dataset1 (cifar10) and Dataset2 (cifar3) are dependent datasets, they can be trained with the same DNN topology and maintain decent accuracy. However, the target dataset in accelerator 7 to 9 is independent to the dataset in accelerator 1 to 6, so the DNN model for other datasets are not trained for those accelerators.

Table 2.2: DNN Library Information (Data1/Data2/Data3 are CIFAR10/CIFAR3/MNIST Datasets)

Accelerator	Perf. (fps)†	Data1	Data2	Data3	Slots
1	33	80.10	90.42	-	(L)
2	20	80.10	90.42	-	(L, M)
3	10	80.10	90.42	-	(L, M, S)
4	20	84.33	92.91	-	(L)
5	10	84.33	92.91	-	(L, M)
6	5	84.33	92.91	-	(L, M, S)
7	33	-	-	98.55	(L)
8	20	-	-	98.55	(L, M)
9	10	-	-	98.55	(L, M, S)
† the frame resolution is 640 $\times$ 480 pixels					

2.4.2 Workload and Application Characteristics

In our experiments, we assumed four end devices periodically sending requests for acceleration to the FPGA edge. Four end devices and the FPGA edge communicate through a dedicated wireless router access point. There are three different workloads: HIGH, MEDIUM, and LOW representing an average of 60, 46.6, and 30 fps in total for the four end devices respectively.

Each workload explores four application sets that individual applications are targeting different fps, datasets, and accuracy to leverage utilizing different DNN libraries. Table 2.3 shows different settings for applications that result in sharing DNN accelerators/parameters or not. Application set 1 allows sharing the DNN accelerator between Application #1, #2, and #3. The target dataset running on application #2 and #3 can also be migrated to the dependent dataset, Data1, within the accuracy requirement. Sharing the DNN accelerator happened in Set 2 as well, but not the share of DNN parameters; thus, the reload overhead was compromised. Set 3 and 4 both are in the tradeoff between sharing accelerators with less performance, or dedicated accelerators with transition overheads, including reconfiguration and reload overhead.

Table 2.3: Application Sets: sharing accelerators/parameters

	Application#1	Application#2	Application#3	Application#4
Set	(Data, Accuracy)	(Data, Accuracy)	(Data, Accuracy)	(Data, Accuracy)
1	(Data1, 80)	(Data2, 80)	(Data2, 80)	(Data3, 90)
2	(Data1, 80)	(Data2, 80)	**(Data2, 90)**	(Data3, 90)
3	(Data1, 80)	(Data2, 80)	**(Data2, 92)**	(Data3, 90)
4	**(Data1, 84)**	(Data2, 80)	(Data2, 80)	(Data3, 90)

We generate the synthetic benchmark in the offline phase of our framework to perform the scheduling decisions. While the edge-sharing system, Figure 2.3, is receiving acceleration requests from the end devices in a wireless network, there is network congestion, TCP/IP, or other indeterministic network policies, and uncertainty to the wireless communication. Therefore, the requests from applications are not received at the edge with a fixed expected pattern; hence we assume a 30% randomized deviation in the continuous task requests from the four applications using the synthetic benchmark. We evaluate the end-to-end performance on the FPGA edge and average the result of one hundred times randomly deviated input for each application set. Although the proposed $SharedDNN/PlanAhead$ policy has proceeded to a static scheduling decision from the given synthetic benchmark, our runtime task manager handles the deviation of continuous task requests in respect of the static scheduling decision.

2.4.3 Experimental Results

To evaluate the efficiency of our proposed framework and sharing policy, *SharedDNN/PlanAhead*, we measured its improvements in performance and utilization in comparison with two other methods: *FixedDNN/FCFS*, and *SharedDNN/FCFS*. Here are the descriptions of these three methods that we ran against each other:

1. *FixedDNN/FCFS* allocates one individual accelerator slot and selects the qualified DNN library only for the first acceleration request from each application. That is, tasks within each application will be fixed to the assigned accelerator and slot. Any task assigned to the same individual slot will be executed in order of their arrival, i.e. First-Come-First-Serve (FCFS).

2. *SharedDNN/FCFS* allocates one individual accelerator slot, that is currently in an idle state, and selects the qualified DNN library for each acceleration request in the task queue. As a result, tasks within each application might not be fixed to one accelerator and the slot. In the *SharedDNN/FCFS* policy, accelerators and slots are shared among tasks and applications. All tasks will be still executed in order of their arrival, i.e. FCFS.

3. *SharedDNN/PlanAhead* is our proposed offline-scheduling policy that not only allows sharing accelerators and slots among tasks but also prioritizes the task scheduling to achieve maximum responsiveness within the target dataset and accuracy requirement.

For simplicity, we use *Fixed*, *Shared*, and *PlanAhead* to represent *FixedDNN/FCFS*, *SharedDNN/FCFS*, and *SharedDNN/PlanAhead* policies.

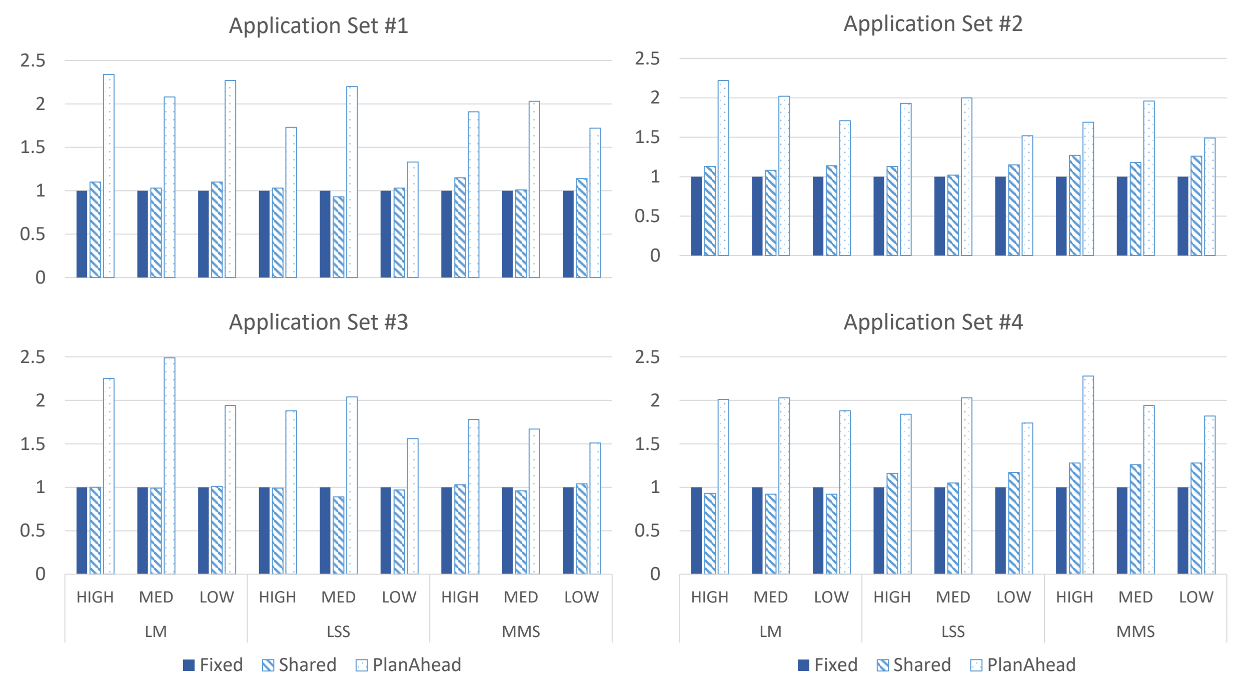

Figure 2.5: Speedup each Application Sets #1-4 with HIGH/MEDIUM/LOW Workloads (Baseline: $FixedDNN/FCFS$ policy)

We evaluate the above three scheduling policies with the exploration of HIGH, MEDIUM, and LOW workload, and three configuration settings, {LM, LSS, MMS}, together with the deployed DNN library in Table 2.2. We assume four end devices continuously sending task requests to the FPGA edge. Although the proposed $PlanAhead$ policy performs the scheduling decision from the synthetic benchmark, the runtime task manager refers to the static scheduling and is flexible to accommodate task requests considering the network congestion. Our runtime task manager halts the individual accelerator slot when the expected task from static scheduling is delayed due to the deviation of continuous task requests, which is introduced by the network congestion.

Figure 2.5 shows the average speedup of application sets #1-4. In application set #1, the application characteristic implies sharing DNN accelerators and parameters between applications #1-#3 due to their low accuracy requirements and dependent datasets. As a result, the $Shared$ policy obtains little performance gain in application set #1 compared to the *Fixed* policy, which has applications fixed to the assigned accelerators. However, our proposed *PlanAhead* policy can still acquire up to 2.35x speedup via well-distributing tasks among accelerator slots on application set #1. Also, application sets #3 and #4 involve the trade-off between sharing large accelerators and allocating dedicated accelerators that our proposed *PlanAhead* policy considers the correlation of sharing accelerators and the application characteristic, as a result, achieves up to 2.49x speedup.

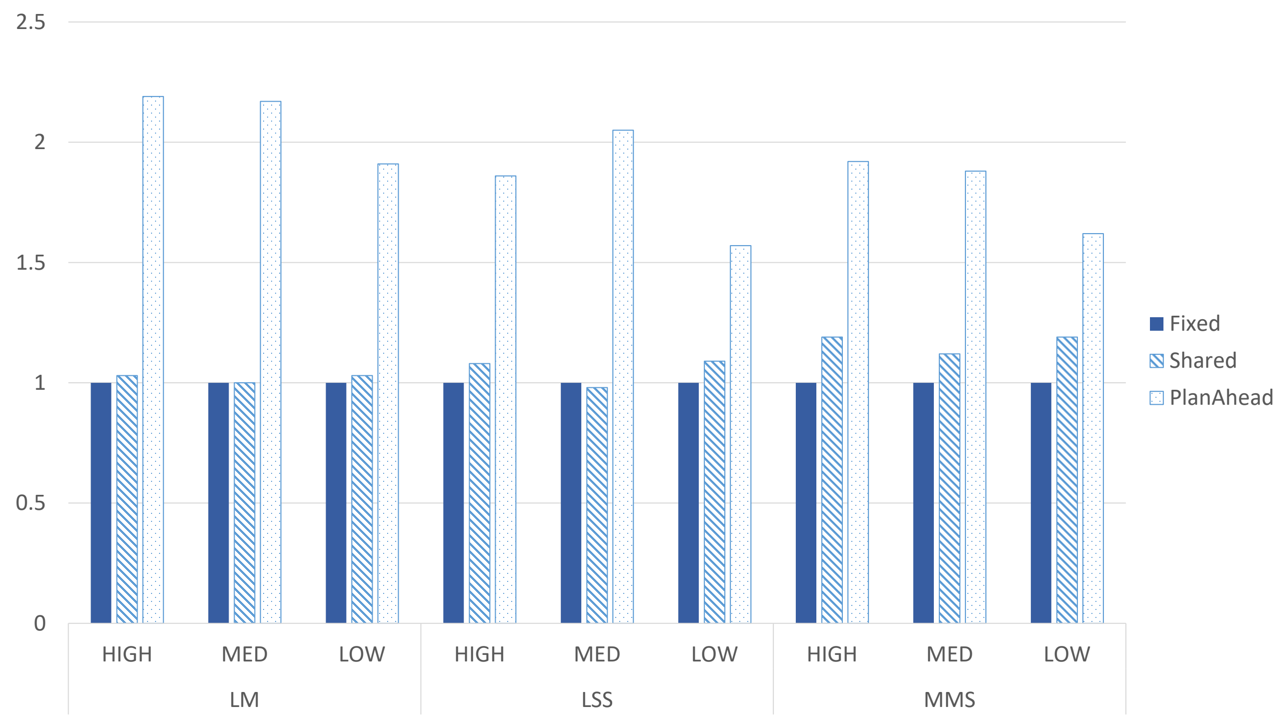

Figure 2.6: Overall Speedup with HIGH/MEDIUM/LOW Workloads (Baseline: *FixedDNN/FCFS* policy)

The overall average speedup on each workload and configuration setting is shown in Figure 2.6. In the LM configuration setting, the *Shared* policy marginally improves the performance compared to the *Fixed* policy since the two slots are already highly shared with multiple individual tasks. On the other hand, in the LSS, MMS configuration settings, the *Shared* policy obtains slightly better speedup due to the increase of individual slots for sharing. But, in all workloads and configuration settings, our proposed *PlanAhead* policy outperforms the *Fixed* and *Shared* policy.

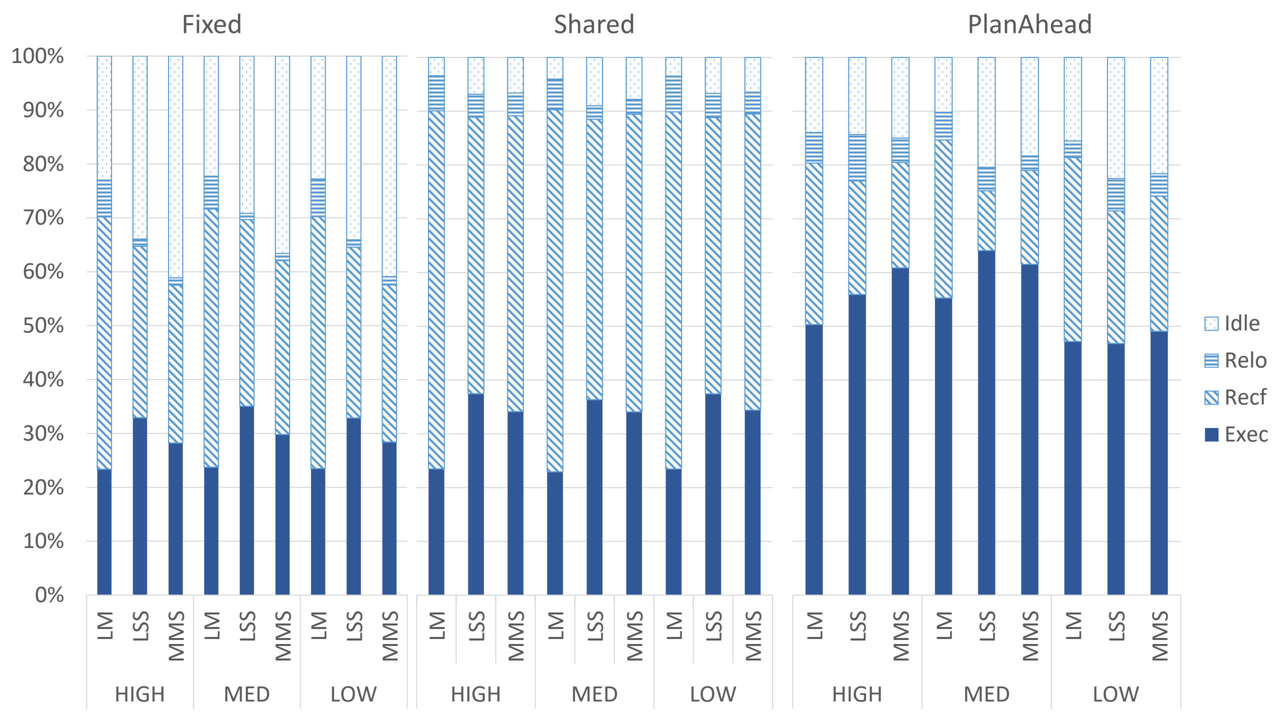

Figure 2.7: Runtime Distribution: Fixed/Shared/PlanAhead

The benefit of sharing in *Shared* is amortized due to the recurring transition overhead, including reconfiguration and reload overhead. While, in *Fixed*, the applications and slots are one-to-one paired, the edge system is not compromised to the recurring reconfiguration/reload overhead. Figure 2.7 demonstrates the time distribution across idle/reload/reconfigure/execution. The *Fixed* policy has shown less transition overhead; however, it failed to balance the load among each application for various sizes of slots, which results in more idle states of slots.

In contrast, the *Shared* policy shares the accelerator slots to reduce the idle state of slots. It, instead, suffers from spending the most time on the transition overhead, updating the accelerators and parameters, which massively penalizes the use of sharing accelerators. In terms of the conflict between sharing and transition overhead, our proposed *PlanAhead* policy considers sharing accelerators, slots, and characteristics of acceleration requests in a task queue that prioritizes the schedule and allocation for each task. Especially in a highly shared configuration setting, LM, when the workload is high, the average speedup achieves 2.20x. And the runtime distribution shows that the transition overhead has greatly reduced.

Our current runtime task manager is simply halting the accelerator slot for the delayed tasks, which come from the injected deviation under the network congestion. As a result, the edge-sharing system is subjected to a 10%-23% idle state but still obtains great performance gain.

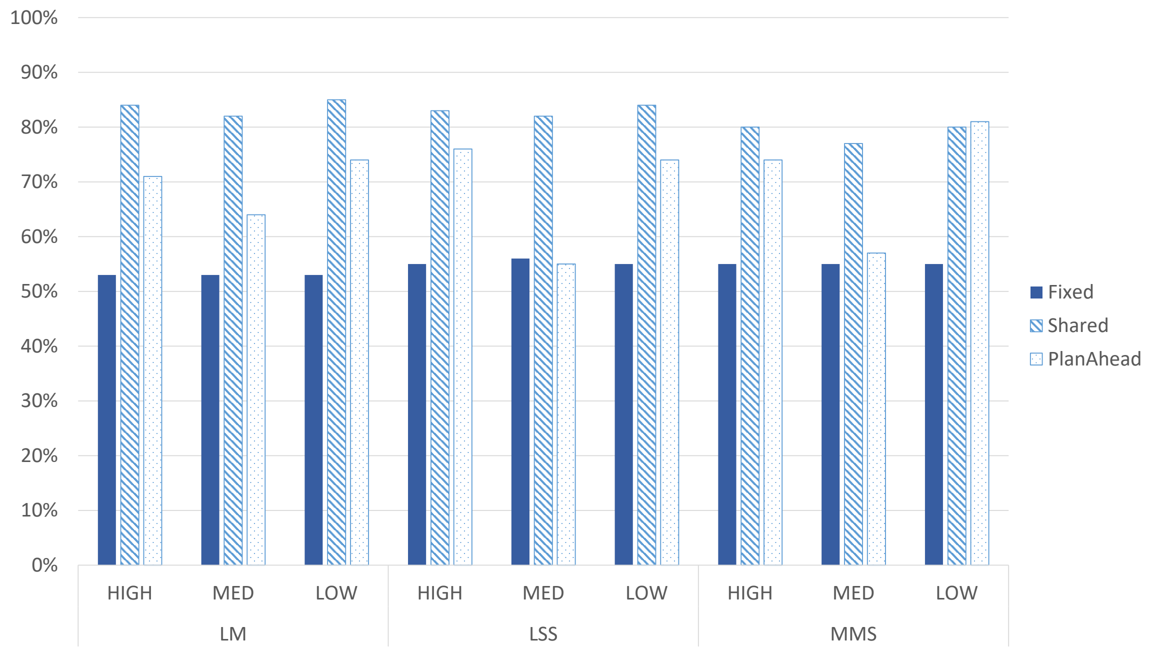

Figure 2.8: DNN Library Usage

While we have manually reduced the design space for deployed DNN library based on the compute resources and slots to our FPGA edge, in Figure 2.8, the *Shared* policy still adopts most accelerators for the sack of sharing DNN accelerators and slots. In our proposed *PlanAhead* policy, the use of the deployed DNN library can be reduced and achieve maximum responsiveness to a queue of acceleration requests. We can also facilitate the deployment of the DNN library to other larger FPGA edge or more diverse acceleration requests. Similarly, to limit the use of DNN accelerators, we can change the objective function in our proposed *PlanAhead* policy to find the minimum use of total DNN accelerators.

2.5 Conclusion

In this chapter, we present a dynamic DNN accelerator-sharing system in FPGA edge devices. Given a queue of requests to access binarized DNN accelerators on FPGA, our proposed offline phase obtains the optimal policy to determine which accelerator to allocate for each request and in what order to respond to the requests. Our proposed framework exploits an a priori known pattern of input task arrivals and matches the suitable accelerators to the tasks according to the utilization of FPGA shared resources and the application's DNN model parameters. The framework achieves maximum responsiveness for a queue of acceleration requests.

Considering the fluctuation in task arrivals, in the next chapter, we'll exploit the regularity of patterns to the input workloads and extend the runtime task manager to include a more sophisticated scheduling scheme to adapt the uncertainty and deviation under various network congestions.

Chapter 3

On Exploiting Patterns For Robust FPGA-based Multi-accelerator Edge Computing Systems

3.1 Introduction

An FPGA-based multi-accelerator edge serves various DNN applications from multiple end devices simultaneously for hardware acceleration. The DNN application can be processed by a set of DNN accelerators from a library of DNN accelerator architectures, each optimized for a certain objective, such as accuracy and performance on the edge. If the system software at the edge processes the acceleration requests on a First-Come, First-Serve (FCFS) basis, the FPGA edge will incur dynamic reconfiguration overhead and degrade the system responsiveness. Researchers have proposed batching the requests during runtime to reduce the overhead, e.g. DeepRT [145], but this can lead to starvation and timed-out requests. In addition, in many edge computing platforms, there is only an online scheduler without any prior knowledge on any periodic nature or some regularity in the arrival time of the tasks such as [83, 72, 48, 18, 63, 133, 153]. However, in various CPS and IoT monitoring applications, sensing occurs at a fixed rate, and hence, the data is sent to the edge for acceleration periodically. When multiple end devices continuously send acceleration requests at a fixed rate, a pattern in the sequence of requests can be observed.

In my previous chapter, I adopted an ideal fixed pattern of arrival times of tasks under a set of periodic tasks for scheduling. In a well-controlled environment without noises, the sequence of tasks can be predicted and formulated without historical data, and, hence, the proposed method takes advantage of the static scheduling method and has shown the potential of optimization [128]. However, noises and uncertainties can happen in the deployment environment due to network delays and system workload congestion. The sequence of tasks becomes less predictable and this pattern is not unique and might change over time. Therefore, instead of assuming the regularity for an ideal fixed pattern, we look into a more sophisticated approach that exploits the historical data and extracts patterns aligned with the deployment environment.

In this chapter, I present a systematic approach to extract the pattern in a sequence of requests to access various accelerators at the pre-deployment phase and exploit it for pre-deployment planning on accelerator allocation and tasks' execution ordering. In addition, at the deployment phase, I propose an adaptive online scheduler to accommodate the network delay variation and mitigate the effect of task arrival time noise.

This system framework leverages the characteristics of input workloads to assist the system in task scheduling and accelerator allocation, acknowledging both predictable patterns and the presence of uncertain or unexpected factors. The experimental results show that using our proposed framework, the responsiveness of response time and timeout tasks is significantly improved, and the timeout tasks are even reduced to zero.

The summary of the contributions is as follows:

- The proposed framework provides a hybrid two-step approach that, during the pre-deployment time, creates optimization opportunities throughout the pattern extraction and static scheduling and accelerator allocation; and, during the deployment time, employs an adaptive online scheduler for the variation in workloads. The results show significantly improving responsiveness and robustness in serving multiple end devices.
- The pre-deployment phase comprises the clustering-based method to extract patterns and the static scheduling and resource allocation, creating optimization opportunities.
- The runtime dynamic exploitation of the static optimization accommodates the network delay variation and alleviates resource contention: An adaptive online scheduler that handles and mitigates uncertainties.

3.2 System Framework

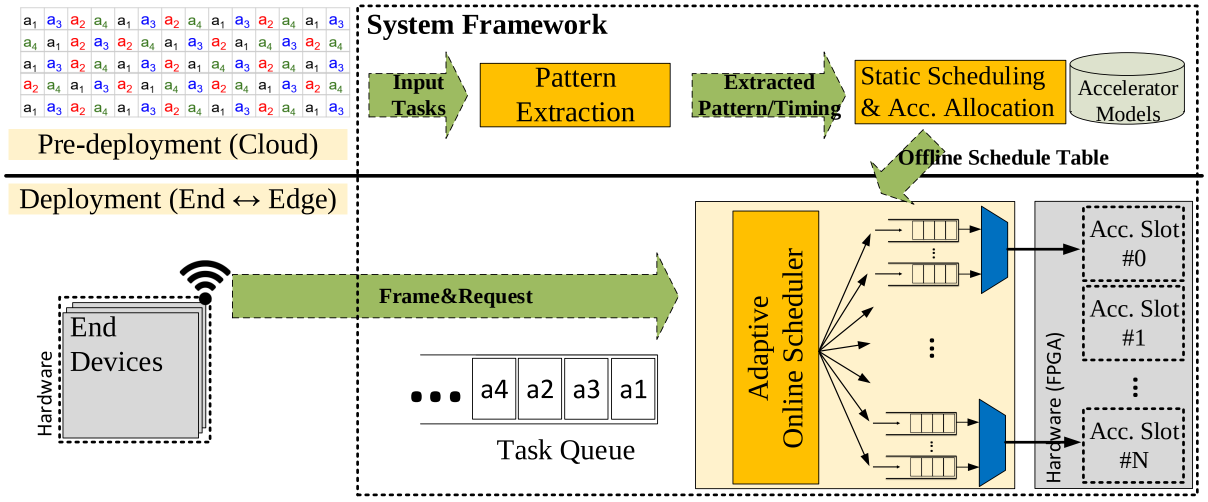

Figure 3.1: System Overview

Figure 3.1 shows my proposed framework. At the pre-deployment phase, pattern extraction is presented to observe the behavior of the tasks' arrival at the edge and find the most dominant pattern in the task queue. Such that the regularity and characteristics of tasks are exploited. Next, I propose a static task scheduling and accelerator allocation, that acquires the extracted pattern, and then generates the offline schedule table. The table includes the tentative ordering of the task and selected DNN accelerators to be implemented on the FPGA.

During runtime, strictly following the Offline Schedule Table might lead to system performance degradation due to random parameters such as network jitter. Therefore, for the deployment phase, I propose a soft real-time scheduler that uses the Offline Schedule Table as a guideline but properly adapts the schedule based on the task arrival times and task queue status.

3.3 Pre-Deployment Phase (Offline)

To extract system characteristics and exploit them to improve the system performance, I propose a pre-deployment phase including two steps 1- pattern extraction, and 2- static scheduling. The output of this phase is expected arrival times and a static schedule that will be used in the deployment phase. The static schedule is an optimized schedule based on the calculated timings from the pattern extraction step.

3.3.1 Pattern Extraction

Figure 3.2 shows an overview of the setup and where pattern extraction plays its role. a_1 to a_n are different applications represented as multiple end devices that simultaneously send tasks to an edge node for acceleration. Each end device periodically sends tasks (t_i) to the

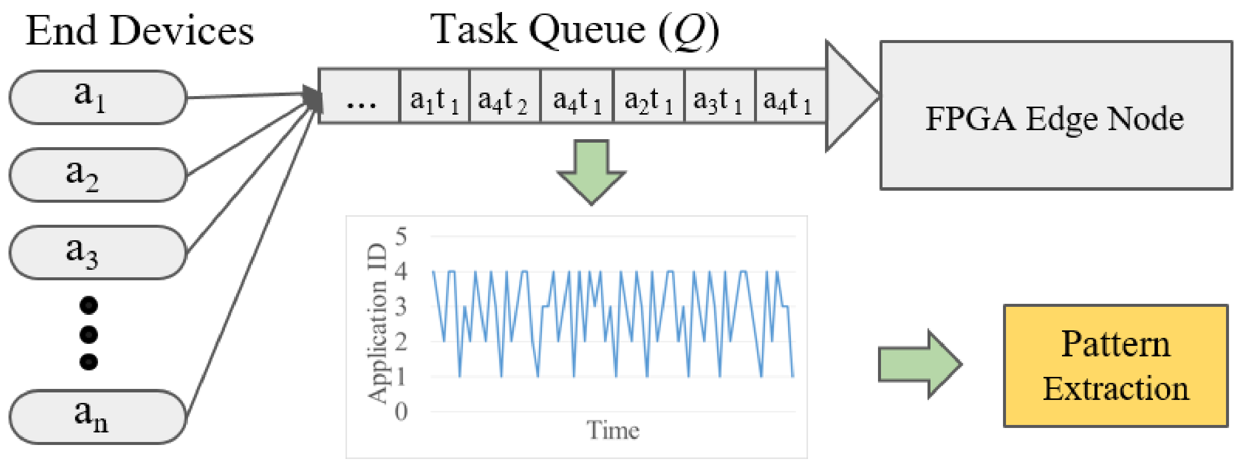

Figure 3.2: Illustration of the regularity of the temporal order of tasks at the edge from a_1 to a_4

edge node continuously, and the edge node receives a queue of incoming tasks (Q). The timeline illustrates how the order of the arriving tasks from different applications varies in different time intervals and yet the pattern extraction approach can be applied to capture regularity over the sequence. To exploit any regularity in receiving acceleration requests at the edge, we use the pattern extraction module to understand the characteristics of the network's behavior. Due to noises and uncertainties in the environment, such a pattern is challenging to extract.

Sequence-based Pattern Extraction: Challenges and Limitations

A standard pattern extraction algorithm identifies sub-sequences that appear frequently within the larger sequence. This most frequently occurring sub-sequence is then recognized as the pattern. The concept of a dominant sub-sequence is crucial in many applications. For example, the sequential pattern mining algorithms in data mining [21] can be used in finding patterns in DNA sequence analysis. In DNA sequences, the identification of these patterns is vital because they can represent common genetic markers, and conserved sequences that have remained unchanged throughout evolution. By identifying these dominant sub-sequences, researchers can gain valuable insights into the characteristics of the historical data. To use these patterns for task scheduling, if the pattern repeats mostly in the historical data, the pattern can be used by a scheduling approach, e.g. *SharedDNN/PlanAhead*, in our previous chapter, to plan ahead and optimize the system performance.

Algorithm 1 Sequence-Based Pattern Extraction

```
    Intput: $Q$                              ▷ historical queue data
    Intput: $min\_seq, win\_size$            ▷ min sequence size, window size per pattern
    Output: $rep\_pattern$                   ▷ sequence-based representative pattern
 1: Create a dictionary $dict\_seq$ for sequence patterns
 2: for $p\_len = min\_seq$ to $win\_size$ do
 3:     $sequence, occurrence$ = find_sequence_pattern($Q$, $p\_len$)
 4:     if $occurrence \geq OCC\_THRESHOLD$ then
 5:         $dict\_seq[sequence] = occurrence$
 6:
 7: /* Extract the pattern $P$ and Cluster irregular tasks $I$ */
 8: $P$ = calc_max_pattern($Q$, $dict\_seq$)
 9: $I$ = calc_irregular_tasks($Q, P, win\_size$)
10: $rep\_pattern = P + I$
11: RETURN $rep\_pattern$
```

Here, in Algorithm 1, I show how a sequence-based pattern extraction approach can be applied to find a representative pattern over the historical queue information, i.e. Q, for task scheduling. The input *min_seq* is defined as the minimum sequence length for the sequence-finding algorithm and the input *win_size* as the target window size for the representative pattern. The above sequence-based approach, firstly, iterates to find various sizes of sequences that occur dominantly in the input Q, and the sequence results are saved in a dictionary (Line 1-5 in Algorithm 1). Then, *calc_max_pattern* finds and calculates timing for a dominant pattern P according to the sequence result dictionary *dict_seq* and the historical queue data Q (Line 8 in Algorithm 1). The *find_sequence_pattern* function can be implemented with any sequence mining or event-analysis algorithms, e.g. [79, 22], that analyze sequences. In our previous collaborated work [106], *EMMA* [36], one of the episode mining algorithms, was used to find the pattern. The pattern P is, then, composed of the temporal order and the average arrival time of tasks.

However, when there exists noise and uncertainty in the environment and the temporal order of tasks becomes irregular, the dominant pattern P may not capture enough sequences to meet the target window win_size requirement. Those irregular tasks within the win_size are not extracted by the sequence-finding algorithm, but these task workloads should still be taken into account for the static scheduling and accelerator allocation. To address the issue of shortage in identifying dominant sequences, the algorithm has to additionally find and cluster the remaining irregular tasks, that follow pattern P, within the win_size (Line 9 in Algorithm 1) and attach them to the dominant pattern to form the representative pattern $P + I$ (Line 10 in Algorithm 1).

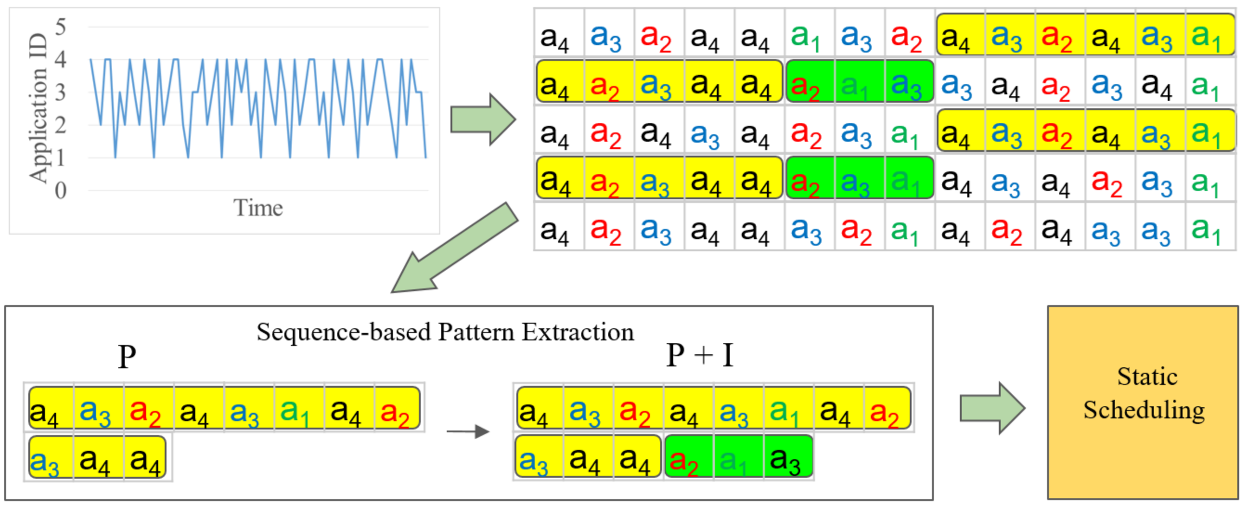

Figure 3.3: The sequence-based pattern extraction approach extracts the dominant pattern P and clusters remaining tasks in the hyperperiod as I and output $P+I$ as the representative pattern

Figure 3.3 includes an example of the incoming task IDs sequence of 4 devices with sampling rates of 2,3,4 and 5 fps below the Q. The historical queue information is visualized by arranging blocks in chronological order, from left to right and top to bottom. The yellow-highlighted blocks denote the pattern $P = [a_4, a_3, a_2, a_4, a_3, a_1, a_4, a_2, a_3, a_4, a_4]$, which is the output of line 8 in Algorithm 1 that finds the maximum pattern and meets the occurrence threshold. Pattern P represents an ordered sub-sequence that occurs frequently from the historical data. The remaining application IDs are tasks that arrive closely to each other and/or, due to network congestion, irregularly deviate in arrival time. within the target window size, there exist some partial deviations from P for the remaining tasks. We average the relative arrival times over those leading-P intervals to find I (i.e. green-highlighted blocks) and create the representative pattern as $P + I$, which includes the pattern and timing information. As the resulting pattern shows in Figure 3.3, the sequence-based approach finds the exact "order" of objects as the sequence P where the sequence failed to represent the pattern within the target window size. Even with the addition of irregular tasks, the representative pattern, $P + I$, is still unable to cover the majority of the example historical data.

Such sequence-based approaches bring the following challenges and limitations. Whenever tasks arrive in an out-of-order manner because of, for example, noises and fluctuation under network congestion, the approach will identify these variations as different sequences and may fail to find the pattern that covers the majority. In addition to the noise and uncertainty, if the end devices' sending rate isn't periodically fixed and tasks are sent in a burst in the deployment environment, the sequence-based approach will suffer in extracting the temporal order of tasks. The sequence of tasks becomes limited in size to extract, and, hence, fails to represent input workload characteristics. As a result, the optimization is compromised due to the difference in task patterns between the pre-deployment and deployment (runtime) phases. The effectiveness of the sequence-based approach can be influenced by several factors, including the quality of the historical data, the chosen frequency threshold, and the specific characteristics of the historical data being analyzed.

Moreover, it's important to note that while sequences may appear different in temporal order, this is not always significant in task scheduling. In task scheduling, the order of tasks can often be flexible, and it's the characteristics of the tasks themselves that hold more importance. This is particularly true when considering the workload to plan ahead, where the exact ordering of tasks is less critical than their general temporal characteristics. This highlights the importance of focusing on the inherent attributes of tasks rather than their order in the sequence. Therefore, I propose the clustering-based approach that doesn't fully rely on the restricted temporal order of tasks (i.e. sequence) but identifies the similarity of patterns where the similarity measurement can be further customized.

Clustering-based Pattern Extraction

The noises and uncertainties in the environment can cause tasks to be received with delay or irregularity. When the system is handling high rates of arrival tasks and large numbers of end devices, tasks tend to arrive irregularly in clusters or bursts. Also, the rates of arrival tasks can be dynamic or non-periodical if other interrupt events happen. As a result, the sequences of tasks vary significantly. The sequence-based approach focuses on the sequence (i.e. the temporal order) of tasks such that it becomes challenging to find a specific representative pattern that covers the majority of sequences. However, there still exists regularity if the data are examined based on the presence and arrival time of tasks. Hence, we propose a clustering-based pattern extraction approach that focuses on the similarity of sequences in terms of task occurrence and arrival time and then obtains the clustered representative pattern.

The similarity measurement method we apply is Jaccard Index [46] (a.k.a. Jaccard Similarity), which is especially effective when the order of items is less important and the presence of items is used for examination. The Jaccard index determines the similarity between two data points by considering the union and intersection of the two points. Such Jaccard index can be defined as the ratio of intersection by the union of the two points. Since our pattern includes (1) the presence and (2) the arrival time of tasks, we define our similarity function that considers both the of the aforementioned metrics such that the measurement method becomes less sensitive to the variant of the order of tasks.

Algorithm 2 Clustering-Based Pattern Extraction

```
    Intput: $Q, win\_size$                        ▷ historical queue data, window size per pattern
    Output: $rep\_pattern$                        ▷ clustered representative pattern
 1: $list\_patterns$ = split_q_by_win($Q, win\_size$)
 2: n = len($list\_patterns$)
 3: Create $n \times n$ matrix $DIST\_MAT$
 4: for $i = 0$ to n-1 do
 5:     for $j = 0$ to n-1 do
 6:         DIST_MAT[i][j] = 1 - find_similarity($list\_patterns[i]$, $list\_patterns[j]$)
 7:
 8: $Z$ = linkage($DIST\_MAT$)                    ▷ Calculate the linkage matrix
 9: $clusters$ = form_flat_cluster($Z$, $DIST\_THRESHOLD$)
10: $rep\_pattern$ = cal_rep_pattern($clusters$, $list\_patterns$)
11: return $rep\_pattern$
12:
13: function find_similarity($pattern1, pattern2$)
14:     /* Find intersection based on task occurrence and arrival time */
15:     $list\_intersection$ = []
16:     for $task$ in $pattern1$ do
17:         if $task$ in $pattern2$ and $diff\_arrival\_time \leq TIME\_THRESHOLD$ then
18:             Add $task$ to $list\_intersection$
19:
20:     $intersection$ = len($list\_intersection$)          ▷ Calculate the size of intersection
21:     $union$ = size_of_union($pattern1$, $pattern2$)     ▷ Calculate the union of $pattern1/2$
22:     $similarity$ = $intersection$ / $union$              ▷ Calculate the Jaccard similarity
23:     return $similarity$
```

Algorithm 2 shows the details of the proposed clustering-based pattern extraction method. To extract the clustered pattern, we first pre-process and split the historical queue data Q into multiple pattern candidates by the interval of the time window win_size. Then, we use our pattern similarity measurement function $find\_similarity$ to calculate the similarity between two pattern candidates (Line 13-22 in Algorithm 2). The similarity index is the value between 0 to 1. Therefore, the distance matrix can be computed by $1 - similarity$ for every pair of these pattern candidates (Line 4-6 in Algorithm 2).

Once we obtain the similarity matrix of our pattern candidates, we use the agglomerative hierarchical clustering solution [136], a bottom-up approach, to cluster the similar pattern candidates based on the distance. The approach starts with the linkage function to decide on the linkage criterion where it takes the distance information and decides pairs of patterns into clusters. Then, these newly formed clusters are linked to each other to create bigger clusters. This process is iterated until all the pattern candidates in the original data set are linked together in a hierarchical tree. Note that, in each iteration of the clustering process, the two smallest-linkage-distance clusters are linked together. We set the distance threshold $DIST\_THRESHOLD$ to determine the partition of our data (Line 8-9 in Algorithm 2). Then, we select the cluster that covers the majority of our data and extract the clustered representative pattern (Line 10 in Algorithm 2).

Figure 3.4 includes the flow of the clustering-based pattern extraction approach which uses the same example input Q history as in the sequence-based one. There is an incoming task ID and arrival time sequence of 4 devices with sampling rates of 2,3,4 and 5 fps below the Q and we visualize the historical queue information (including application IDs and arrival times) by arranging blocks in chronological order, from left to right and top to bottom. The clustering-based approach finds similarity based on the task occurrence and arrival time. Therefore, despite that the temporal orders (i.e. sequences) are different, the pattern candidates in yellow/green/orange/gray-highlighted blocks are considered similar and in the

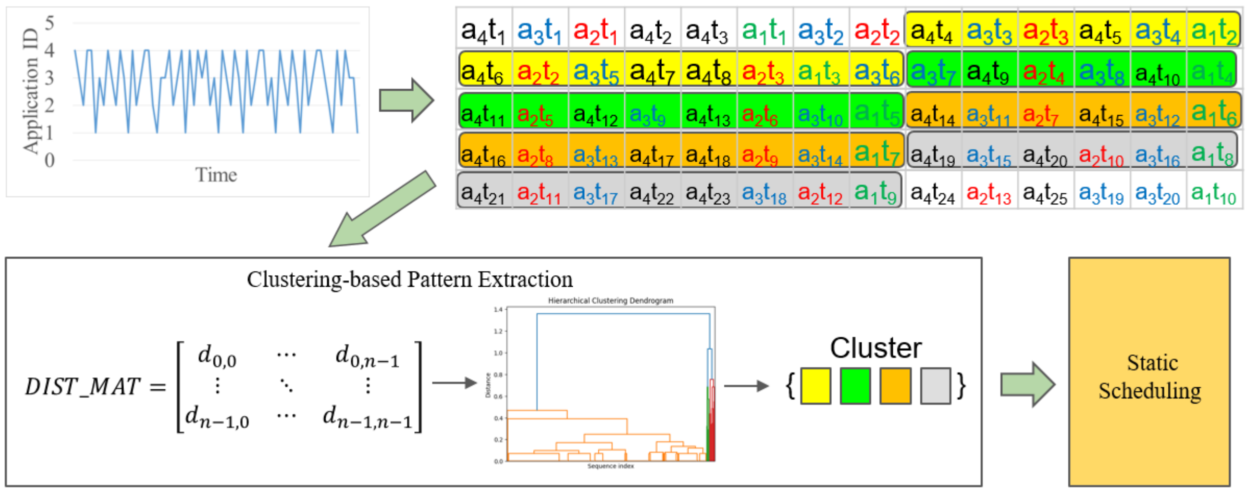
Application ID
Time
Clustering-based Pattern Extraction
$DIST\_MAT = \begin{bmatrix} d_{0,0} & \cdots & d_{0,n-1} \\ \vdots & \ddots & \vdots \\ d_{n-1,0} & \cdots & d_{n-1,n-1} \end{bmatrix}$
Cluster
Static Scheduling

Figure 3.4: The clustering-based pattern extraction module calculates the distances between pattern candidates and finds the cluster that covers the majority of the Q history

same cluster. Therefore, the clustering-based pattern extraction is more tolerant to the fluctuation of task arrival while still being able to capture the regularity in the majority of the Q information.

Next, we show a more complex example and demonstrate the capability of handling a higher range of fps and number of devices. The example is also adopted and shown in the evaluation section of this chapter.

Figure 3.5 shows the dendrogram on a 7-end-device input Q sequence which has a total of 30 fps. The dendrogram is a branching diagram that represents the relationships of similarity among a group of pattern candidates. The x-axis represents the indexes of the pattern candidate. As shown, we can find the clustered representative pattern that covers 97.4% of pattern candidates within the distance 1. The dendrogram can be used to determine the threshold for cutting off pattern candidate selection in the trade-off between pattern coverage and accuracy.

Once the representative pattern is extracted, the system proceeds to explore static scheduling and accelerator allocation, which seeks to maximize the responsiveness based on the extracted

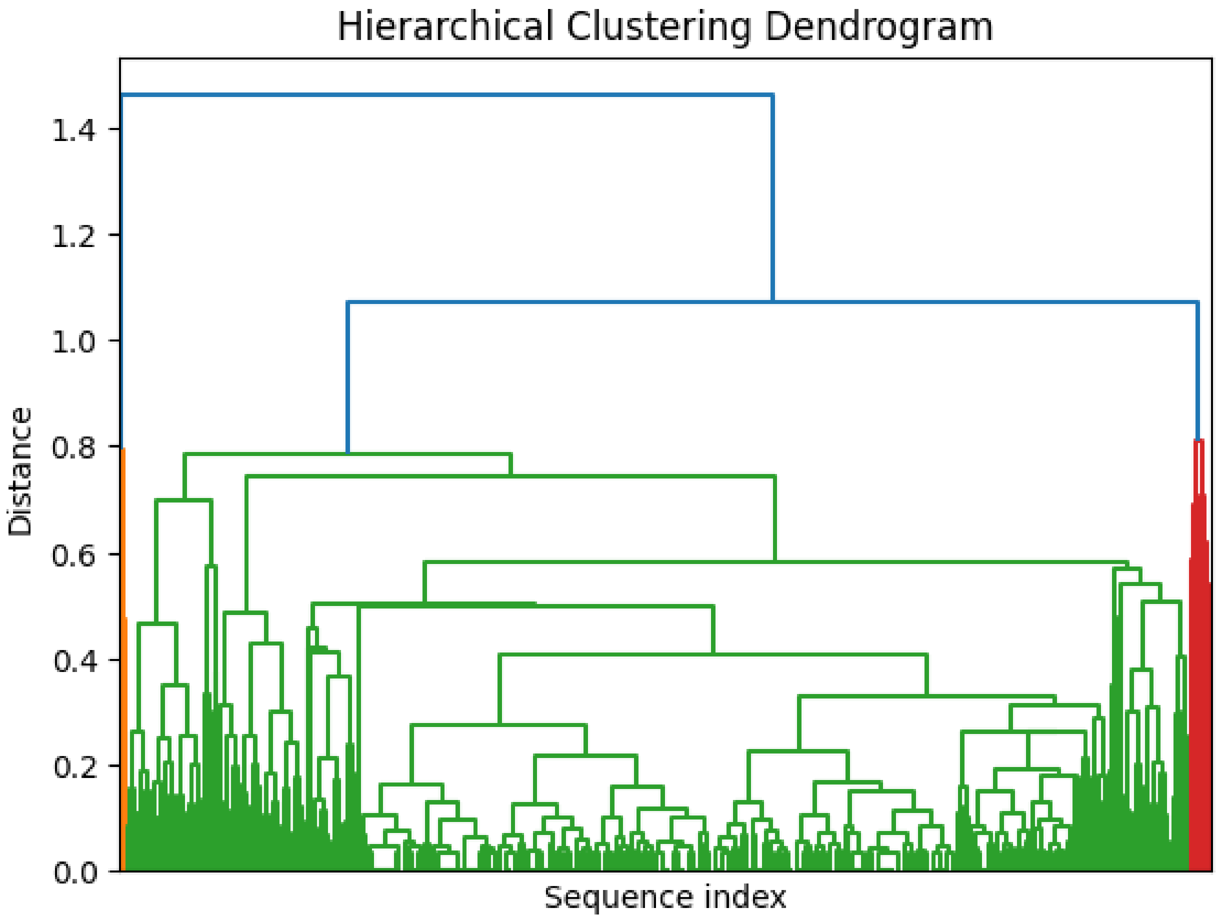

Figure 3.5: Dendrogram Example of the clustering-based pattern extraction approach on a 7-end-device input Q information with 30 fps in total

pattern and timing.

3.3.2 Static Scheduling and Accelerator Allocation

Each task in the representative pattern is a request for inference acceleration under certain requirements, such as applications (i.e. dataset), accuracy, and performance. Given these requirements, tasks might be compatible with various types of accelerators, and mapping them to these accelerators depends on the schedule and available resources.

An FPGA can be partitioned into multiple reconfiguration regions, called accelerator slots, that are able to execute accelerators simultaneously and independently. In each slot, there is a time overhead for reloading (and potentially partial reconfiguration) if the accelerator needs to be changed during runtime. Therefore, for each task, the system must decide when (timestamp) to start the task on which slot using what accelerator. As a target application, we applied the accelerator models in [128] that adopts the Binarized Neural Network accelerator, e.g. [6], with different factors in DNN models and hardware optimization so they can be accommodated properly given the size of the slots and application's requirements, i.e. performance and inference accuracy. For instance, the DNN topology will affect the inference performance. In addition, the same DNN topology can be deployed into different accelerators in various optimization parameters, such as speed and size, without compromising inference accuracy, for example, in Table 3.1, Acc_1 to Acc_3 are different accelerators but the same DNN topology. And some accelerators may not be able to fit in all slots. A high-performance accelerator might require more FPGA fabrics and not fit in a small accelerator slot. On the other hand, a DNN topology could be trained for different applications (i.e. datasets), which results in different coefficients (i.e. weight files). Therefore, an accelerator for a certain DNN topology can be shared among different applications with the same DNN topology, and it just needs weights reloading. There are

Table 3.1: Accelerator Models

Accelerator	Exec_Time † (ms)	Weight Files (<u>Dataset</u>/Accuracy)	Slot {Large, Medium, Small} Compatibility
Acc_1	30	{<u>cifar10</u>/80, <u>cifar3</u>/90, ✖}	{✔, ✖, ✖}
Acc_2	50	{<u>cifar10</u>/80, <u>cifar3</u>/90, ✖}	{✔, ✔, ✖}
Acc_3	100	{<u>cifar10</u>/80, <u>cifar3</u>/90, ✖}	{✔, ✔, ✔}
Acc_4	50	{<u>cifar10</u>/84, <u>cifar3</u>/92, ✖}	{✔, ✖, ✖}
Acc_5	100	{<u>cifar10</u>/84, <u>cifar3</u>/92, ✖}	{✔, ✔, ✖}
Acc_6	200	{<u>cifar10</u>/84, <u>cifar3</u>/92, ✖}	{✔, ✔, ✔}
Acc_7	30	{✖, ✖, <u>mnist</u>/98}	{✔, ✖, ✖}
Acc_8	50	{✖, ✖, <u>mnist</u>/98}	{✔, ✔, ✖}
Acc_9	100	{✖, ✖, <u>mnist</u>/98}	{✔, ✔, ✔}
† the frame resolution is 640 × 480 pixels			

trade-offs between accelerator performance, inference accuracy, DNN topology, and the size of the slot. Therefore, we can acquire a set $\{Acc_1, Acc_2, ..., Acc_n\}$ of accelerator models, where each accelerator Acc_i has a set $\{wt_1, wt_2, ..., wt_n\}$ of weight files wt_i, trained for different datasets. For example, in Table 3.1, Acc_1 can run the cifar10 or cifar3 dataset (with the same DNN topology), but it needs to load wt_1 or wt_2. Note that cifar3 is a subset of cifar10 trained for detecting less number of objects, but with higher inference accuracy. Thus, an accelerator slot can simply load the weight files into BRAMs for other applications if they use the same DNN topology other than re-configuring the accelerator slot. If the DNN topology changes, the slot has to change to support the new topology (new accelerator), which has a partial reconfiguration overhead. We use the variables opr_i and $orel_i$ to denote the overhead of partial reconfiguration and weight files reloading for each accelerator. We represent the elapsed time of inference on the Acc_i accelerator as te_i and the timestamp to start the task i as ts_i.

I formulate the above temporal and spatial scheduling and accelerator allocation problem for tasks in the extracted pattern as a mixed-integer linear programming (MILP) problem and optimize it to maximize the system responsiveness, i.e. minimize the average response time among all tasks in the extracted pattern. Response time includes time to wait for available slots, time for execution on FPGA, and time for reconfiguring/reloading accelerators if necessary. In addition to the primary scheduling equations [128], I added two extended constraints to consider the regularity and workload between hyper-periods as follows:

1. **Cross-Pattern Constraint:** The schedule will repeat after every hyper-period, a potential reconfiguration/reload overhead across its head and tail should be considered.

$$\begin{aligned}&opr_{u,v} + orel_{p,q} \leq ts_{head} \mid (u,p)_{tail}! = (v,q)_{head},\\ &\forall head, tail \in N, \forall u, v \in Acc, \forall p, q \in Weight\end{aligned} \tag{3.1}$$

2. **Load-Balance Constraint:** The maximum duration constraint is set to make sure that all individual slots' schedule for the pattern will not exceed the hyper-period; thus, preventing the growing tasks in the queue and risks of timeout.

$$ts_{tail} - ts_{head} \leq HYPER - PERIOD, \forall head, tail \in N \tag{3.2}$$

To maximize the responsiveness, I formulate the objective in minimizing the average response time for all the tasks that are scheduled on the given accelerator slot configuration,

$$\begin{aligned}&min(\frac{1}{N}\sum_{i \in N} resp_i),\\ &resp_i = \{ts_i + te_u - ArrivalTime_i | \forall i \in N, \forall u \in Acc\}\end{aligned} \tag{3.3}$$

Our goal is to provide a static scheduling and accelerator allocation for each task in the extracted pattern. We seek to minimize the average response time which, as a result, can efficiently use the shared computing resource on all available accelerator slots in temporal and spatial aspects. We formulate the above optimization problem as a mixed-integer linear programming (MILP) problem and solve it using mathematical programming solvers such as IBM CPLEX© Optimizer [39].

3.4 Deployment Phase (Online)

The Offline Schedule Table determines the schedule of accelerators on each slot of the FPGA based on the expected arrival times of tasks. However, because of various factors such as network delay variation, the task arrival times will drift from the expected arrival times after a while which will negatively affect the response time. We propose an adaptive Online scheduler to handle these uncertainties during runtime.

3.4.1 Adaptive Online Scheduler

Task arrival time varies from one task to another due to random parameters such as network jitter. We have obtained the optimal scheduling from the extracted patterns and timing of tasks in the pre-deployment phase. To exploit this knowledge, and also reduce the effect of tasks' arrival time fluctuation on the response time, we propose a lightweight Online scheduler, which approximately follows the Offline Schedule Table while accommodating the uncertainties. The adaptive online scheduler we are proposing follows the accelerator allocation from the offline result but allows additional out-of-order execution of tasks to accommodate the uncertainties of task arrivals. By doing these, our proposed online scheduler uses Offline Schedule Table as a guideline and adaptively reacts to the arbitrary arrival of tasks to increase the system's responsiveness.

To achieve that, there are multi-accelerator-task-queue buffers allocated per accelerator slot. Based on the assignment in the offline schedule table, every arrived task will be placed into

Algorithm 3 Adaptive Online Scheduler

```
Input: taskQueue                                        ▷ Task queue on the edge
Input: AccSchedule                           ▷ Offline Schedule Table for all slots
Input: AccQueue                          ▷ Accelerators' task buffers for all slots
1: /* Run in parallel using multi-threading */
2: assign_task(taskQueue, AccSchedule, AccQueue)    ▷ Assigning tasks into AccQueue
3: for slot_id = 1 to N do                                ▷ N = number of slots
4:     slot_scheduler(AccSchedule[slot_id], AccQueue[slot_id])  ▷ One thread per slot
5:
6: function slot_scheduler(AccScheduleSlot, AccQueueSlot)    ▷ Running as a thread
7:     while sys_terminate != True do
8:         while not (AccQueueSlot[AccIndx]).empty() do
9:             if Check_AccTime_End(AccIndx) and Check_Over_Borrow() then
10:                break
11:            Task=AccQueueSlot[AccIndx].pop()
12:            if not_timed_out(Task, WaitTime_Threshold) then
13:                run(Task);
14:        if Check_Pre_Fetch(Task) or Check_AccTime_End(AccIndx) then
15:            AccIndx=fetch_next_accel(AccScheduleSlot,'roundrobin')
```

the corresponding buffer. Tasks will be processed when the accelerator is on the scheduled time. Algorithm 3 shows the pseudocode for Online scheduler. The inputs to the algorithm are (1) a queue (*taskQueue*) which holds all arrived tasks in the order they arrive, (2) the Offline Schedule Table (*AccSchedule*), and (3) the buffers for assigned acceleration tasks (*AccQueue*).

All incoming tasks in *taskQueue* are assigned in an FCFS order to the corresponding accelerator's queue (*AccQueue*) based on the Offline Schedule Table (*AccSchedule*). Note that each accelerator has its accelerator task queue. There are N independent accelerator slots on the edge. Therefore, each slot has its own schedule for processing accelerator tasks. We run the task assignment and the scheduler for each slot in parallel using multi-threading (Line 2-4 in Algorithm 3).

Online scheduler has two main policies to mitigate the effect of tasks' arrival time fluctuation:

1. **Extensive Batching Policy**: it processes all the tasks in the current accelerator's

queue, even if the time for the currently loaded accelerator has passed the scheduled duration in Offline Schedule Table. To prevent starvation from excessively 'borrowing' time from the next accelerator in line, if there is any task in the next accelerator's task queue that is being timed out due to the delay of execution, the scheduler will proceed to fetch the next accelerator task (Line 8-13 in Algorithm 3).

2. **Early-End Pre-Fetching Policy**: if the currently loaded accelerator finishes the scheduled batch of tasks earlier than it was expected, the next accelerator will be pre-fetched, earlier than the scheduled time in Offline Schedule Table. Therefore, the slot can start to process the tasks in the next queue earlier (Line 14-15 in Algorithm 3).

Using these two policies, Online scheduler utilizes Offline Schedule Table as a guideline rather than strictly following it, and accommodates the task arrival time fluctuations.

The Offline Schedule Table can include various accelerator task schedules on each slot; However, only one accelerator can be deployed at a time. We call the accelerator that is currently loaded on the slot the current accelerator and name the accelerator type as $AccIndx$. For each task in the current accelerator queue $AccQueue[AccIndx]$, the scheduler pulls tasks in an FCFS order and runs on the FPGA's slot if the task is not timed out, i.e. waiting in the queue for more than a certain time threshold $WaitTime\_Threshold$. The accelerator call is non-preemptive. The scheduler keeps running the tasks in $AccQueue[AccIndx]$ on the current accelerator till there is no task left in the queue, or the scheduler is overly 'borrowing' time from the next accelerator and causes any tasks in the next accelerator task queue to be timed out. The scheduler will, then, fetch the next accelerator. When there are no tasks in the current accelerator queue $AcTskQs[AccIndx]$, the online scheduler checks if proceeding to the next accelerator using the following two conditions (Line 14) as follows:

1. *Check_Pre_Fetch* returns if the last processed task is the ending task for the current accelerator schedule in *AccSchedule*[*AccIndx*]. For example, if the *AccQueue*[*AccIndx*] is assigned to process a batch of tasks $t_1, t_2, .., t_i$ based on expected arrival times, *Check_Pre$_F$etch* returns if the last processed task in Line 8-13 is t_i. Using this flag, if the final task execution finishes early, the slot scheduler takes advantage of the idle time, proceeding to fetch the next accelerator, which will reduce the overhead in switching accelerators and increase the slot utilization.

2. *Check_AccTime_End* returns if the current accelerator's time is over based on *AccSchedule*.

If both conditions #1 and #2 are false, it means that there exist expected tasks arriving in *AccQueue*[*AccIndx*], hence, the scheduler keeps the current accelerator on the slot. Otherwise, Online scheduler fetches the next schedule on a round-robin basis based on Offline Schedule Table, which provides fairness and prevents starvation.

3.5 Evaluation

3.5.1 Platform Setup

We used a Xilinx Ultrascale+ MPSoC ZCU104 board as the FPGA Edge, partitioned into three accelerator slots, and multiple Raspberry Pi 3B+/4B as the End devices. The End devices (Edge) are connected to a wireless access point through 5GHz WiFi (Ethernet), and communicate using TCP/IP socket in an office environment. For each experiment, end devices, starting at a random time, periodically send requests for acceleration to the FPGA edge for 10 minutes. We set the timeout threshold for tasks to 1 second. The accelerator models we applied are from [128], which perform DNN inference for vision applications, such as object detection and classifications. We measured and reported the time interval between the task arrival at the edge and the beginning (end) of execution on the FPGA edge as the wait (response) time.

3.5.2 Evaluation Scenario

Input Wokrload Characteristics

We explore seven end devices periodically sending requests for acceleration to the Edge. Two types of input workloads are deployed for evaluation as follows:

- <u>Fixed Input Workloads</u>: There are three different workloads representing 34/30/26 fps in total. We also explore the variants of fps per end device. For example, in Table 3.3, w34d2, w34d1, and w34d0 achieve a total FPS of 34 but in different fps distribution per end device, and, for the former one, the FPS for individual end devices are $\{2, 2, 5, 5, 6, 6, 8\}$. Other workloads of 30/26 fps in total follow the same naming rule.
- <u>Dynamic Input Workloads</u>: Compared to the aforementioned fixed workload, the sending rate for each end device will dynamically change between every interval. Therefore, the total input workload on the edge could vary at every interval. The total input workload is set to be about 32 fps on average to avoid having workloads beyond the compute capacity on the FPGA edge. We also explore the variants of fps per end device. For example, in Table 3.4, $w32d2_{dyn}$, $w32d1_{dyn}$, and $w32d0_{dyn}$ achieve a total FPS of 32, and for the former one, the FPS for individual end devices starts with $\{1, 2, 4, 5, 6, 7, 7\}$ and varies at every interval. We set the interval as 1 minute which results in a change of the total workload on the edge every 1 minute.

Since the WiFi noise level in an office environment is limited, in both types of workloads, during runtime, we add various levels of extra delays with normal distribution [148] ($X \sim \mathcal{N}(\mu, \sigma)$ where $0 < \mu < 70$, and $0 < \sigma < 10$) to the end devices to evaluate the performance of the proposed framework in tackling the changes in network delay and end device timing. Note that, in the fixed input workload, there are marginal differences in responsiveness for using sequence-based or clustering-based approaches during the pre-deployment phase in our evaluation. That is, the reported results in Table 3.3 are based on the clustering-based pattern extraction approach.

Baseline Algorithms

We compared our proposed method with two other scheduling methods, $FCFS$ and $SCYLLA$ [48] as the baseline.

- $FCFS$: accelerators and slots are shared among tasks and applications, and tasks in the task queue are processed in an FCFS order.
- $SCYLLA$ [48]: a heuristic scheduling scheme that involves multiple decision phases, including task division, scheduling determination, and model selection. We took the scheme and customized it for our accelerator models and environments; thus, tasks (accelerators) can be dynamically scheduled/allocated to our three heterogeneous accelerator slots.

3.5.3 Result and Discussion

Experimental Results in fixed input workloads

<u>Input Regularity Analysis</u>: we now analyze how our pattern extraction approach works

efficiently in the fixed-input-workload experiments. Figure 3.6 shows the dendrogram of the input workload, which is profiled by the clustering-based pattern extraction approach (Section 3.3.1). As it is shown, there exists the dominant cluster which covers the majority of the pattern candidates within the range of distance one for all our experiments. The result demonstrates that, in a fixed input workload of such IoT monitoring applications, there exists regularity and patterns to exploit and improve the system performance/responsiveness.

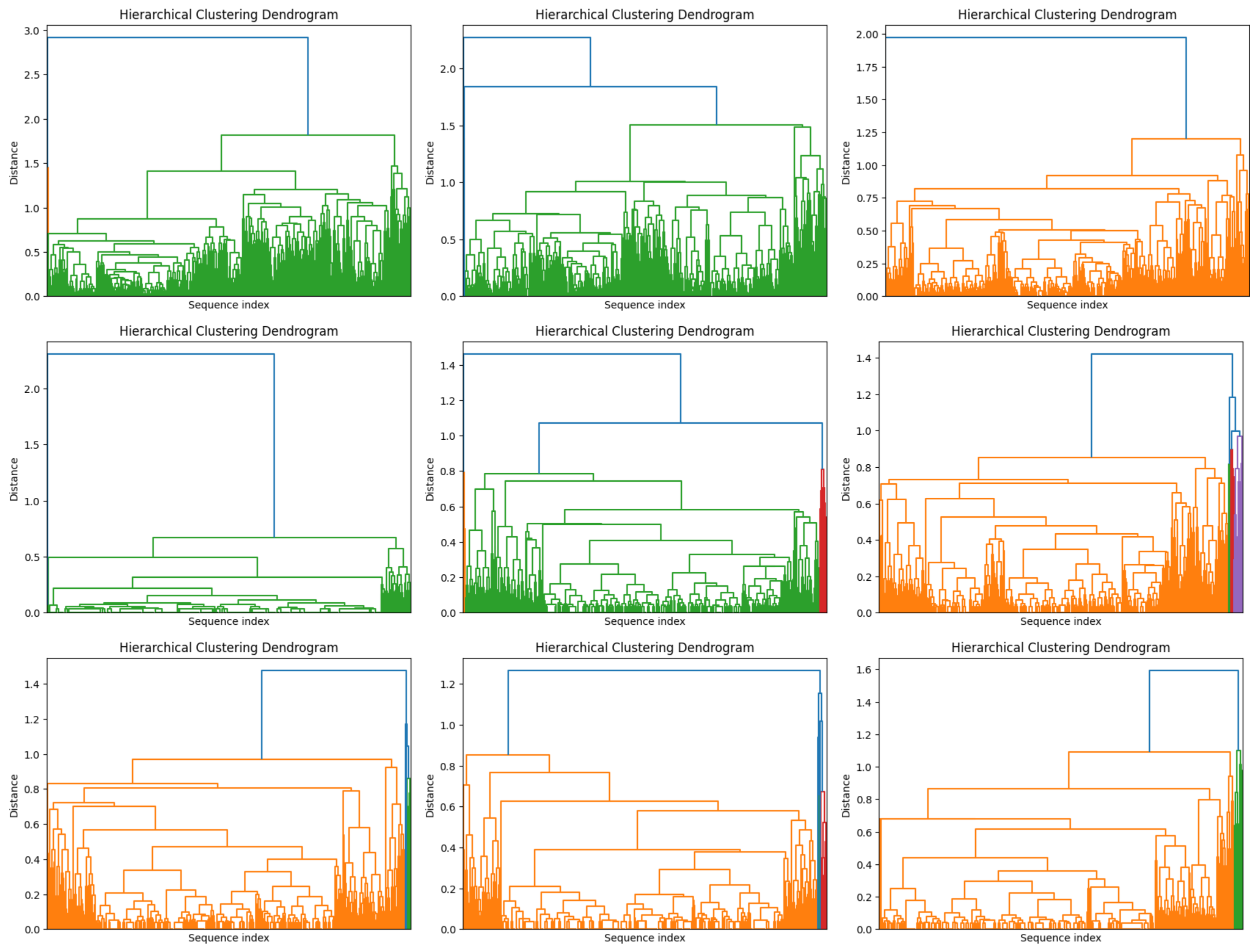

Figure 3.6: Dendrograms for fixed input workloads

Responsiveness Analysis for the proposed framework:

Given the regularity of the input workloads shown in the previous section, we, next, analyze the responsiveness improvement in our proposed framework which involves the pre-deployment and deployment phases. Exploiting the pre-deployment knowledge from the pattern extraction and static scheduling approach, the system has provided enhanced responsiveness to various fixed input workloads ranging from 26 to 34 fps and their variants. However, strictly following the Offline Schedule Table makes the system vulnerable to task arrival time fluctuations. As shown in Table 3.2, some tasks have timed out in *Ours* when only the offline method is applied.

Table 3.2: Responsiveness Analysis of the purposed framework with and without deployment phase

		Ours (Offline Only)		*Ours* (Offline+Online)	
WL.	FPS	Tdrop	Rsp	Tdrop	Rsp
w34d2	2,2,5,5,6,6,8	0	245	0	218
w34d1	3,4,5,5,5,6,6	4	198	0	220
w34d0	4,5,5,5,5,5,5	57	335	1	230
w30d2	2,2,3,4,5,7,7	4	198	0	184
w30d1	2,4,4,4,5,5,6	0	149	0	201
w30d0	4,4,4,4,4,5,5	0	188	0	180
w26d2	2,2,3,3,3,5,8	0	135	0	122
w26d1	2,3,3,4,4,5,5	15	180	0	168
w26d0	3,3,4,4,4,4,4	0	132	0	159

$\star$ There are $10mins \times 60 \times FPS$ tasks in total in each experiment.

In addition to the pre-deployment phase, we utilize the system knowledge and handle the random arrival times by using Online scheduler during the deployment phase. As a result, the responsiveness of the response time and timeout tasks has improved, and timeout tasks are even reduced to zero. Note that the computation overhead of Online scheduler is negligible (the relative CPU utilization on the Edge was under 5% during our experiments). Tasks assigned to the accelerator are based on Offline Schedule Table, the same in both analyses. The execution times are the same for both with and without the deployment phase.

In our proposed framework, combining both the pre-deployment and deployment phases, the system achieves great responsiveness and tolerates the task arrival time variation.

Evaluation with Baseline Methods: In the previous section, we have shown and analyzed

Table 3.3: Comparison (Tout/Rsp denotes the number of timeout tasks/the average response time.)

		$FCFS$		$SCYLLA$ [48]		$Ours$	
WL.	FPS	Tdrop	Resp. Time	Tdrop	Resp. Time	Tdrop	Resp. Time
w34d2	2,2,5,5,6,6,8	8343	872	1190	341	0	218
w34d1	3,4,5,5,5,6,6	7789	881	3088	377	0	220
w34d0	4,5,5,5,5,5,5	8074	873	2787	303	1	230
w30d2	2,2,3,4,5,7,7	5836	876	631	301	0	184
w30d1	2,4,4,4,5,5,6	5915	880	604	296	0	201
w30d0	4,4,4,4,4,5,5	6464	876	2307	317	0	180
w26d2	2,2,3,3,3,5,8	4348	859	189	278	0	122
w26d1	2,3,3,4,4,5,5	4474	864	600	285	0	168
w26d0	3,3,4,4,4,4,4	4577	866	1233	296	0	159

$\star$ There are $10mins \times 60 \times FPS$ tasks in total in each experiment.

the responsiveness improvement in cooperation with the pre-deployment and deployment phases in our proposed framework. In Table 3.3, we compare the result against the baseline methods in terms of response time and the number of timeout tasks. For the $FCFS$, there are large amounts of timeout tasks and high response time. $FCFS$ shares slots to all applications; however, processing tasks in an $FCFS$ order lacks the consideration of overhead in re-allocating accelerators, which results in recurring reconfiguration/reload. Unlike the $FCFS$ method, $SCYLLA$ batches and runs execution sequences when tasks per slot are allocated with the same accelerator, and, therefore, avoids intermittent accelerator reconfiguration and achieves improved responsiveness.

In our proposed framework, tasks not only run on an out-of-order basis to avoid changing accelerators intermittently but also are dynamically batched to improve the throughput. The pre-deployment phase uses the schedule and accelerator allocation module which considers sharing accelerators, and slots, together with the extracted patterns/timing. As a result, $Ours$ significantly improves both response time and the number of timeout tasks compared to $FCFS$ and $SCYLLA$.

Experimental Results in dynamic input workloads

Input Regularity Analysis: When IoT applications are event-based and/or change at every interval. the patterns in the task queue on the edge can differ over time. We profile the similarity of patterns in these dynamic input workloads. Figure 3.7 shows the dendrogram of the dynamic input workloads. Compared to the analysis in fixed input workload (Figure 3.6), both the variants of clusters and distances between pattern candidates/clusters increase, which becomes challenging in extracting the representative pattern.

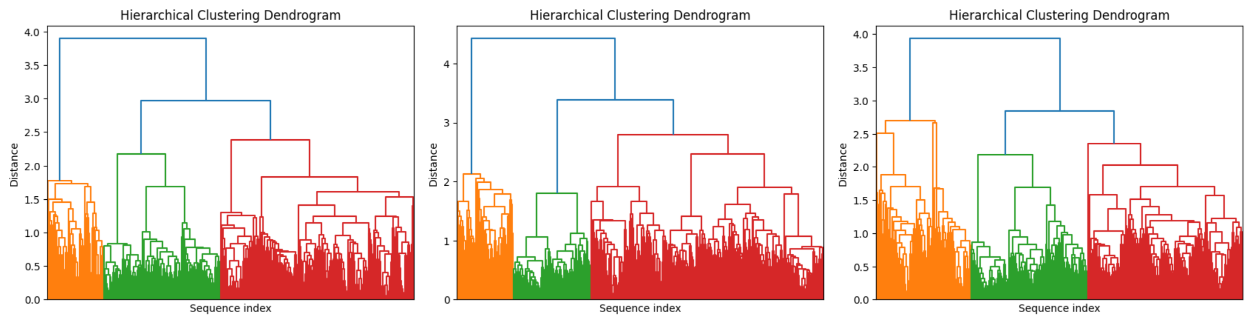

Figure 3.7: Dendrogram for dynamic input workloads

Responsiveness Analysis for the proposed framework using variants of pattern extraction approaches: In the previous section, we have seen an increase in irregularity in the historical queue information. We now analyze the impact of pattern extraction approaches on responsiveness in the dynamic input workload experiments. We apply different implementations for the pattern extraction module: (1) the sequence-based pattern extraction approach from our previous work [106], (2) the fine-tuned sequence-based pattern extraction approach (Algorithm 1), and (3) the clustering-based one (Algorithm 2 in Section 3.3.1). Note that (1) and (2) are both sequence-based pattern extraction approaches. When the pattern sub-sequence has been determined, the former adopts the hyperperiod time as the window size for clustering the remaining irregular tasks where the occurrence of the sub-sequence affects the cluster. Hence, the representative pattern becomes biased to the sub-sequence. We update the mechanism to adopt the specified target window size as the input where we set the size of the average workload in our experiments. So that the expected load of tasks can be kept and planned in static scheduling.

	Our framework w/ Sequence-based Pattern Extraction in [106]		*Our* framework w/ Sequence-based Pattern Extraction (fine-tuned)		*Our* framework w/ Clustering-based Pattern Extraction	
	Tdrop	Resp. Time	Tdrop	Resp. Time	Tdrop	Resp. Time
$w32d2_{dyn}$	4080	265.1	1387	304.7	142	279.6
$w32d1_{dyn}$	2244	199.8	694	328.0	0	235.1
$w32d0_{dyn}$	2876	226.2	82	373.5	61	292.9

$\star$ fps($w32d2_{dyn}$) = {1, 2, 4, 5, 6, 7, 7}
$\star$ fps($w32d1_{dyn}$) = {2, 4, 4, 5, 5, 5, 7}
$\star$ fps($w32d0_{dyn}$) = {4, 4, 4, 4, 5, 5, 6}

Table 3.4: Comparison of the proposed framework using the sequence-based or clustering-based pattern extraction approaches on dynamic-input-workload experiments

Table 3.4 shows the result of using the above three different pattern extraction approaches to our proposed framework. In such dynamic workloads, the clustering-based pattern extraction approach outperforms both the sequence-based ones. Since the number of sequence variants increases, the sequence-based pattern extraction approach exhibits much less efficiency in finding the exact order of the pattern; as a result, the irregular tasks become dominant to the pattern. The regularity of the pattern is not extracted anymore, and those noises and task arrival time fluctuations are concluded in the representative pattern. However, the clustering-based approach is capable of finding the clustered representative pattern that can cover the majority of pattern candidates, where the similarity doesn't simply rely on the order of tasks. Therefore, in such dynamic workloads, using the clustering-based pattern extraction approach for the proposed framework is recommended.

3.6 Related work to increase the robustness of patterns

In the pre-deployment phase, the representative pattern will be used for static scheduling and accelerator allocation. However, given that the noise and uncertainty happen in the deployment environment, the optimization could be deduced due to the discrepancy in task patterns between the pre-deployment and deployment (runtime) phases. The reason is that the result of the static scheduling module determines an optimal schedule of accelerators on each slot of the FPGA. If a task arrives when the corresponding accelerator is not loaded, it has to wait in the queue till the accelerator becomes available. Therefore, the task arrival time will affect the task response time. If the task arrival times do not match the timing that the static scheduling module used to optimize the schedule, the response time will increase. On the other hand, because of variations in network delay for reasons such as mobility of the device, change in the network traffic, and environment change, the arrival times will drift from the expected arrival times after a while.

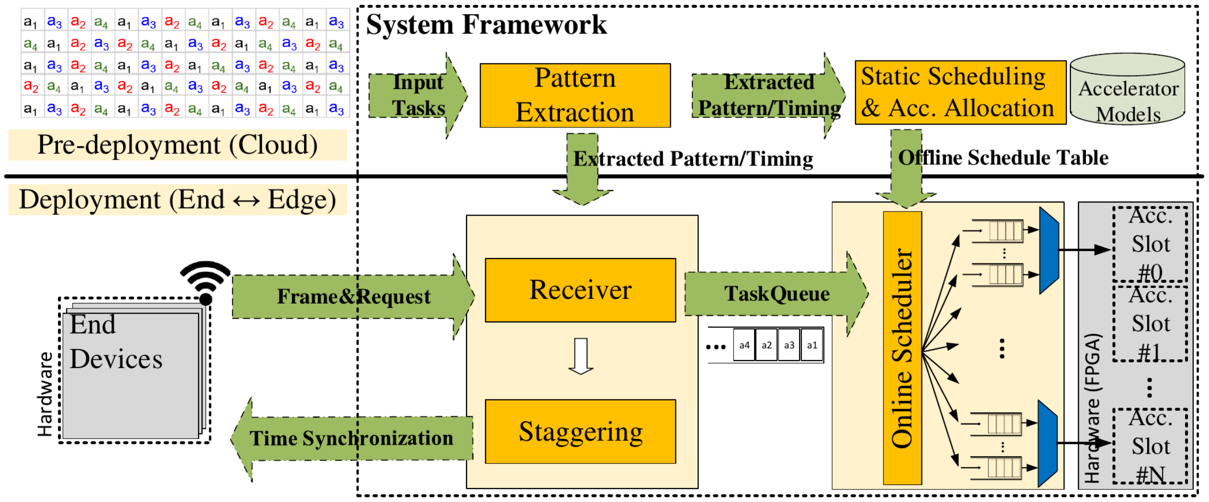

Figure 3.8: Integration with the Task Arrival Time Staggering module [106]

In our collaborated work [106], we propose to integrate a task arrival time staggering module that adjusts the end device sending time during the deployment phase, so the edge receives tasks close to expected arrival times (See Figure 3.8). The proposed module is a feedback loop that monitors the task arrival time during runtime and guides end devices to adjust their sending time, so the arrival times of the incoming future tasks become aligned with the expected arrival times. By applying Staggering module, the edge sends 'Wait' commands to the end nodes to adjust the sending time which, as a result, increases the regularity of task arrival. However, not all end devices allow the change of sending/sampling time, and, also, this falls outside the scope of the dissertation.

3.7 Conclusion

In this chapter, we present a system framework that extracts patterns of tasks at the edge for static scheduling and accelerator allocation, and, during runtime, employs a soft real-time scheduler to improve the responsiveness on a multi-tenant FPGA-based DNN acceleration system.

However, considering IoT applications are mostly event-driven, characterized by varying task rates changing over time, the input workload experiences dynamic and uncertain changes. In the next chapter, I'll extend the two-step approach to include a learning-based scheme to learn insights into task scheduling and accelerator allocation such that the system adapts to the dynamic of changing edge conditions.

Chapter 4

Learning-based Multi-Accelerator Management for Deep Learning Applications on the FPGA Edge

4.1 Introduction

In the previous chapter, we focus on the exploitation of regularity in input workloads with pattern extraction and offline scheduling modules and the integration of an adaptive online scheduler. While the offline phase leverages regular patterns, the online scheduler serves as a responsive mechanism. This mixed offline/online method has proven effective in handling consistent DNN workloads under noise and uncertainty due to network congestions.

However, other than the stable monitoring system, there are more challenges in IoT systems where IoT applications mostly operate in an event-driven paradigm, characterized by various task rates that evolve over time The input workload characteristics can drastically change from one state to another. And, usually, the event is not predictable and the system may not be able to make its sample data for the above offline/online approach. Even though collecting and making sample data is possible, the approach still takes days to update the scheduling and allocation. So the system is not resilient and flexible to the changes in input workloads. Therefore, We need a fully online runtime system software that exploits the current input workload characteristic and can adapt to the changes as well, which brings the following work, the learning-based approach.

Existing works on FPGA-based accelerator management at the edge are mostly online heuristic methods [48, 72, 59, 60] that offer solutions to make locally optimal decisions and they fail to observe and capture the input workload characteristics, hence, leading to poor accelerator utilization. In contrast, mixed offline/online methods [128, 106], while effective in handling consistent workload scenarios, may exhibit limitations in capturing the patterns of dynamic and evolving workloads. They rely on predetermined offline strategies and might result in sub-optimal performance when faced with unexpected changing conditions. It also becomes challenging to extract a diverse range of dynamic input patterns and plan ahead accordingly. Additionally, the pre-defined nature of offline methods may struggle to adapt to real-time fluctuations, leading to potential inefficiencies in resource utilization. As a result, the mixed offline/online approach lacks the adaptability of managing resources in non-static computing environments.

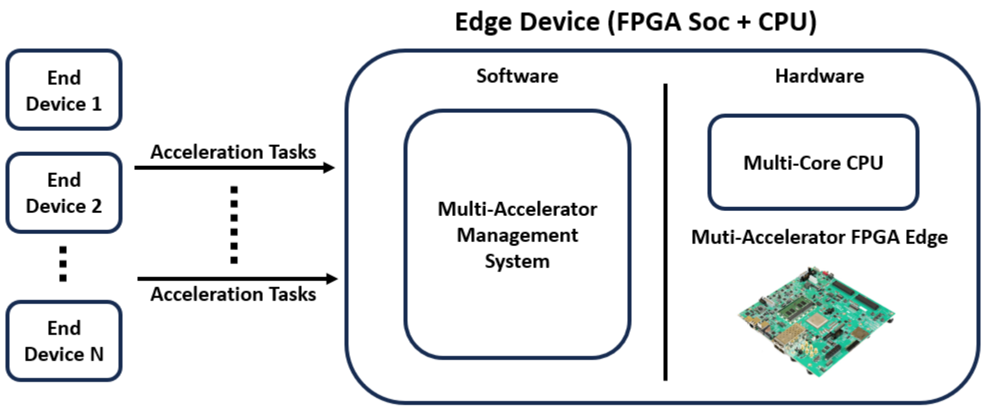

Figure 4.1: System Model

In recent years, deep reinforcement learning methods have shown the potential and superiority of decision-making in dynamic environments. There has been quite an amount of effort in applying reinforcement learning for resource management in general-purpose edge and cloud computing [12, 37, 131, 159, 119, 81, 141, 156, 120]. Most of the works are not designed with compute resource scarcity or custom acceleration in mind. In addition, due to the lack of consideration for the heterogeneity and restriction in allocating computing resources, they are not suited for multi-accelerator FPGA edge systems. Sheng et al. [120] is related work in this category that can adapt to the FPGA accelerator model. However, the proposed neural network model and architecture cannot fully support hardware-acceleration-specific needs such as the heterogeneity in custom hardware accelerators, the hardware configuration overhead, and the accelerator sharing mechanism. In [44], an ML-based approach is proposed for power management on an FPGA edge during accelerator allocation. However, it cannot support out-of-order execution and heterogeneous sets of accelerators, hence, not applicable for scheduling and accelerator allocation.

In this chapter, we propose a learning-based multi-accelerator management framework in an FPGA-based edge system, where the dynamic scheduling for DNN applications and accelerator allocation on the shared FPGA resource are jointly optimized on the fly. In order to reduce configuration overhead on competing accelerators, we promote the accelerator-sharing policy and exploit the dynamic accelerator/slot configuration. The goal is to maximize the system throughput and minimize the task drop rate on the edge. The proposed model provides dynamic slot configuration on the FPGA edge during runtime to improve resource efficiency and enhance overall system performance. In order to monitor the workloads for each acceleration type, we propose a multi-queue module in which there is a separate queue for each acceleration type.

An asynchronous learning mechanism is proposed in order to continuously and concurrently make scheduling and allocation decisions while improving the policy through learning, and, hence, the system adapts to changes in input workloads. The *Exploration* actors are proposed to expedite exploration during the deployment time and are still lightweight enough to process on a single edge node. In our proposed framework, both the learning process and interactions with the environment (actors) are adequately lightweight that they can run entirely within the target resource-constrained edge system and provide nearly real-time data processing.

We apply DNN-based vision applications to our FPGA edge system. Experimental results show that the proposed framework can achieve significant improvement of up to 2.1×/1.9× in average throughputs and 45.5%/41.0% lower in the task drop rates compared to state-of-art heuristic [48] and learning-based [120] approaches.

4.2 Multi-Accelerator FPGA Edge Platform Overview

In this work, the underlying edge node comprises an FPGA-based multi-accelerator hardware connected to a multi-core CPU. Multiple end devices send DNN acceleration tasks to the edge; therefore, the edge receives a stream of requests to accelerate DNN-based tasks, such as image classification and object detection in vision applications.

The goal is to provide efficient task scheduling and a dynamic accelerator allocation on the multi-accelerator edge that seeks to maximize the system throughput in processing acceleration tasks while satisfying the required accuracy and performance. Figure 4.2 shows an overview of the system platform that consists of the FPGA-based multi-accelerator edge hardware infrastructure and our proposed multi-accelerator management system software.

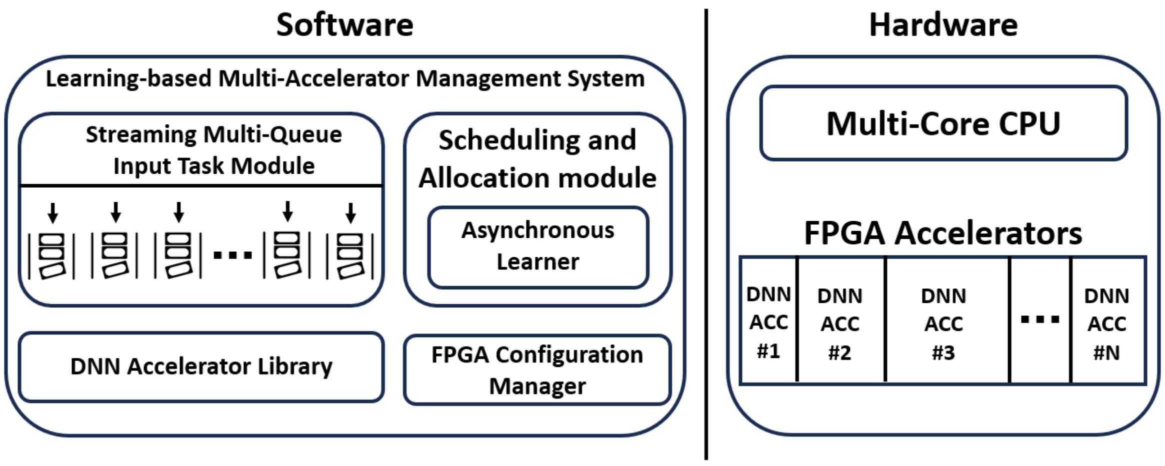
Software
Hardware
Learning-based Multi-Accelerator Management System
Streaming Multi-Queue Input Task Module
Scheduling and Allocation module
Asynchronous Learner
DNN Accelerator Library
FPGA Configuration Manager
Multi-Core CPU
FPGA Accelerators
DNN ACC #1
DNN ACC #2
DNN ACC #3
DNN ACC #N

Figure 4.2: Multi-Accelerator FPGA Edge Platform

4.2.1 FPGA-based Multi-Accelerator Edge Hardware

The DNN accelerators are implemented on the FPGA programmable hardware. The edge node is allowed to re-partition the FPGA during runtime to generate accelerator slots with different sizes to host a heterogeneous set of accelerators. This will cause *slot re-partitioning delay overhead*. Thereafter, each accelerator slot is configured to run the designated DNN accelerator. Such an operation will introduce *accelerator configuration delay overhead*.

4.2.2 Learning-based Multi-Accelerator Management Software

We present the learning-based multi-accelerator management system software that schedules tasks, dynamically reconfigures accelerator slots, and allocates accelerators. The system software is composed of Multi-Queue-Input-Task, DNN Accelerator Library, and Learning-based Scheduling-and-Allocation modules:

Multi-Queue-Input-Task Module: The input acceleration tasks are distributed in the multi-queue module in which there is a separate queue for each acceleration type. The system monitors the workload for each acceleration type and accordingly, considers this distribution during scheduling and accelerator allocation. By separating tasks in multi-queue, characteristics of input workloads for each acceleration type could be easily observed and captured by the system software.

when the input tasks cannot be processed in time, they will accumulate in the task buffer. However, the size of the task queue buffer is always limited in real systems. We set the size of each accelerator queue to 3x the average number of task requests per second. When the task queue is full, the new incoming tasks to this queue will be dropped.

DNN Accelerator Library: We are given a rich library of DNN accelerators for the applications. When a DNN model is trained, the specified NN topology can be deployed with different hardware architectural block types (e.g. dataflow, pipeline) and sizes (e.g. hardware parallelism) of accelerators. And, under the same NN topology, the model can also be trained for different target object types, i.e., dataset, and, as a result, it achieves different inference accuracy, where the inference varies by loading different weight and activation parameters propagated throughout the NN topology.

Learning-based Scheduling-and-Allocation Module: This module makes scheduling decisions based on the observation from the current tasks in queues and the underlying FPGA hardware resources. We apply the learning-based method to allow the system to acquire knowledge and improve the scheduling policy during runtime. We adopt Deep Q-Network (DQN) [87] and deliberately design it to be lightweight so that the learning model is fully contained in a single edge node.. The primary role of this module is to provide a control policy that maximizes system throughput. We consider minimizing the maximum queue utilization as the objective, which reflects on the throughput of processing acceleration tasks.

Sharing accelerators reduces accelerator reconfiguration overhead, and hence we promote *accelerator-sharing policy* in our scheduling to further improve the system throughput. For example, in DNN vision applications, task requirements are typically the accuracy constraint and the target object type. An accelerator with higher accuracy can be downward compatible with those DNN applications with lower inference accuracy. A request to access an AlexNet accelerator with 55.5% accuracy on ImageNet dataset can be processed via SqueezeNet (with 57.5% accuracy) or GoogleNet accelerator (with 67.4% accuracy) on ImageNet dataset.

The *asynchronous-learner* process learns policies from gathered experiences, which are collected during the interaction with the environment. The Asynchronous-Learner is decoupled with the scheduling process, the learning process can learn and update policies asynchronously. Given the limited computing resources of embedded cores on the FPGA edge, we run the asynchronous-learner process on a multi-core CPU edge node as the computing node for learning.

4.3 Reinforcement Learning Model

We propose a Reinforcement Learning model for our learning-based multi-accelerator management system. In our work, the multi-accelerator edge node is considered as the environment. The Scheduling-and-Allocation module plays the role of an agent with learned knowledge in decision-making based on the observation on the environment, e.g., the current tasks in the queues and the current allocated accelerators on the FPGA resources. Our Scheduling-and-Allocation module acts as a mapping unit from the state to an action that consists of task scheduling and accelerator allocation; therefore, it becomes a task-processing unit for the dynamic input workload. Given that only the current states of FPGA edge node and the scheduling action determine the future state and expected rewards, we formulate this problem as a Markov Decision Process (MDP). The Markov Decision Process contains three major components, i.e. **state, action,** and **reward**, which are defined as follows.

4.3.1 State Space

To capture the characteristics of the system and workloads, the state S contains the status information of accelerators and queues on the edge, which is formulated as:

$$S = SS \times SI \times QS \tag{4.1}$$

, where SS denotes the runtime status of accelerator slots, SI determines the size of each accelerator slot, and QS denotes the status of multi-acceleration queues. SS is formally presented as $[ss_1, ss_2, ..., ss_{n_s}]$. Each vector $ss \in SS$ consists of observed parameters for each accelerator slot, which includes: (a) the estimated time to finish, (b) the idle state, and (c) the current accelerator on the slot. Each accelerator is assigned a unique identifier, which is a categorical, not continuous, value. Given that there are n_m supported models of applications on the edge, we convert the parameter to a one-hot vector of size n_m and concatenate it with the rest observed parameters. Therefore, the SS can be formed as a $(n_s \times (3 + n_m))$ matrix. In addition to the parameters for each individual slot, to distinguish the size of the accelerator slots, we adopt a SI vector to indicate the use of FPGA resources. In this work, we provide three kinds of slot sizes, i.e., Large, Medium, and Small slots. There are at most n_s slots, Hence, SI becomes a $(n_s \times 3)$ matrix.

There are up to n_q task queues. The status of each acceleration task queue can be presented as $QS = [qs_1, qs_2, ..., qs_{n_q}]$. The system receives and distributes tasks to individual queues according to their task requirements. Each $qs \in QS$ is composed of observation on the corresponding queue, which includes the queue utilization. As a result, the QS can be formed as a vector of size n_q.

4.3.2 Action Space

The action space A represents the interaction with the task and/or accelerator slots. Since the FPGA resources are reconfigurable, actions are: EXE- execution of a task on an existing accelerator on the slot, SW_CFG- accelerator software configuration on the corresponding FPGA accelerator slot, HW_CFG- FPGA hardware configuration for a new set of accelerator slots, and NOP-no operations. Therefore, we divide the action space A into four categories:

$$A = EXE \cup SW\_CFG \cup HW\_CFG \cup NOP \tag{4.2}$$

In the EXE category, there are n_q individual task queues held for scheduling and n_s accelerator slots serving the computation. The execution action category is formed as a $(n_s \times n_q)$ matrix. For a certain $exe_{i,j} \in EXE$, it indicates the action value for the execution of the j-th queue's task on the i-th accelerator slot. The scheduler can choose to first carry out software configuration on the slot rather than task execution. Thus, for SW_CFG, the configuration action can be modeled as a $(n_s \times n_m)$ matrix. For a certain $sw\_cfg_{i,j} \in SW\_CFG$, it indicates the action value for the configuration of the i-th accelerator slot with the j-th DNN accelerator. In addition to configuring a single accelerator slot, the FPGA resource is also allowed to re-partition during runtime. There are three slot sizes in our setup. HW_CFG becomes a $(n_s \times 3)$ matrix. For a certain $hw\_cfg_i \in HW\_CFG$, it indicates the action value for reconfiguring the hardware resources, i.e. FPGA fabric, into $i - th$ accelerator slot set. And, lastly, NOP is simply one action for doing nothing on the edge. These actions are discrete and we model them in vectors to indicate each specific action; therefore, the size of the actions A becomes $[n_s \times (n_q + n_m + 3) + 1]$ in total.

However, in a given state s, not all actions in A are necessarily valid, e.g., performing actions of execution on a slot is not valid when the slot is already in use. Only a subset of actions is valid in a state s, which can be denoted as $A_{valid}(s) \subseteq A$. Since the unconstrained action space A cannot be directly used for scheduling and allocation decisions, we add the action mask AM to "filter" out the invalid actions. Thus,

$$A_{valid}(s) = A \cap AM(s) \tag{4.3}$$

For simplicity, the rules of AM are as follows: (1) If the slot is not idle, any actions to the slot are invalid, (2) If the existing accelerator on the slot cannot fulfill a task requirement, the execution action for such a task on this accelerator slot is invalid, (3) If the task queue is empty, any action related to the task queue is invalid.

4.3.3 Reward

The reward is feedback from interaction with the environment. The reward $r(s, a)$ can be constructed as a value function of state $s \in S$ and action $a \in A$, which is generated in each scheduling decision along with the state transition, $s \xrightarrow{a, r(s,a)} s'$, where s' denotes the next state. As described in the action space, A, the operations onto the environment can be divided into four categories, EXE, SW_CFG, HW_CFG, NOP. To guide the agent in maximizing the objective function, we give different metrics of rewards for each action category.

The goal is to maximize the system throughput in processing acceleration tasks, which we model as a min-max optimization problem on queue utilization. We design the reward to guide the agent to consider the system's overall queue utilization. When a queue is full and can easily lead to overflow, we give a penalty, a negative reward of -1, to discourage such "high-load" queue utilization. Otherwise, the reward is designed to have the main, R_{base}, and auxiliary reward, R_{Q_i}. The reward of execution is, then, composed of two cases, which are shown in the following:

$$R_{exe}(s,a) = \begin{cases} -1, if\ any\ Q\ is\ FULL \\ \alpha \cdot R_{base}(s) + \beta \cdot R_Q(s,a), otherwise \end{cases} \tag{4.4}$$

The R_{base} is designed to be state-dependent and aligns with the system's primary objective of minimizing the maximum queue utilization. That is, the less maximum queue utilization the system achieves, the higher reward the state transition generates. The reward range of R_{base} is [0, α]. Thus,

$$R_{base}(s) = (1 - MAX(Util(Q_i)|\forall i \in QS)) \tag{4.5}$$

We added R_Q to direct the agent to take actions on specific queues. A queue in high utilization should be prioritized to take action, which can substantially reflect on decreasing the maximum queue utilization and reduce risks of task drops. The R_Q is action-dependent, where a positive reward is given if the queue, Q_i, for actions is in high utilization. The reward range of R_{Q_i} is [0, β]. Then, it can be formed in the following:

$$R_{Q_i}(s,a) = \begin{cases} 1, if\ Q_i\ \in MAX(Util(QS)) \\ 0, otherwise \end{cases} \tag{4.6}$$

In other categories of actions, i.e., accelerator software and hardware (i.e. re-partitioning hardware resources) configuration, and no-operation, we use penalties (negative rewards) for such "non-execution" actions, noted as P_{SW}, P_{HW}, and P_{NOP}. P_{SW} is the penalty for making accelerator software configuration. P_{HW} is the penalty for making FPGA hardware configuration, that re-partitions the FPGA into a new set of accelerator slots. Both the amount of penalty P_{SW} and P_{HW} are based on the delay overhead over an accelerator slot and the FPGA platform, and normalized into a range of [-2, -1]. P_{NOP} is the penalty for doing no-op in scheduling and leaving tasks accumulated in the queues, which is defined as the negated value of the maximum utilization of tasks in queues. The P_{NOP} is normalized into a range of [-1, 0], where 0 in penalty means no tasks (0% utilization) in queues.

$$P_{NOP}(s) = -1 \cdot MAX(Util(Q_i)|\forall i \in QS) \tag{4.7}$$

Note that the SW_CFG actions will follow any HW_CFG action in order to initialize new accelerators.

Therefore, the reward function $r(s, a)$ for a given state s and a valid action $a \in A_{valid}(s)$ is denoted as follows:

$$r(s,a) = \begin{cases} R_{EXE}, & a \in EXE \\ P_{SW}, & a \in SW\_CFG \\ P_{HW}, & a \in HW\_CFG \\ P_{NOP}, & a \equiv NOP \end{cases} \tag{4.8}$$

4.3.4 Problem Formulation

According to the above reward functions, we can further obtain the expected return G_t at the t-th decision epoch, which is defined as the cumulative sum of discounted future rewards from the current time step over the long run.

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} = \sum_{k=0}^{\infty} \gamma^k r(s_k, a_k) \tag{4.9}$$

, where $\gamma, 0 < \gamma \leq 1$, is the discount rate, and R_t denotes the reward at the t-th decision epoch. In the learning model, we **maximize the expected return** G_t, which is equivalent to maximizing the cumulative rewards at the t-th step; thus, minimizing the maximum queue utilization in the long run.

4.4 Asynchronous Learning Architecture for Multi-Accelerator Management

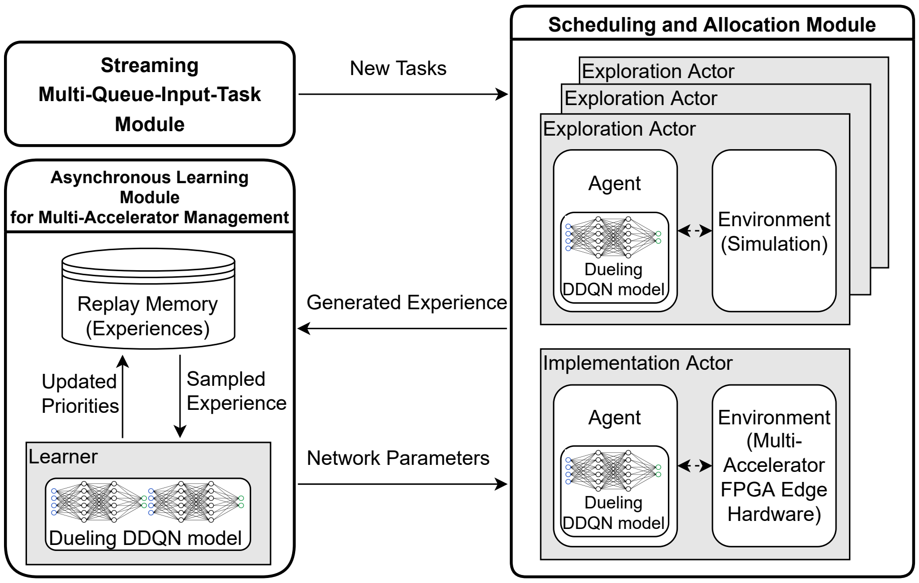

Figure 4.3: Asynchronous Learning Architecture for Multi-Accelerator Management

We apply Deep Reinforcement Learning methods to solve our MDP problem. We adopt the commonly-used Deep Q-Network (DQN) algorithm [87] due to its low computational demand and enhanced data efficiency by recycling valuable experiences, which is favorable for resource-constrained environments. Since our action value comprises not only the value of states, i.e., the current queue and system status (action-independent), but also the effect of actions, i.e., execution, configuration, re-partitioning, and no-op (action-dependent), we incorporate the Dueling Double DQN (Dueling DDQN) model [134, 139] to consider both types of action values in our system. Also, Prioritized Experience Replay [116] is applied in the learning algorithm to focus its efforts on newly discovered action spaces.

Table 4.1: DNN Accelerator Library: Inference Latency (ms) for DNN models under different accelerator slot sizes

	Classification								
	General						Car Type	Human Face	
Accelerator Slot Size	AlexNet (M1, 55.5%)	SqueezeNet (M2, 57.5%)	ZynqNet (M3, 63.0%)	GoogleNet (M4, 67.4%)	ResNet18 (M5, 69.1%)	ResNet50 (M6, 72.9%)	GoogleNet (M7, 91.2%)	CaffeNet (M8, 54.3%)	GoogleNet (M9, 56.2%)
Large	8.0	7.8	7.9	21.9	17.5	55.8	21.9	8.0	21.9
Medium	11.9	14.2	13.4	41.9	44.1	180.5	41.9	11.9	41.9
Small	29.3	27.8	29.5	85.4	95.2	308.2	85.4	29.3	85.4

	Detection		
	General		License Plate
Accelerator Slot Size	MobileNetSSD (M10, 72.7%)	VGGSSD (M11, 78.2%)	MobileNetSSD (M12, 72.7%)
Large	26.0	752.1	26.0
Medium	57.5	1009.0	57.5
Small	×	×	×

Inspired by the asynchronous methods in [33, 19], we decouple the learning (learner) and acting (actor) processes from the deep reinforcement learning method. Then, we use actors to select actions (i.e. scheduling decisions) and collect experience along with the learner to learn policies from sampling gathered experience. When the learning and acting processes run asynchronously and concurrently, the actors can continuously step through the environment and gather experience from up-to-date workloads, rather than the outdated ones. Asynchronous learning is suitable for our near real-time scenario in processing a stream of requests. Thus, in our framework, our asynchronous learning architecture includes the learner, actors, and a shared prioritized replay memory. We run actors in the Scheduling-and-Allocation module and the learner in the Asynchronous-Learning module. (See Figure 4.3)

On the actor side, there are two types of actors, named *Implementation* and *Exploration* actors. Both *Exploration* and *Implementation* actors make decisions based on current learned knowledge, task loads, and resource utilization. The *Implementation* actor interacts with the multi-accelerator edge hardware and makes the “greedy” action, which the actor believes to be optimal based on its existing knowledge, to maximize the efficiency of the system. On the other hand, the *Exploration* actor interacts with a simulated environment, designed to simulate task processing and acceleration activities. In the *Exploration* actor, a static time model (measured and averaged time on the DNN accelerator library) is used in the simulated environment to act as the accelerator execution/configuration. The *Exploration* actors explore and may make random actions on the simulated environment. So actors collect different solution spaces of experiences for the learner to improve the understanding of the environment. In order to capture the input, the new tasks from the Streaming Multi-Queue-Input-Task module are replicated to the *Exploration* actors. The interface between actors and the learner is asynchronous, allowing modules to run concurrently. The action spaces are exploited and explored in parallel to provide a more time-efficient decision-making process and, meanwhile, to explore action spaces to allow the learning module to asynchronously improve decisions over time.

In our work, with the constrained computing resources of embedded cores on the FPGA edge, we can run a limited number of *Exploration* actors on embedded CPU cores on an FPGA device. We run the asynchronous-learning module, i.e., the learner process, on the multi-core CPU platform at the edge.

Algorithm 4 Learning-Based Dynamic Scheduling with Asynchronous Learning

```
1: Create one learner, $Learner$, one implementation actor, $Actor_{imp}$, and multiple
   exploration actors, $Actor_{exp}$
2:
3: /* Run in parallel using multi-threading */
4: STREAM_TASK()                ▷ streaming tasks to all $Actor$s' Environment
5: $Actor_{imp}$.RUN_THREAD()
6: ASYNCHRONOUS_LEARNING()
7:
8: function ASYNCHRONOUS_LEARNING
9:     while sys_terminate != True or not sys_qos_met do
10:        $Actor_{exp}$.RUN_THREAD()
11:        $Learner$.RUN_THREAD()
```

A summary of the system flow for the asynchronous learning method is shown in Algorithm 4. In Line 4-6, these procedures are created in multiple threads; thus, the system learns and exploit the information asynchronously. When the system starts, the multi-queue-input-task module starts to collect tasks into multi-queues and sends them to actors. Then, the implementation actor $Actor_{imp}$ makes the optimal scheduling decision based on the current knowledge whenever any slot becomes available and there are tasks on hold in the queues. Meanwhile, the asynchronous learning module launches threads for multiple exploration actors $Actor_{exp}$ and the learner *Learner* (Line 9-11 in Algorithm 4). Note that the multi-queue-input-task module replicates tasks for the simulation environment. The $Actor_{exp}$ explores the information and gathers experiences to the replay memory *Replay* for *Learner* to learn asynchronously. Therefore, the system provides efficient scheduling decisions on the edge computing resources and asynchronously improves decisions through the asynchronous learning process.

Algorithm 5 shows the actor flow of interacting with the multi-accelerator edge environment. There are mainly two scenarios for the actor to take actions. When the system is busy, i.e. all slots are either executing acceleration or configuring slots for the subsequent acceleration, the actor takes NOP operation, and then the actor process is blocked waiting for the environment until any slot becomes available (Line 8 - 10 in Algorithm 5). On the other hand, depending on the actor type, the actor process chooses to make the optimal or random decision with the $\epsilon - greedy$ policy (Line 13 - 17 in Algorithm 5). The experiences of interactions with the environment will be collected and stored in the shared replay memory buffer. The knowledge of actors is asynchronously updated in every interval $Interval_{sync}$.

Algorithm 6 shows how the learner process learns insights and updates knowledge asynchronously. Given that we are applying the D3QN algorithm in our model, there are two networks, i.e. online and target network, in the learner process. The learner process randomly samples a batch of experiences from the replay memory, which helps break the

Algorithm 5 Actor Process of Interacting with the Multi-accelerator Edge Environment

```
    Input actor_type                ▷ Actor Type is either Implementation or Exploration
 1: /* Assume global shared Replay */
 2:
 3: Initialize local environment, Env ▷ Reset states $s$, flush task queue and accelerator slots
 4: Obtain online network $Q(s, a; \theta)$ with Learner's latest weights $\theta^+$
 5:
 6: for $t = 1$ to $\infty$ do
 7:     if Env.no_idle_slot then
 8:         $a_t = NOP$
 9:         $r = 0$
10:         $s_{t+1} \longleftarrow$ Env.WAIT_FOR_IDLE_SLOTS()
11:         Replay.ADD($(s_t, a_t, r, s_{t+1})$)
12:     else
13:         if actor_type == Implementation then
14:             Select action $a_t = argmax_a Q(s_t, a; \theta)$ // Exploitation
15:         else
16:             Select action $a_t$ with decayed $\epsilon$-greedy policy based on $Q(s_t, a; \theta)$
17:         $(s_{t+1}, r) \longleftarrow$ Env.STEP($a_t$)
18:         Replay.ADD($(s_t, a_t, r, s_{t+1})$)
19:
20:         if $t$ % $Interval_{sync}$ == 0 then
21:             Update online network weight $\theta$ from Learner's latest weights $\theta^+$
```

Algorithm 6 Learner Process for Asynchronous Learning

```
1: /* Assume global shared Replay */
2: Initialize online network $Q(s, a; \theta)$ with weights $\theta$
3: Initialize target network $Q(s, a; \theta^-)$ with weights $\theta^- = \theta$
4: for $i = 1$ to $\infty$ do
5:     Replay.SAMPLE() // Sample mini-batch of transition $(s, a, r, s')$
6:     Compute loss based on the learning rule, e.g. D3QN, for back-propagation wrt online
   network $Q(s, a; \theta)$
7:     Update online network weights $\theta$
8:     if $i$ % $Interval_{update}$ == 0 then
9:         Update target network $Q(s, a; \theta^-)$ with weights $\theta^- = \theta$
```

temporal correlation in the sequence of experiences (Line 5). It, then, calculates the temporal difference (TD) error, which is the difference between the current Q-value estimate and the target Q-value (Line 6). Lastly, update the Q-value of the selected action using gradient descent to minimize the TD error for the online network (Line 7). The learner periodically updates the parameters of the target Q-network with the current Q-network parameters, which is designed to help stabilize the learning process (Line 8 - 9).

4.5 Evaluation

4.5.1 Experimental Setup

In this subsection, we describe the platform setup, multi-accelerator design profiles, the DNN accelerator library, and the input workload characteristics to evaluate our framework. Finally, we address the setup for the learning model.

Platform Setup: In our experiments, eight end devices, Raspberry Pi 4B, periodically capture images from their environment and send requests for DNN-based vision acceleration to the FPGA edge. We consider image classification and object detection applications. These end devices and the edge communicate through a dedicated wireless router access point. The edge is a multi-accelerator FPGA edge device, AMD Xilinx ZCU102, which is also connected to a single multi-core Intel i5 CPU platform as the computing node for learning and processing other applications.

DNN Accelerator Library: we use CHaiDNN [9] to implement and generate the DNN accelerators. We use HLS pragmas and modify the source codes to explore various parallelism and resource utilization. Table 4.1 lists our DNN accelerator library in ascending order of accuracy for DNN models. The inference latency increases when the accelerator is smaller. MobileNetSSD and VGGSSD can only run with the large or medium DNN accelerator. All models run under Xilinx 6-bit quantization with CHaiDNN. We modified the software stack of CHaiDNN to move its online parameter quantization process to be executed offline.

Dataset and Input Workload Characteristics: To facilitate the training and evaluation of our proposed framework, a substantial stream of input images is needed, encompassing a combo of different working cases. We create a collection of video dataset generated by each end device from Multiple Object Tracking (MOT) Benchmark[84], Driving Event Camera Dataset [107] and the VIRAT Video Dataset[127] to simulate various use cases such as a surveillance camera at parking lots or a smart camera at a traffic intersection in smart spaces. We generate eight data sets by randomizing, mixing and rotating existing video clips. In our case, each training data set has eight streams of tasks, coming from different video data sets, timelines, and application configurations. To generate training/testing data, we first separate the selected video clips into end devices, simulating cameras capturing real-time videos.

In our experiments, some end devices run an on-device object detection algorithm by Q-Engineering[103], and then, send the detected objects to the edge for classification using hardware acceleration. Some will directly send the images for detection and classification processing at the FPGA edge. Detection tasks generate additional classification acceleration tasks from detected objects in the input queue. In this work, we consider three categories of classification tasks, i.e. Car, Person, and General, and two categories of detection tasks, i.e. General and License Plate. We have organized our input acceleration tasks into four sets as follows:

Set A: requests three types (car/ person/ general) of classification tasks with the minimum accuracy requirement (91%/ 54%-56%/ 55%-69%), accordingly. The accumulated task ratio for each category is 1:1:2. The total accelerator accesses range from 40-100 per second.

Set B: requests two types (person/general) of classification tasks with the same minimum accuracy requirement as Set A. The accumulated task ratio for each category is around 1:1. The total accelerator accesses range from 40-100 per second.

Set C: involves a combination of requests for classification and detection. The accumulated task ratio for classification and detection is around 3:2. The total accelerator accesses range from 50-90 per second.

Set D: involves even more detection and fewer classification tasks. The accumulated task ratio for classification and detection becomes 2:3. The accelerator accesses range 50-90 per second.

All model learning is done for two hours of deployment time and testing is evaluated during the last 10 minutes using a collection of video datasets from the real world in the above. We evaluate the performance of our proposed framework with the two state-of-the-art heuristic and learning-based approaches. (1) $SCYLLA$: [48] is a greedy heuristic algorithm for task scheduling and resource allocation, which provides out-of-order scheduling and dynamic accelerator allocation. (2) $REINFORCE$: We adopt the learning-based approach [120] to provide scheduling and accelerator allocation on the multi-accelerator FPGA edge. It includes out-of-order access to accelerators but lacks dynamic slot configuration, multi-queue scheme, accelerator sharing, and asynchronous learning.

4.5.2 Experimental Results

Edge Resource Utilization for the learning model

Our framework allows having multiple *Exploration* actors for exploration in action spaces. Figure 4.4 shows the average input and system throughput in set A_2 over a two-hour duration when using two to sixteen *Exploration* actors for learning. Each number in the

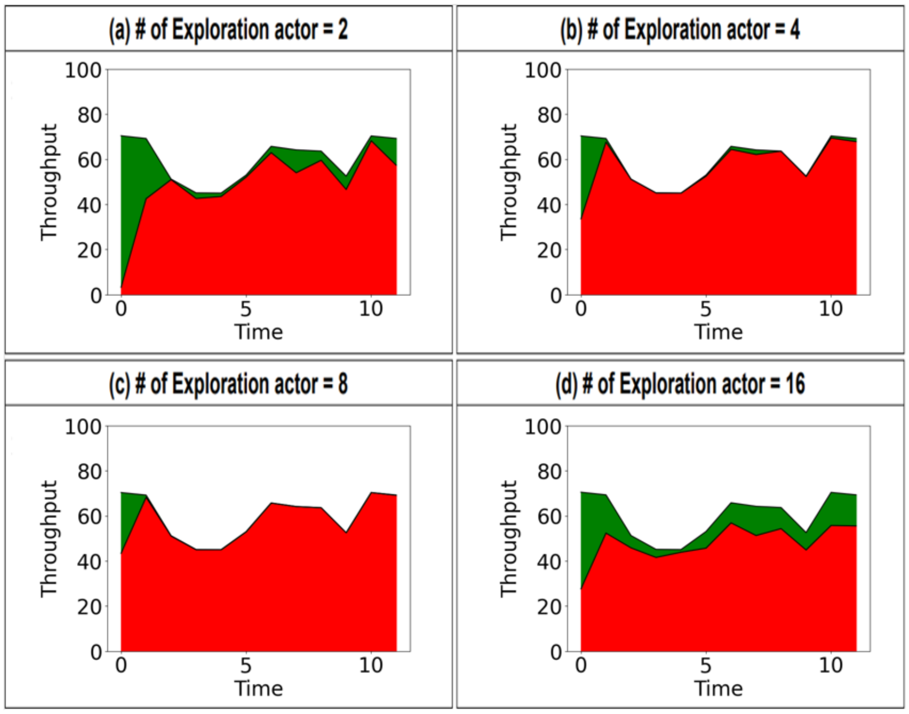

Figure 4.4: Scheduling Performance over time with two to sixteen *Exploration* actors (Experiment: A_2). Each number in the x-axis is the average throughput in every 10-minute interval.

x-axis is the average value in every 10-minute interval. The data in green color represents the average input throughput; the red one is the system throughput of our framework. In the early stage, we see the average throughput mismatch significantly. This is because the system is still exploring action spaces and can only discover sub-optimal policies; thus, scheduling policies change frequently. Once the action spaces have been sufficiently explored, the optimal policy will be discovered and learned. In eight actors, we observe the scheduling policy improves quickly and has reached the maximized system throughput. When using two actors, fluctuation remains over time, and when using sixteen actors, due to the limited computing resources on embedded cores, actors suffer in high turnaround time, hence, the system learns an inferior policy. We observed the same behavior in other experiments, and the eight actors outperformed the others. Therefore, we use eight *Exploration* actors for the target FPGA edge in our experiments. Note that the percentage of CPU consumed by each *Exploration* actor is around 4.3% on a core, which still allows our quad-core ZCU102 edge device to process other routines/tasks during runtime. During the training phase, the learner process utilizes around 27.5% per CPU core on the i5 processor. We observe that the learning often converges within one hour, and the proposed model is lightweight and efficient enough to run fully on the resource-constrained edge.

Table 4.2: Average Throughput (accelerator access per second) and Task Drop Rate Comparison

	Avg. Throughput/Task Drop Rate		
Set	SCYLLA[48]	REINFORCE[120]	Our
A_1 (40)	30.7/23.3%	28.4/29.0%	40.0/0.0%
A_2 (60)	32.5/45.8%	36.3/39.5%	59.8/0.3%
A_3 (80)	33.7/57.9%	34.7/56.6%	64.8/19.0%
A_4 (100)	32.3/67.7%	32.8/67.2%	63.8/36.2%
B_1 (40)	30.9/22.8%	23.5/41.3%	39.9/0.3%
B_2 (60)	34.3/42.8%	36.7/38.8%	59.9/0.2%
B_3 (80)	32.2/59.8%	36.6/54.3%	66.4/17.0%
B_4 (100)	34.7/65.3%	40.7/59.3%	65.2/34.8%
C_1 (50)	41.2/17.6%	35.9/28.2%	50.0/0.0%
C_2 (70)	46.7/33.2%	44.8/36.0%	69.7/0.4%
C_3 (90)	42.1/53.2%	47.4/47.3%	65.1/27.7%
D_1 (50)	34.1/31.8%	31.3/37.4%	49.3/1.4%
D_2 (70)	34.8/50.3%	40.5/42.1%	58.5/16.4%
D_3 (90)	41.0/54.4%	42.8/52.4%	61.3/31.9%

Evaluation on Average Throughput and Task Drop Rate

To evaluate the system performance, we measured the number of tasks that obtain access to accelerators and are processed on the edge. We use the rate of average accelerator accesses per second as the average throughput, which is formulated as the number of accelerator accesses divided by the testing time. The task drop rate due to overflow is also reported.

Table 4.2 summarizes the average throughput and task drop rate measured during the last 10 minutes, as the testing phase, out of the total 2-hour deployment time and we compare *Our* method against the state-of-art heuristic and learning-based approaches.

In most of our experiments, $SCYLLA$ [48] saturated at a throughput of around 34 accelerator access per second. Due to the greedy scheme, this two-step heuristic method achieves the approximate optimal performance. Instead, Our can achieve up to 2.1x speedup in throughput and 45.5% lower in the task drop rate.

Both $REINFORCE$ [120] and Our adopt the reinforcement learning approach. Our obtains the multi-queue-input-task module, accelerator-sharing policy, and the exploration of slot configurations; as a result, we achieve up to 1.9x higher in throughput compared to $REINFORCE$. We obtain 19.7% to 41.0% lower in the task drop rates. $REINFORCE$ is only capable of scheduling on pre-set static resources, which, in a multi-accelerator platform, are fixed slot numbers and not capable of exploring configurations to optimize during learning. Thus, the computing resource usage could be unbalanced in workloads between each slot, which prevents tasks from being allocated on the shared resource more efficiently. Therefore, in $REINFORCE$, despite applying the learning approach, the system can only achieve a slight throughput improvement compared to the heuristic method. Certain experiments, like $A\_1, B\_1, C\_1/2$, and D_1, the throughput of $REINFORCE$ is even lower than $SCYLLA$. Note that, the $REINFORCE$ doesn't have multiple *Exploration* actors for the asynchronous learning and requires exploring the action spaces directly in the real environment during the deployment phase. Although the exploration space has been deduced to optimize on pre-set static resources, in experiments, $REINFORCE$ still requires a longer time (one to two hours) to converge compared to Our (with less than an hour) Our can adapt to the dynamic environment faster and more efficiently.

Our achieves the best average throughput and lower task drop rates compared to $SCYLLA$ and $REINFORCE$. In most of our experiments, e.g. $A\_1/2, B\_1/2, C\_1/2$ and D_1, the system reaches the ideal maximum throughput according to the input. For higher input throughput, e.g. experiment $A\_3/4, B\_3/4$, C_3, and $D\_2/3$, the system can't process the acceleration tasks literally due to the limited computing resources to schedule for the heterogeneity and high loads of acceleration tasks; however, *Our* still surpasses them. This demonstrates the efficiency of the proposed model and the fact that the joint scheduling and accelerator allocation policy has learned to accommodate the dynamic workload.

Insights of Scheduling and Allocation Results

We now provide insights about the difference in scheduling and allocation acquired from *SCYLLA*/*REINFORCE*/*Ours* methods.

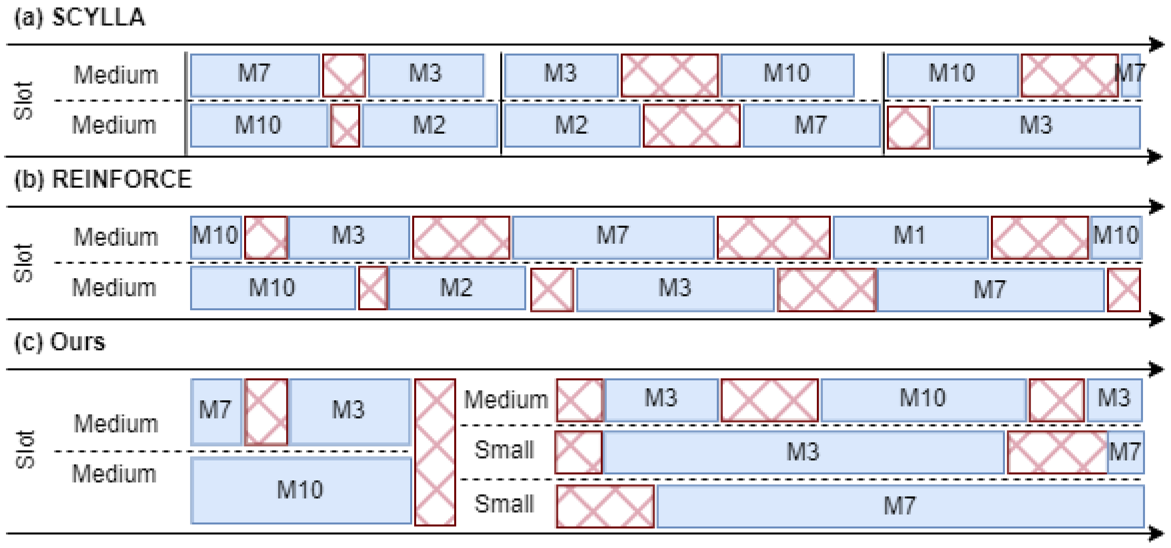

Figure 4.5: Scheduling Sequence Snapshot (The blue painted/cross-hatch/blank area denotes batched execution (of which accelerator)/accelerator configuration/idle state.)

Figure 4.5 shows the representative of a partial real scheduling sequence on experiment D_1, from which we can see the difference in scheduling policies. In experiment D_1, applications are mixed in categories and classification/detection tasks.

SCYLLA tends to process batched tasks using larger accelerators, which provide lower execution time. As a result, the number of accelerators running in parallel decreases. This leads to less parallelism and potentially more tasks waiting in serial. In experiment D_1, even with the flexibility of exploring slot configurations, *SCYLLA* greatly uses two medium accelerator slots. In Figure 4.5:(a), $SCYLLA$ schedules batches of tasks in waiting queues every a while when the previous schedule of tasks among all slots has finished. Since there is no "memory" of the previous schedule in the heuristic algorithm, the algorithm is unlikely to further optimize and improve the throughput. In $REINFORCE$, the system is pre-set and fixed on two medium accelerator slots according to the minimum resource requirement for detection tasks. The $REINFORCE$ observes waiting tasks in the waiting matrix, i.e. the observation part, and dynamically schedules tasks for the existing accelerator on slots. However, tasks that are not capable of processing via the existing accelerators due to application requirements start accumulating and eventually fill up the observation set and drop tasks, which, as a result, forces the system to configure accelerator slots for processing other tasks.

When Our applies the proposed multi-queue-input-task module and accelerator-sharing mechanism, these improvements allow the system to globally observe the task workloads in queues, process tasks in dynamic batches, and, thus, balance the load on both queues and accelerators/slots. In addition, Our is optimized to acquire various accelerator slots. In Figure 4.5:(c), the system starts with processing tasks on two medium accelerator slots and changes to 1 medium plus 2 small accelerator slots, where despite that, on the small slot, each task takes a relatively prolonged execution time compared to the medium one, but the system can run more (different types of) tasks in parallel. The system also learns to initialize the next accelerator for predicted future workload and, as a result, it hinders the accelerator configuration delay overhead and increases the efficiency of computing resources, i.e. accelerator slots.

4.6 Related Works

There have been related works focusing on using learning-based approaches to dynamically offload tasks to assigned compute resources, e.g. pre-set IPs (Accelerators), CPU-based/VM-based nearby edge or cloud nodes.

Sheng et al. [120] introduces a DRL-based solution for optimizing both resource allocation and task scheduling, with a focus on maximizing Quality of Experience (QoE) value over the long term while considering expected delay requirements. The method determines both the scheduling order and the assignment of tasks to specific Virtual Machines (VMs). Although VMs can treated as accelerators, the authors consider no functional difference in accelerators and no assignment restriction of tasks to accelerators which means tasks can be freely assigned to any of the VMs. However, in an accelerator-rich edge, tasks may not be able to be processed on certain deployed accelerators. Because such a deployed hardware accelerator, which isn't as general-purposed as CPUs, may not fulfill the task requirement, i.e. network topologies, and accuracy requirements. The proposed model lacks the consideration of heterogeneity of computing resource functionalities and can't restrict the task assignment to certain computing resources. Also, it lacks the consideration of changes in accelerators in terms of sizes/functions. The amount of VMs is fixed where their computational capacity and functionality are pre-configured and unchanged. However, on a shared edge featuring FPGA-based accelerators, the computing resource can be dynamically reconfigured into different sizes/types of accelerators to maximize resource utilization based on the variation of system workloads. The proposed model lacks the flexibility of changing and re-allocating compute resources. Besides, while maximizing accumulated QoS, i.e. the ratio of the response time and expected delay, as the objective, the task drop rate is also important. There is no control of task drops and fairness to multiple end devices. Despite the improved task success ratio reported in the experiment, the proposed method lacks the modeling for task drops and can't extend to support the (weighted) fairness for processing tasks to control/improve drops due to the designed waiting ”set” mechanism. Tuli et al. [131] explore leveraging temporal patterns for scheduling in a hybrid Edge-Cloud setup. The authors use coarser-grained scheduling to reduce the frequency of scheduling, which occurs at consistent 5-minute intervals. The action involves assigning tasks (cloudlets) to Virtual Machines using a bijection approach that establishes a one-to-one mapping between tasks and VMs. However, in a resource-constrained edge environment, the computing resource may not be able to serve all hosts and require the task/host to wait in the queue/unscheduled. Or allowing the hosts to process on multiple computing resources in parral. The proposed model cannot handle such restrictions or allocations. Besides, their proposed A3C-based approach in the learning engine requires quite an amount of actor agents for training and collecting experience, which isn’t suitable for a resource-constrained edge compared to ”resource-unlimited” clouds. Iranfar et al. [44] presents a reinforcement-learning-based method for efficient resource management. It addresses real-time streaming complexity by optimizing system performance using heterogeneous hardware acceleration and general-purpose cores in MP-SoCs. The approach involves dynamically assigning different video streaming tasks to these deployed components while adjusting their frequencies to meet diverse task requirements.

The aforementioned related works focus on establishing a mapping between tasks and certain computing resources. The amount of deployed IP configuration guarantees streams/tasks to pair with any of the IPs without the contention on computing resources. Overall, these previous studies concentrate on task scheduling within specific setups and deployment. In [131], the bijection mechanism makes it impossible to apply to a computing resource that could be dynamically allocated and shared among tasks. As a result, these approaches hinder the potential for optimizing the allocation of shared computing resources during task scheduling. However, this is essential and not uncommon in a resource-constrained edge computing system.

Other related works focus on adapting compute resource allocation, which enables resource sharing among tasks. In [95], Pan et al. propose a deep reinforcement learning (DRL) framework to dynamically allocate bandwidth in routers featuring multiple queues. This framework employs DRL algorithms to train a bandwidth controller, distributing bandwidth weights across queues based on real-time queue lengths. It controls the proportion of bandwidth for multiple splits. The results demonstrate that the trained controllers yield notably lower average delays and packet loss rates compared to rule-based policies. The problem of managing the bandwidth controller has limitations in handling homogeneous tasks and accelerators, which has no concerns about the compatibility of task-accelerator pairs. Raeis et al. [104] propose a reinforcement learning-based service-rate controller that provides probabilistic upper bounds on system end-to-end delay while avoiding excessive service resource utilization within multi-server queueing systems. However, the model is restricted in deployment same as [95].

Compared to bandwidth control, Xiong et al. [141] present a resource allocation policy for enhancing resource utilization efficiency in a mobile edge computing (MEC) system. In the MEC system, IoT device-uploaded jobs are queued, awaiting scheduling. The approach employs deep reinforcement learning to train the system and minimize the weighted sum of average job completion times and average requested resource count over the long term.

Nevertheless, these works include task scheduling and resource allocation other than the resource mapping problems. This adaptation is tailored to processing tasks on general-purpose CPUs or similar resources at the edge. Existing works lack consideration in changes of VMs/IPs in terms of sizes/functions. There is an assumption that the streams/tasks can always allocate an available IP for acceleration. There is no system software support that considers both task scheduling and resource allocation on FPGA multi-accelerator architectures, where, in addition to the functions/sizes of tasks and accelerators, runtime configuration overheads for re-allocating accelerators for diverse computing resource types should also be considered.

4.7 Conclusion

In this chapter, we present a learning-based multi-accelerator management framework that provides efficient joint task scheduling and accelerator allocation. Our proposed asynchronous learning architecture allows the system to consistently make scheduling decisions in real time and asynchronously improve the scheduling policy. Under changing input workloads in the environment, the result significantly outperforms related scheduling schemes in terms of average throughput and task drop rate.

Chapter 5

Conclusions and Future Directions

5.1 Conclusion

Edge computing is becoming increasingly important in the era of big-data applications with high demand for processing power. Unlike cloud computing, which involves processing data in a centralized data center, edge computing processes data locally, closer to the source of data. This is particularly useful for real-time processing and immediate decision-making, as it reduces latency and network congestion. As the demand for computing power continues to grow and data processing requirements become more diverse, heterogeneous architectures and hardware accelerators play a key role in accommodating the demanding processing powers of resource-constrained edge devices. FPGAs offer the advantage of implementing various independent accelerators, enabling a single FPGA to process various applications simultaneously. However, due to the limited amount of resources on FPGAs, it is important to share FPGA resources among different applications to avoid under-utilization of resources. When multiple end devices send DNN applications to the FPGA edge for hardware acceleration, we need system software that can provide efficient task scheduling and resource allocation on the shared FPGA edge.

In this dissertation, we discuss the limitations in the related task scheduling and resource management approaches for the FPGA-based multi-accelerator edge system. We realize the major flaws and investigate optimization opportunities.

In Chapter 2, we present the DNN accelerator sharing service on the edge system. The task scheduling and accelerator allocation problem is modeled into an optimization problem as Mixed Integer Linear Programming and solved using mathematical programming solvers. The experimental results show an up to 2.20x performance gain and improve the utilization by reducing up to 27% of DNN library usage while still meeting the requests' requirement and resource constraints.

In Chapter 3, we present a two-phase systematic approach that not only exploits the characteristics of applications in patterns and optimally balances the concurrent execution on accelerators but also employs a mixed offline/online multi-queue scheduling method to optimize responsiveness by reducing response time and minimizing task drops and accommodate noises from various network delays for consistent IoT DNN workloads. With this two-phase approach, the edge system can significantly improve the responsiveness of average response time and task drops in serving multiple end devices where the task drops are even reduced to zero.

In Chapter 4, we present a knowledge-based multi-accelerator management framework that learns the insights to optimize task scheduling and accelerator allocation. This framework asynchronously learns the scheduling and allocation policy, dynamically partitioning shared resources to optimize system performance through continuous interaction with the edge. The experimental results show improved throughput as well as task drops, compared to related learning-based and heuristic approaches.

5.2 Directions for Future Work

In this dissertation, we present a systematic solution to optimize responsiveness on the edge for consistent workloads, leveraging regularity while accommodating fluctuations. Additionally, for dynamic and event-driven workloads, we introduce a knowledge-based solution, i.e. the learning-based multi-accelerator management framework. This framework learns to efficiently manage tasks and computing resources. Finally, we outline potential opportunities and directions for future development.

5.2.1 Efficiency and Adaptability for the mixed offline/online approach

The proposed mixed offline/online approach is well-suited to consistent input workloads while accommodating noises and uncertainty through the adaptive online scheduler. Future improvements may focus on enhancing methods, e.g. machine learning algorithms, to better explore and exploit regularities in dynamic workloads. This could involve advanced analytics for identifying hidden patterns, leading to more effective offline model training.

Additionally, researchers may explore the development of adaptive hybrid models that dynamically adjust the balance between offline and online components based on the evolving characteristics of the input workloads. This adaptability could enhance the system's performance across a broader range of scenarios.

5.2.2 Reliability and Resilience in the learning-based framework

In addressing noisy and event-based input workloads, the learning-based framework's future evolution could involve integrating robustness mechanisms. While the framework was initially designed for task fairness, there is potential for critical tasks with strict deadlines to benefit from prioritization mechanisms. This approach enhances system reliability and robustness, particularly in scenarios with diverse and unpredictable task characteristics.

Furthermore, the framework's future development may focus on improving adaptability and resilience. This could include exploring transfer learning techniques for leveraging knowledge across different workloads, continual learning mechanisms for staying current with evolving requirements, and investigating meta-reinforcement learning for efficient policy adaptation.

Bibliography

[1] M. Aazam and E.-N. Huh. Fog computing micro datacenter based dynamic resource estimation and pricing model for iot. In 2015 IEEE 29th international conference on advanced information networking and applications, pages 687–694. IEEE, 2015.

[2] AMD Xilinx Vitis HLS. Avaialble: `https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html/`.

[3] R. Ayachi, Y. Said, and A. Ben Abdelali. Optimizing neural networks for efficient fpga implementation: A survey. Archives of Computational Methods in Engineering, pages 1–11, 2021.

[4] E. Baek, D. Kwon, and J. Kim. A multi-neural network acceleration architecture. In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), pages 940–953. IEEE, 2020.

[5] M. V. Barbera, S. Kosta, A. Mei, and J. Stefa. To offload or not to offload? the bandwidth and energy costs of mobile cloud computing. In 2013 Proceedings Ieee Infocom, pages 1285–1293. IEEE, 2013.

[6] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser, and K. Vissers. Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks. ACM Transactions on Reconfigurable Technology and Systems (TRETS), 11(3):1–23, 2018.

[7] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. ACM Transactions on Embedded Computing Systems (TECS), 13(2):1–27, 2013.

[8] A. Canziani, A. Paszke, and E. Culurciello. An analysis of deep neural network models for practical applications. arXiv preprint arXiv:1605.07678, 2016.

[9] CHaiDNN: An HLS based Deep Neural Network Accelerator Library for Xilinx Ultrascale+ MPSoCs. Avaialble: `https://github.com/Xilinx/CHaiDNN`.

[10] G. Charitopoulos, I. Koidis, K. Papadimitriou, and D. Pnevmatikatos. Hardware task scheduling for partially reconfigurable fpgas. In Applied Reconfigurable Computing:

11th International Symposium, ARC 2015, Bochum, Germany, April 13-17, 2015, Proceedings 11, pages 487–498. Springer, 2015.

[11] H. Chen, W. Qin, and L. Wang. Task partitioning and offloading in iot cloud-edge collaborative computing framework: a survey. Journal of Cloud Computing, 11(1):86, 2022.

[12] J. Chen, S. Chen, Q. Wang, B. Cao, G. Feng, and J. Hu. iraf: A deep reinforcement learning approach for collaborative mobile edge computing iot networks. IEEE Internet of Things Journal, 6(4):7011–7024, 2019.

[13] M. Chirila, P. D'Alberto, H.-Y. Ting, A. Veidenbaum, and A. Nicolau. A heterogeneous solution to the all-pairs shortest path problem using fpgas. In 2022 23rd International Symposium on Quality Electronic Design (ISQED), pages 108–113. IEEE, 2022.

[14] Y. Choi and M. Rhu. Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units. In 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 220–233. IEEE, 2020.

[15] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. arXiv preprint arXiv:1602.02830, 2016.

[16] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A {Low-Latency} online prediction serving system. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pages 613–627, 2017.

[17] A. Dairi, F. Harrou, Y. Sun, and M. Senouci. Obstacle detection for intelligent transportation systems using deep stacked autoencoder and k-nearest neighbor scheme. IEEE Sensors Journal, 18(12):5122–5132, 2018.

[18] N.-N. Dao, T.-T. Nguyen, M.-Q. Luong, T. Nguyen-Thanh, W. Na, and S. Cho. Self-calibrated edge computation for unmodeled time-sensitive iot offloading traffic. IEEE Access, 8:110316–110323, 2020.

[19] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In International conference on machine learning, pages 1407–1416. PMLR, 2018.

[20] J. Faraone, G. Gambardella, D. Boland, N. Fraser, M. Blott, and P. H. Leong. Customizing low-precision deep neural networks for fpgas. In 2018 28th International Conference on Field Programmable Logic and Applications (FPL), pages 97–973, 2018.

[21] P. Fournier-Viger, J. C.-W. Lin, R. U. Kiran, Y. S. Koh, and R. Thomas. A survey of sequential pattern mining. Data Science and Pattern Recognition, 1(1):54–77, 2017.

[22] P. A. Gagniuc. Markov chains: from theory to implementation and experimentation. John Wiley & Sons, 2017.

[23] M. Ghasemzadeh, M. Samragh, and F. Koushanfar. Rebnet: Residual binarized neural network. In 2018 IEEE 26th annual international symposium on field-programmable custom computing machines (FCCM), pages 57–64. IEEE, 2018.

[24] S. Ghodrati, B. H. Ahn, J. K. Kim, S. Kinzer, B. R. Yatham, N. Alla, H. Sharma, M. Alian, E. Ebrahimi, N. S. Kim, et al. Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks. In 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 681–697. IEEE, 2020.

[25] Google. Tensor Processing Unit (TPU). Avaialble: `https://cloud.google.com/tpu/docs/intro-to-tpu#edge_tpu`.

[26] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In International conference on machine learning, pages 1861–1870. PMLR, 2018.

[27] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services, pages 123–136, 2016.

[28] C. Hao, X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W.-m. Hwu, and D. Chen. Fpga/dnn co-design: An efficient design methodology for iot intelligence on the edge. In Proceedings of the 56th Annual Design Automation Conference 2019, pages 1–6, 2019.

[29] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. ACM SIGARCH Computer Architecture News, 43(3S):27–40, 2015.

[30] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778, 2016.

[31] G. Hegde, Siddhartha, N. Ramasamy, and N. Kapre. Caffepresso: An optimized library for deep learning on embedded accelerator-based platforms. In Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, pages 1–10, 2016.

[32] C.-H. Hong and B. Varghese. Resource management in fog/edge computing: a survey on architectures, infrastructure, and algorithms. ACM Computing Surveys (CSUR), 52(5):1–37, 2019.

[33] D. Horgan, J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. Van Hasselt, and D. Silver. Distributed prioritized experience replay. arXiv preprint arXiv:1803.00933, 2018.

[34] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861, 2017.

[35] P. Hu, S. Dhelim, H. Ning, and T. Qiu. Survey on fog computing: architecture, key technologies, applications and open issues. Journal of network and computer applications, 98:27–42, 2017.

[36] K.-Y. Huang and C.-H. Chang. Efficient mining of frequent episodes from complex sequences. Information Systems, 2008.

[37] L. Huang, X. Feng, C. Zhang, L. Qian, and Y. Wu. Deep reinforcement learning-based joint task offloading and bandwidth allocation for multi-user mobile edge computing. Digital Communications and Networks, 5(1):10–17, 2019.

[38] IBM Article: Edge computing architecture and use cases. Avaialble: `https://developer.ibm.com/articles/edge-computing-architecture-and-use-cases/`.

[39] IBM CPLEX® Optimizer. Available: `https://www.ibm.com/analytics/cplex-optimizer`.

[40] IDC White Paper - #US44413318: The digitization of the World From Edge to Core. Avaialble: `https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf`.

[41] S. Iftikhar, S. S. Gill, C. Song, M. Xu, M. S. Aslanpour, A. N. Toosi, J. Du, H. Wu, S. Ghosh, D. Chowdhury, et al. Ai-based fog and edge computing: A systematic review, taxonomy and future directions. Internet of Things, page 100674, 2022.

[42] Intel oneAPI. Avaialble: `https://www.intel.com/content/www/us/en/developer/tools/oneapi/training/fpga-development-flow.html/`.

[43] Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025. Avaialble: `https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/`.

[44] A. Iranfar, W. A. Simon, M. Zapater, and D. Atienza. A machine learning-based strategy for efficient resource management of video encoding on heterogeneous mpsocs. In 2018 IEEE International Symposium on Circuits and Systems (ISCAS), pages 1–5. IEEE, 2018.

[45] Z. István, G. Alonso, and A. Singla. Providing multi-tenant services with fpgas: Case study on a key-value store. In 2018 28th International Conference on Field Programmable Logic and Applications (FPL), pages 119–1195. IEEE, 2018.

[46] P. Jaccard. The distribution of the flora in the alpine zone. 1. New phytologist, 11(2):37–50, 1912.

[47] S. Jiang, D. He, C. Yang, C. Xu, G. Luo, Y. Chen, Y. Liu, and J. Jiang. Accelerating mobile applications at the network edge with software-programmable fpgas. In IEEE INFOCOM 2018 - IEEE Conference on Computer Communications, pages 55–62, 2018.

[48] S. Jiang, Z. Ma, X. Zeng, C. Xu, M. Zhang, C. Zhang, and Y. Liu. Scylla: Qoe-aware continuous mobile vision with fpga-based dynamic deep neural network reconfiguration. In IEEE INFOCOM 2020-IEEE Conference on Computer Communications, pages 1369–1378. IEEE, 2020.

[49] W. Jiang, E. H.-M. Sha, X. Zhang, L. Yang, Q. Zhuge, Y. Shi, and J. Hu. Achieving super-linear speedup across multi-fpga for real-time dnn inference. ACM Transactions on Embedded Computing Systems (TECS), 18(5s):67, 2019.

[50] W. Jiang, X. Zhang, E. H.-M. Sha, L. Yang, Q. Zhuge, Y. Shi, and J. Hu. Accuracy vs. efficiency: Achieving both through fpga-implementation aware neural architecture search. In Proceedings of the 56th Annual Design Automation Conference 2019, pages 1–6, 2019.

[51] M. Jridi, T. Chapel, V. Dorez, G. Le Bougeant, and A. Le Botlan. Soc-based edge computing gateway in the context of the internet of multimedia things: experimental platform. Journal of Low Power Electronics and Applications, 8(1):1, 2018.

[52] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. ACM SIGARCH Computer Architecture News, 45(1):615–629, 2017.

[53] S.-C. Kao and T. Krishna. Gamma: Automating the hw mapping of dnn models on accelerators via genetic algorithm. In Proceedings of the 39th International Conference on Computer-Aided Design, pages 1–9, 2020.

[54] S.-C. Kao and T. Krishna. Domain-specific genetic algorithm for multi-tenant dnn accelerator scheduling. arXiv preprint arXiv:2104.13997, 2021.

[55] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with AmorphOS. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 107–127, Carlsbad, CA, Oct. 2018. USENIX Association.

[56] K. Kim, S.-J. Jang, J. Park, E. Lee, and S.-S. Lee. Lightweight and energy-efficient deep learning accelerator for real-time object detection on edge devices. Sensors, 23(3):1185, 2023.

[57] J. H. Ko, T. Na, M. F. Amir, and S. Mukhopadhyay. Edge-host partitioning of deep neural networks with feature space encoding for resource-constrained internet-of-things platforms. In 2018 15th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS), pages 1–6. IEEE, 2018.

[58] H. Kooti and E. Bozorgzadeh. Reconfiguration-aware task graph scheduling. In 2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing, pages 163–167. IEEE, 2015.

[59] G. Korol, M. G. Jordan, M. B. Rutzig, and A. C. S. Beck. Synergistically exploiting cnn pruning and hls versioning for adaptive inference on multi-fpgas at the edge. ACM Transactions on Embedded Computing Systems (TECS), 20(5s):1–26, 2021.

[60] G. Korol, M. G. Jordan, M. B. Rutzig, and A. C. S. Beck. Adaflow: a framework for adaptive dataflow cnn acceleration on fpgas. In 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 244–249. IEEE, 2022.

[61] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems, 25, 2012.

[62] S. Kumar, P. Tiwari, and M. Zymbler. Internet of things is a revolutionary approach for future technology enhancement: a review. Journal of Big data, 6(1):1–21, 2019.

[63] D. C. Le, E. Y. Oh, J. H. Jeong, S. H. Kim, M. Jeon, J. Jang, and C.-H. Youn. An opencl-based sift accelerator for image features extraction on fpga in mobile edge computing environment. In 2018 International Conference on Information and Communication Technology Convergence (ICTC), pages 1406–1410. IEEE, 2018.

[64] Y. LeCun, Y. Bengio, et al. Convolutional networks for images, speech, and time series. The handbook of brain theory and neural networks, 3361(10):1995, 1995.

[65] J. Lee, J. Choi, J. Kim, J. Lee, and Y. Kim. Dataflow mirroring: Architectural support for highly efficient fine-grained spatial multitasking on systolic-array npus. In 2021 58th ACM/IEEE Design Automation Conference (DAC), pages 247–252. IEEE, 2021.

[66] Y. Li, C. Hao, X. Zhang, X. Liu, Y. Chen, J. Xiong, W.-m. Hwu, and D. Chen. Edd: Efficient differentiable dnn architecture and implementation co-search for embedded ai solutions. In 2020 57th ACM/IEEE Design Automation Conference (DAC), pages 1–6. IEEE, 2020.

[67] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei. Fp-bnn: Binarized neural network on fpga. Neurocomputing, 275:1072–1086, 2018.

[68] J. D. Little. A proof for the queuing formula: L= λ w. Operations research, 9(3):383–387, 1961.

[69] B. Liu, Z. Luo, H. Chen, and C. Li. A survey of state-of-the-art on edge computing: Theoretical models, technologies, directions, and development paths. IEEE Access, 10:54038–54063, 2022.

[70] M. Liu, W. Kuehn, Z. Lu, and A. Jantsch. Run-time partial reconfiguration speed investigation and architectural design space exploration. In 2009 International Conference on Field Programmable Logic and Applications, pages 498–502. IEEE, 2009.

[71] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. Ssd: Single shot multibox detector. In Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14, pages 21–37. Springer, 2016.

[72] X. Liu, J. Yang, C. Zou, Q. Chen, X. Yan, Y. Chen, and C. Cai. Collaborative edge computing with fpga-based cnn accelerators for energy-efficient and time-aware face tracking system. IEEE Transactions on Computational Social Systems, 9(1):252–266, 2021.

[73] P. Lou, L. Shi, X. Zhang, Z. Xiao, and J. Yan. A data-driven adaptive sampling method based on edge computing. Sensors, 20(8):2174, 2020.

[74] Q. Luo, S. Hu, C. Li, G. Li, and W. Shi. Resource scheduling in edge computing: A survey. IEEE Communications Surveys & Tutorials, 23(4):2131–2165, 2021.

[75] X. Luo, D. Li, Y. Yang, S. Zhang, et al. Spatiotemporal traffic flow prediction with knn and lstm. Journal of Advanced Transportation, 2019, 2019.

[76] D. Ma and X. Jiao. Hyperdimensional computing vs. neural networks: Comparing architecture and learning process. arXiv preprint arXiv:2207.12932, 2022.

[77] X. Ma, T. Yao, M. Hu, Y. Dong, W. Liu, F. Wang, and J. Liu. A survey on deep learning empowered iot applications. IEEE Access, 7:181721–181732, 2019.

[78] X. Ma, A. Zhou, S. Zhang, Q. Li, A. X. Liu, and S. Wang. Dynamic task scheduling in cloud-assisted mobile edge computing. IEEE Transactions on Mobile Computing, 2021.

[79] N. R. Mabroukeh and C. I. Ezeife. A taxonomy of sequential pattern mining algorithms. ACM Computing Surveys (CSUR), 43(1):1–41, 2010.

[80] H. Mannila, H. Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. Data mining and knowledge discovery, 1:259–289, 1997.

[81] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. Resource management with deep reinforcement learning. In Proceedings of the 15th ACM workshop on hot topics in networks, pages 50–56, 2016.

[82] J. M. Mbongue, A. M.-I. Shuping, P. Bhowmik, and C. Bobda. Architecture support for fpga multi-tenancy in the cloud. In 2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP), pages 125–132. IEEE, 2020.

[83] J. Meng, H. Tan, C. Xu, W. Cao, L. Liu, and B. Li. Dedas: Online task dispatching and scheduling with bandwidth constraint in edge computing. In IEEE INFOCOM 2019-IEEE Conference on Computer Communications, pages 2287–2295. IEEE, 2019.

[84] A. Milan, L. Leal-Taixé, I. Reid, S. Roth, and K. Schindler. Mot16: A benchmark for multi-object tracking. arXiv preprint arXiv:1603.00831, 2016.

[85] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In International conference on machine learning, pages 1928–1937. PMLR, 2016.

[86] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602, 2013.

[87] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. nature, 518(7540):529–533, 2015.

[88] V. N. Moothedath, J. P. V. Champati, and J. Gross. Energy efficient sampling policies for edge computing feedback systems. IEEE Transactions on Mobile Computing, 2022.

[89] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. De Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, et al. Massively parallel methods for deep reinforcement learning. arXiv preprint arXiv:1507.04296, 2015.

[90] H. Nakahara, T. Fujii, and S. Sato. A fully connected layer elimination for a binarizec convolutional neural network on an fpga. In 2017 27th international conference on field programmable logic and applications (FPL), pages 1–4. IEEE, 2017.

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

[92] Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2023, with forecasts from 2022 to 2030. Avaialble: `https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/`.

[93] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr. Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic. In 2016 International Conference on Field-Programmable Technology (FPT), pages 77–84. IEEE, 2016.

[94] L. Otten. LaTeX template for thesis and dissertation documents at UC Irvine. `https://github.com/lotten/uci-thesis-latex/`, 2012.

[95] J. Pan, G. Chen, H. Wu, X. Peng, and L. Xia. Deep reinforcement learning-based dynamic bandwidth allocation in weighted fair queues of routers. In 2022 IEEE 18th International Conference on Automation Science and Engineering (CASE), pages 1580–1587. IEEE, 2022.

[96] D. R. Patrikar and M. R. Parate. Anomaly detection using edge computing in video surveillance system. International Journal of Multimedia Information Retrieval, 11(2):85–110, 2022.

[97] H. Peng, S. Zhou, S. Weitze, J. Li, S. Islam, T. Geng, A. Li, W. Zhang, M. Song, M. Xie, et al. Binary complex neural network acceleration on fpga. In 2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP), pages 85–92. IEEE, 2021.

[98] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P. Reyes, M.-L. Shyu, S.-C. Chen, and S. S. Iyengar. A survey on deep learning: Algorithms, techniques, and applications. ACM Computing Surveys (CSUR), 51(5):1–36, 2018.

[99] G. Premsankar, M. Di Francesco, and T. Taleb. Edge computing for the internet of things: A case study. IEEE Internet of Things Journal, 5(2):1275–1284, 2018.

[100] Press Release: Intel acquired the Altera Corp. for $16.7 billion. Avaialble: `https://www.intc.com/news-events/press-releases/detail/302/intel-completes-acquisition-of-altera`.

[101] A. Purgato, D. Tantillo, M. Rabozzi, D. Sciuto, and M. D. Santambrogio. Resource-efficient scheduling for partially-reconfigurable fpga-based systems. In 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pages 189–197. IEEE, 2016.

[102] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. ACM SIGARCH Computer Architecture News, 42(3):13–24, 2014.

[103] Q-engineering. Available: `https://qengineering.eu/`.

[104] M. Raeis, A. Tizghadam, and A. Leon-Garcia. Queue-learning: A reinforcement learning approach for providing quality of service. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 35, pages 461–468, 2021.

[105] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In European conference on computer vision, pages 525–542. Springer, 2016.

[106] S. A. Razavi, H.-Y. Ting, T. Giyahchi, and E. Bozorgzadeh. On exploiting patterns for robust fpga-based multi-accelerator edge computing systems. In 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 116–119. IEEE, 2022.

[107] H. Rebecq, R. Ranftl, V. Koltun, and D. Scaramuzza. High speed and high dynamic range video with an event camera. IEEE transactions on pattern analysis and machine intelligence, 43(6):1964–1980, 2019.

[108] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 779–788, 2016.

[109] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. arXiv preprint arXiv:1804.02767, 2018.

[110] M. Rusci, A. Capotondi, and L. Benini. Memory-driven mixed low precision quantization for enabling deep network inference on microcontrollers. Proceedings of Machine Learning and Systems, 2:326–335, 2020.

[111] M. Sadeghi, S. A. Razavi, and M. S. Zamani. Reducing reconfiguration time in fpgas. In 2019 27th Iranian Conference on Electrical Engineering (ICEE), pages 1844–1848. IEEE, 2019.

[112] T. J. Saleem and M. A. Chishti. Deep learning for the internet of things: Potential benefits and use-cases. Digital Communications and Networks, 7(4):526–542, 2021.

[113] F. Samie. Resource management for edge computing in internet of things (iot). 2018.

[114] F. Samie, L. Bauer, and J. Henkel. From cloud down to things: An overview of machine learning in internet of things. IEEE Internet of Things Journal, 6(3):4921–4934, 2019.

[115] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. IEEE pervasive Computing, 8(4):14–23, 2009.

[116] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. arXiv preprint arXiv:1511.05952, 2015.

[117] J. Schmidhuber. Deep learning in neural networks: An overview. Neural networks, 61:85–117, 2015.

[118] ScienceTechworld.com: Scientech IoT Builder. Avaialble: `https://www.scientechworld.com/it-educational-platforms/iot-solutions/iot-builder/`.

[119] S. Shahhosseini, T. Hu, D. Seo, A. Kanduri, B. Donyanavard, A. M. Rahmani, and N. Dutt. Hybrid learning for orchestrating deep learning inference in multi-user edge-cloud networks. arXiv preprint arXiv:2202.11098, 2022.

[120] S. Sheng, P. Chen, Z. Chen, L. Wu, and Y. Yao. Deep reinforcement learning-based task scheduling in iot edge computing. Sensors, 21(5):1666, 2021.

[121] A. Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. Physica D: Nonlinear Phenomena, 404:132306, 2020.

[122] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556, 2014.

[123] D. Singh and C. K. Mohan. Deep spatio-temporal representation for detection of road accidents using stacked autoencoder. IEEE Transactions on Intelligent Transportation Systems, 20(3):879–887, 2018.

[124] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 1–9, 2015.

[125] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 2818–2826, 2016.

[126] M. M. Taye. Understanding of machine learning with deep learning: Architectures, workflow, applications and future directions. Computers, 12(5):91, 2023.

[127] The VIRAT Video Dataset. Available: `https://viratdata.org/index.html`.

[128] H.-Y. Ting, T. Giyahchi, A. A. Sani, and E. Bozorgzadeh. Dynamic sharing in multi-accelerators of neural networks on an fpga edge device. In 2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP), pages 197–204. IEEE, 2020.

[129] H.-Y. Ting, A. A. Sani, and E. Bozorgzadeh. System services for reconfigurable hardware acceleration in mobile devices. In 2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig), pages 1–6. IEEE, 2018.

[130] Towards a framework for developing extensible IoT applications. Avaialble: `https://mimove.inria.fr/zefxis/`.

[131] S. Tuli, S. Ilager, K. Ramamohanarao, and R. Buyya. Dynamic scheduling for stochastic edge-cloud computing environments using a3c learning and residual recurrent neural networks. IEEE transactions on mobile computing, 21(3):940–954, 2020.

[132] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers. Finn: A framework for fast, scalable binarized neural network inference. In Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays, pages 65–74, 2017.

[133] A. Vaishnav, K. D. Pham, D. Koch, and J. Garside. Resource elastic virtualization for fpgas using opencl. In 2018 28th International Conference on Field Programmable Logic and Applications (FPL), pages 111–1117. IEEE, 2018.

[134] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In Proceedings of the AAAI conference on artificial intelligence, volume 30, 2016.

[135] S. I. Venieris and C.-S. Bouganis. fpgaconvnet: A framework for mapping convolutional neural networks on fpgas. In 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pages 40–47. IEEE, 2016.

[136] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. Nature methods, 17(3):261–272, 2020.

[137] J. Wang, J. Hu, G. Min, A. Y. Zomaya, and N. Georgalas. Fast adaptive task offloading in edge computing based on meta reinforcement learning. IEEE Transactions on Parallel and Distributed Systems, 32(1):242–253, 2020.

[138] X. Wang, Y. Han, V. C. Leung, D. Niyato, X. Yan, and X. Chen. Convergence of edge computing and deep learning: A comprehensive survey. IEEE Communications Surveys & Tutorials, 22(2):869–904, 2020.

[139] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas. Dueling network architectures for deep reinforcement learning. In International conference on machine learning, pages 1995–2003. PMLR, 2016.

[140] W. Xiangxue, X. Lunhui, and C. Kaixun. Data-driven short-term forecasting for urban road network traffic based on data processing and lstm-rnn. Arabian Journal for Science and Engineering, 44:3043–3060, 2019.

[141] X. Xiong, K. Zheng, L. Lei, and L. Hou. Resource allocation based on deep reinforcement learning in iot edge computing. IEEE Journal on Selected Areas in Communications, 38(6):1133–1146, 2020.

[142] C. Xu, S. Jiang, G. Luo, G. Sun, N. An, G. Huang, and X. Liu. The case for fpga-based edge computing. IEEE Transactions on Mobile Computing, 21(7):2610–2619, 2020.

[143] P. Xu, X. Zhang, C. Hao, Y. Zhao, Y. Zhang, Y. Wang, C. Li, Z. Guan, D. Chen, and Y. Lin. Autodnnchip: An automated dnn chip predictor and builder for both fpgas and asics. In Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pages 40–50, 2020.

[144] L. Yan, S. Cao, Y. Gong, H. Han, J. Wei, Y. Zhao, and S. Yang. Satec: A 5g satellite edge computing framework based on microservice architecture. Sensors, 19(4):831, 2019.

[145] Z. Yang, K. Nahrstedt, H. Guo, and Q. Zhou. Deeprt: A soft real time scheduler for computer vision applications on the edge. In 2021 IEEE/ACM Symposium on Edge Computing (SEC), pages 271–284. IEEE, 2021.

[146] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, and Q. Li. Lavea: Latency-aware video analytics on edge computing platform. In Proceedings of the Second ACM/IEEE Symposium on Edge Computing, pages 1–13, 2017.

[147] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang. A survey on the edge computing for the internet of things. IEEE access, 6:6900–6919, 2017.

[148] Y. Zeng, M. Chao, and R. Stoleru. Emuedge: A hybrid emulator for reproducible and realistic edge computing experiments. In 2019 IEEE International Conference on Fog Computing (ICFC), pages 153–164. IEEE, 2019.

[149] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays, pages 161–170, 2015.

[150] D. Zhang, N. Vance, Y. Zhang, M. T. Rashid, and D. Wang. Edgebatch: Towards ai-empowered optimal task batching in intelligent edge systems. In 2019 IEEE Real-Time Systems Symposium (RTSS), pages 366–379. IEEE, 2019.

[151] W. Zhang, Z. Zhang, S. Zeadally, and H.-C. Chao. Efficient task scheduling with stochastic delay cost in mobile edge computing. IEEE Communications Letters, 23(1):4–7, 2018.

[152] Z. Zhang. Improved adam optimizer for deep neural networks. In 2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS), pages 1–2, 2018.

[153] C. Zhao, C. Xiao, and Y. Liu. A real-time reconfigurable edge computing system in industrial internet of things based on fpga. In 2021 IEEE 16th Conference on Industrial Electronics and Applications (ICIEA), pages 480–485. IEEE, 2021.

[154] H. Zhao and R. Sakellariou. An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm. In Euro-Par 2003 Parallel Processing: 9th International Euro-Par Conference Klagenfurt, Austria, August 26-29, 2003 Proceedings 9, pages 189–194. Springer, 2003.

[155] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang. Accelerating binarized convolutional neural networks with software-programmable fpgas. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pages 15–24, 2017.

[156] T. Zheng, J. Wan, J. Zhang, and C. Jiang. Deep reinforcement learning-based workload scheduling for edge computing. Journal of Cloud Computing, 11(1):3, 2022.

[157] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. arXiv preprint arXiv:1606.06160, 2016.

[158] Z. Zhu, J. Zhang, J. Zhao, J. Cao, D. Zhao, G. Jia, and Q. Meng. A hardware and software task-scheduling framework based on cpu+fpga heterogeneous architecture in edge computing. IEEE Access, 7:148975–148988, 2019.

[159] J. Zou, T. Hao, C. Yu, and H. Jin. A3c-do: A regional resource scheduling framework based on deep reinforcement learning in edge scenario. IEEE Transactions on Computers, 70(2):228–239, 2021.