UNIVERSITY OF CALIFORNIA,
IRVINE

Hardware/Software Co-design Methodologies for Efficient AI Systems and Applications

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Electrical and Computer Engineering

by

Mohanad Mohamed Abdelmagid Abdelkhalek Odema

Dissertation Committee:
Professor Mohammad Abdullah Al Faruque, Chair
Professor Marco Levorato
Professor Hyoukjun Kwon

2024

DEDICATION

To my Mum and Brother

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

I thank my advisor and committee chair, Professor Mohammad Abdullah Al Faruque, for his mentorship, guidance, and support throughout my doctoral studies. I would also like to express my gratitude to my committee members, Professor Marco Levorato and Professor Hyoukjun Kwon, for serving on my doctoral committee and for all our research collaborations.

I collaborated with several colleagues during my graduate studies on various research projects. Specifically, I would like to thank Luke Chen, Halima Bouzidi, Nafiul Rashid, Arnav Malawade, and Berken Utku Demirel for their contributions to the projects presented in this dissertation. Additionally, I thank my external collaborators, James Ferlez, Yasser Shoukry, Smail Niar, and Hamza Ouarnoughi for our collaborations on several projects presented in this dissertation. I also want to thank my past and present colleagues at the AICPS lab: Trier Mortlock, Yonatan Giazchew, Kohei Tsujio, Junyao Wang, Mohammed Habib Fakih, Yasamin Moghaddas, Tyler Zhang, Rozhin Yasaei, and Shih-Yuan Yu. I enjoyed sharing this journey with them immensely.

I would like to thank UCI's Department of Electrical Engineering and Computer Science for their support during my PhD. I also thank the National Science Foundation for their funding assistance via awards CCF-2140154 and CMMI-1739503. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies. I thank IEEE, ACM, and original authors for granting permission to include the content for my dissertation based on content originally published in their IEEE and ACM transactions and conference papers.

VITA

Mohanad Mohamed Abdelmagid Abdelkhalek Odema

EDUCATION

Doctor of Philosophy in Electrical and Computer Engineering	**2024**
University of California, Irvine	*Irvine, California*
Master of Science in Computer Engineering	**2018**
Ain Shams University	*Cairo, Egypt*
Bachelor of Science in Electrical Engineering	**2014**
Ain Shams University	*Cairo, Egypt*

RESEARCH EXPERIENCE

Graduate Research Assistant	**2019-2024**
University of California, Irvine	*Irvine, California*
AI Architecture and Algorithm Intern	**2023-2023**
MediaTek	*San Jose, California*
Research Assistant	**2015-2018**
Ain Shams University	*Cairo, Egypt*

TEACHING EXPERIENCE

Teaching Assistant	**2022-2024**
University of California, Irvine	*Irvine, California*
Teaching Assistant	**2015-2019**
Ain Shams University	*Cairo, Egypt*

REFEREED JOURNAL PUBLICATIONS

EPIC: Efficient Autonomous Driving Perception via Integrated Accelerator Chiplets (Under Review) **2024**
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)

MaGNAS: A Mapping-Aware Graph Neural Architecture Search Framework for Heterogeneous MPSoC Deployment **2023**
ACM Transactions on Embedded Computing Systems (TECS)

PrivyNAS: Privacy-Aware Neural Architecture Search for Split Computing in Edge-Cloud Systems **2023**
IEEE Internet of Things Journal (IoTJ)

Testudo: Collaborative Intelligence for Latency-Critical Autonomous Systems **2022**
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)

SAGE: A Split-Architecture Methodology for Efficient End-to-End Autonomous Vehicle Control **2021**
ACM Transactions on Embedded Computing Systems (TECS)

REFEREED CONFERENCE PUBLICATIONS

SCAR: Scheduling Multi-Model AI Workloads on Heterogeneous Multi-Chiplet Module Accelereators (Accepted) **2024**
IEEE/ACM International Symposium on Microarchitecture (MICRO)

SEO: Safety-Aware Energy Optimization Framework for Multi-Sensor Neural Controllers at the Edge **2023**
ACM/IEEE Design Automation Conference (DAC)

Map-and-Conquer: Energy-Efficient Mapping of Dynamic Neural Nets onto Heterogeneous MPSoCs **2023**
ACM/IEEE Design Automation Conference (DAC)

EnergyShield: Provably-Safe Offloading of Neural Network Controllers for Energy Efficiency **2023**
ACM/IEEE 14th International Conference on Cyber-Physical Systems (ICCPS)

HADAS: Hardware-Aware Dynamic Neural Architecture Search for Edge Performance Scaling **2023**
Design, Automation & Test in Europe Conference & Exhibition (DATE)

Romanus: Robust Task Offloading in Modular Multi-Sensor Autonomous Driving Systems **2022**
IEEE/ACM International Conference on Computer-Aided Design (ICCAD)

Template Matching Based Early Exit CNN for Energy-efficient Myocardial Infarction Detection on Low-power Wearable Devices **2022**
ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT)

LENS: Layer Distribution Enabled Neural Architecture Search in Edge-Cloud Hierarchies **2021**
ACM/IEEE Design Automation Conference (DAC)

EExNAS: Early-Exit Neural Architecture Search Solutions for Low-Power Wearable Devices **2021**
IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)

Energy-Aware Design Methodology for Myocardial Infarction Detection on Low-Power Wearable Devices **2021**
Asia and South Pacific Design Automation Conference (ASP-DAC)

SOFTWARE

EnergyShield `https://github.com/MohanadOdema/EnergyShield`
Oepn Source Code for EnergyShield provably-safe optimizations in CARLA Simulator

ABSTRACT OF THE DISSERTATION

Hardware/Software Co-design Methodologies for Efficient AI Systems and Applications

By

Mohanad Mohamed Abdelmagid Abdelkhalek Odema

Doctor of Philosophy in Electrical and Computer Engineering

University of California, Irvine, 2024

Professor Mohammad Abdullah Al Faruque, Chair

The landscape of AI research is dominated by the search for powerful deep learning models and architectures that enable fascinating applications from the edge to the cloud. Indeed, we have witnessed the emergence of efficient, on-device deep learning models that facilitate smart edge applications (autonomous vehicles, AR/VR systems), and the emergence of billion parameter foundation/LLM models that excel at tasks thought achievable only through human-level understanding. On the other hand, the calls for more advanced hardware and systems continue to grow considering the scale at which deep learning model workloads evolve, and to facilitate sustainable, efficient model operation across the various application contexts.

This suggests a natural way to design deep learning models and their systems: viz, through hardware/software co-design methodologies, capturing the interplay and mutual dependencies across various HW/SW layers of the computing stack to guide different design choices. From the algorithmic side, an awareness of the target platform's compute capabilities and resources guides the deep learning model architectural and optimization choices (e.g., compression) towards maximizing performance efficiency on the target hardware at deployment time. From the hardware side, understanding the deep learning workloads and computing kernels can shape future architectures of AI hardware that improves on efficiency from the

lower levels (as seen through customized accelerators). Even more so, frameworks like TVM and ONNX Runtime have also emerged to standardize model deployment on various target hardware systems, offering unified interfaces to enact necessary compiler optimizations.

As hardware and software continue to undergo continuous innovation, this dissertation aims to investigate relevant emergent technologies and challenges at this unified research frontier to guide the design of future AI systems and models. The dissertation focuses on characterizing nascent design spaces, exploring various optimization opportunities, and developing new methodologies to maximize the impact of such innovations. In brief, this dissertation goes over the following topics:

- Understanding the benefits of dynamic neural networks for efficient inference, and how to optimize their design for target platform deployment
- Studying emergent models (like Graph Neural Networks) with irregular computational flows and how their design can be optimized for deployment on heterogeneous SoCs
- Understanding how multi-model workloads can be scheduled and co-located on multi-chip AI Accelerator modules based on 2.5D chiplets technology while accounting for workloads' diversity, affinities, and memory access patterns
- Exploring new methodologies to maximize the impact of split computing inference in edge-cloud architectures, and elevate resource efficiency of edge devices
- Studying the impact emergent schemes like split computing could have on the broader cyber-physical system and application with regards to safety and privacy, and proposing methods to counteract potential disruptions and maintain desired formal guarantees

Chapter 1

Introduction

The AI ecosystem is going through seismic transformations driven by recent innovations in the field. Though the release of ChatGPT is widely regarded as the watershed moment for the AI field, its success can be attributed to decades of technological advancements across multiple sectors leading up to that moment. To put it succinctly, the brilliance and promise of AI technology rests on the strong, mature foundation of enabling industries of semiconductor, hardware manufacturing, and software development. And today, AI is becoming a vortex around which these industries - and more - are oriented. In fact, the nature of AI's relationship with these industries is shifting from reliance towards co-dependence. For instance, AI advancements and application use cases are fueling innovation and shaping the vision of semiconductor and hardware manufacturing industries, while at the same time they remain dependent on breakthroughs in the very same fields to realize their full potential.

This mutual co-dependence underpins an intuitive philosophy: Hardware/software co-design is pivotal to bringing the best out of AI applications by enhancing their performance efficiency on target systems. From an algorithmic perspective, optimizing the design and deployment of underlying machine learning models is to be performed in a hardware-aware manner. This is driven by how underlying resource constraints and hardware capabilities could influence decisions pertaining to the extent of a model's representational capacity through architecture parameter choices, compression techniques, scheduling options, and so on. As for the hardware, understanding the emerging trends in AI model workloads guides the design of the next generations of AI hardware, as we see today through the progression of computing capabilities in GPUs, domain specific accelerators, and heterogeneous SoCs.

While standard tools such as ONNX Runtime and TVM have emerged to optimize model deployment on target hardware platforms, the rapid evolution of AI usage scenarios, model architectures, semiconductor technologies has outpaced current optimization practices. That is, despite the effectiveness of standard model optimization techniques like quantization, AutoML, operator fusion, new challenges have emerged from the trends of multi-model workloads in AR/VR systems and data center multi-tenancy, as well as from recent innovations in dynamic and graph neural networks. Similarly, advancements in semiconductor technologies, such as chiplet-based 2.5D multi-chip module AI accelerators, have expanded the HW/SW optimization space beyond the capabilities of existing practices.

In this dissertation, our aim is to study and understand these nascent frontiers, exploring the optimization opportunities they present from a hardware/software co-design perspective. As detailed in the following chapters, our research consistently guides us towards characterizing novel design spaces, for which we propose methodologies to maximize the performance of AI applications and systems in these evolving settings.

1.1 Research Scope

Figure 1.1 illustrates our abstract hardware/software research vertical that is traversed throughout this dissertation. This characterization serves to organize the the various con-

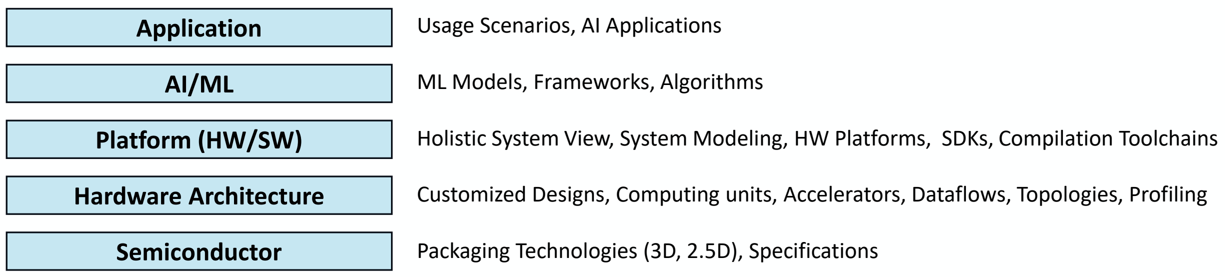

Application
Usage Scenarios, AI Applications
AI/ML
ML Models, Frameworks, Algorithms
Platform (HW/SW)
Holistic System View, System Modeling, HW Platforms, SDKs, Compilation Toolchains
Hardware Architecture
Customized Designs, Computing units, Accelerators, Dataflows, Topologies, Profiling
Semiconductor
Packaging Technologies (3D, 2.5D), Specifications

Figure 1.1: Abstract model of the HW/SW research vertical in this dissertation.

cepts, technologies, and innovations pertaining to different layers of abstraction from the broader hardware/software stack. That way, not only do we analyze and innovate methods/techniques in isolation, but also understand cross-layer impacts, and how to improve the overall AI system utility and efficiency. For instance, advancements in 2.5D packaging technology in the 'Semiconductor' layer enable new customized designs in the 'Hardware Architecture' layer, improving the efficiency of AI/ML workload processing, and providing better application usage experiences at the higher levels of the stack.

In this dissertation, we study the hardware-software co-design challenges for AI algorithms and systems from a variety of perspectives, and propose novel methodologies to enhance the efficiency of AI on respective target hardware systems. Our studies are motivated by and shaped through real-world, practical applications, use cases, and emerging trends of AI models and systems from the edge to the cloud. In particular, emphasis is drawn from: (i) The Edge-AI application domain, which entails machine learning models deployed on constrained edge computing devices in the wild (e.g., autonomous driving systems). (ii) The multi-model AI workload trend, which entails a diverse set of large, complex models running simultaneously on a target system to meet application demands (e.g., multi-tenancy and AR/VR). From here, our studies take us to understand the underlying model architectures and classes (transformers, CNNs, GNNs), computational workflow (static, dynamic), and constituent compute kernels (GEMMs, Non-GEMMs), enabling us to shape the design space of algorithmic optimizations.

As for the hardware, we focus our analysis on two classes of systems corresponding to the above areas. (i) Heterogeneous SoCs, which service numerous Edge-AI applications (e.g., Tesla FSD for autonomous driving), and constitute a diverse set of computing units, such as CPUs, GPUs, and DSPs, all integrated onto the same die. (ii) Chiplet-based 2.5D multi-chip modules, which - owing to the mix-and-match characteristic of chiplets - enable integration of heterogeneous accelerator chips on the package level to efficiently service multi-model workloads. Through both, we are able to characterize hardware design spaces that cover different dimensions of hardware optimization options, including scheduling, pipelining, hardware reconfigurability, and accelerator hardware parameter choices.

Furthermore, we study the emerging paradigm of split-computing and how its effectiveness can be maximized through co-optimization approaches. As inferred from the name, split-computing partitions a machine learning model between a constrained user edge device and a compute capable edge server, offering a viable alternative for enabling powerful, on-device machine learning without compromising on the model's representational capacity. We also engage specific challenges of split-computing including context-aware runtime adaptation and maintaining formal safety guarantees when applied to mission-critical systems.

1.2 Primer: Neural Architecture Search

A recurring theme throughout this dissertation is Neural Architecture Search (NAS), which is an automated design space exploration technique to optimize the design of neural architectures for target machine learning tasks. NAS can provide DNN architectures that are on-par or outperform their manually-crafted counterparts [22, 25, 26, 232]. In a nutshell, the purpose of NAS is to effectively navigate an enormous design space of neural architectural parameters to identify optimal candidate model architecture designs suited for the target objectives. Primarily, there are three foundational pillars to any NAS framework:

Search Space. Through the coalescence of various combinations of architectural design parameters, a pool of candidate designs can be constructed for the NAS engine to access. Thus, DNN architectural parameter choices such as the #layers, # channels, type of layer operation, etc can all be encoded into a single unified search string.

Search Controller. Due to the colossal size of typical DNN architectural search spaces, NAS frameworks employ search controllers adopting sophisticate search strategies to effectively balance the exploration/exploitation of the search space. In particular, search controllers learn to identify promising design subspaces from which they can sample superior population of architectural candidates, reducing the likelihood of considering sub-optimal designs in the interim. These controllers are typically adopt strategies that follow a learning-based approach (e.g., reinforcement learning based [26, 232]) or a metaheuristic approach (e.g., evolutionary algorithm [22]).

Performance Evaluation. In order to guide the search controller on promising architectural design sub-spaces, performance evaluations of the candidate model design on the target objectives (e.g., accuracy and execution latency) are fed back to the search controller for it to exploit optimizations around the top-performing candidates in a progressive manner.

Traditionally, classical NAS frameworks [142, 232] relied on *training* candidate models from scratch to determine the accuracy scores needed for comparison. However, to circumvent the inefficiency of training each candidate model to convergence before throwing away all its parameters, recent NAS approaches [22, 25, 26] proposed a *one-shot* approach in which all candidate models are trained simultaneously through the concepts of *supernet* and *shared weights*. Briefly, the idea relies on specifying the search space as a single, multi-path over-parameterized network model (i.e., a supernet) that encapsulates all candidate architectural designs within (i.e., subnets). In this case, the sampling of a candidate model from the supernet is achieved through selecting for each potential layer position a particular path reflecting a specific architecture choice, as in choosing a path representing a 3×3 convolution.

Then, the weights associated with the 3×3 convolution are loaded within the supernet, used for the candidate model evaluation, and are updated in the next update step.

1.3 Organization

The rest of the disseration is organized as follows. Chapter 2 discusses how to co-optimize dynamic neural networks design and deployment onto heterogeneous MPSoCs. Chapter 3 introduces our novel mapping-aware graph neural architecture search for heterogenous MPSoC deployment. Chapter 4 addresses the challenges of scheduling multi-model workloads on emerging 2.5D architectures, and presents a novel framework to effectively enact such scheduling. Chapter 5 moves on to discuss efficient split-computing approaches for Edge-AI autonomous system applications, and chapter 6 engages the offloading challenges with regards to maintaining guarantees on safety in the cases of autonomous driving. Chapter 7 presents our overall findings and insights from our studies, lays out our study limitations, and provides directions for future research.

Chapter 2

Optimizing Dynamic Neural Networks Design and Deployment onto Heterogeneous MPSoCs

2.1 Introduction

The hardware era has witnessed the emergence of various computing devices, from powerful GPUs to tiny Micro-controllers. To meet the requirements of compute-intensive applications, such as Deep Learning workloads, MPSoCs are designed to incorporate heterogeneous computing units (CU) within the same die, typically sharing the same system memory (DRAM). This hardware architecture paradigm enables the collaborative usage of multiple CUs to accelerate different operations of the same application, hence providing energy savings and performance benefits. However, the causality between the hardware heterogeneity of MPSoC and the obtained performance for similar and different operations remains an open research question. Indeed, some CUs (e.g., GPUs) can offer high execution speedup at the cost of being energy-hungry, while others, such as NPUs, are power-friendly at the cost of being slow. Conventional deployment schemes lack a holistic overview of how heterogeneous CUs may behave regarding various computing workloads. In addition, the systematic approach of considering a single CU to deploy an entire application is suboptimal since it overlooks opportunities for further performance gains through maximizing the utilization of the MPSoC's hardware resources.

Recent works have shed light on the *computation mapping* problem for MPSoC by providing comprehensive modeling methodologies in [40, 132, 173, 212] to characterize computing workloads performances. The resulting models are typically used to map computations onto CUs in a sequential pipeline fashion. However, for workloads exhibiting a high degree of parallelism, such as Neural Networks ($\mathcal{NN}$), there's still room for improvement by refashioning the execution pipeline into parallel stages running concurrently on different CUs, especially considering the inherent capacity for concurrency within $\mathcal{NN}$ layers such as convolutional and multi-head self-attention layers [66].

On the other hand, recent research works have introduced Dynamic Neural Networks (DyNNs) [69] which contrary to conventional *static* neural network models with fixed computational graphs, offer to adapt their model structure or parameters to suit the runtime context, offering resource efficiency at the edge while maintaining the models' utility. The motivation behind DyNNs being the in-the-wild deployment of machine learning models at the edge which makes them susceptible to considerable runtime variations. One prominent DyNN technique is early exiting which facilitates concluding the processing of the "easier" input samples at earlier layers of a model for resource efficiency. This feature is often realized through a multi-exit architecture that integrates intermediate classifiers onto a shared backbone model [140, 144, 186].

Typically, the design workflow of dynamic networks like multi-exit models initially assumes that a backbone architecture has been *optimally* designed to maximize performance on a target task. Evidently, backbones in related works were either based on renowned state-of-the-art NN architectures, e.g., ResNets in [186], or models rendered through the design automation frameworks of Neural Architecture Search (NAS) [194]. This means that backbones were originally designed to serve as *standalone* static models. Thus, a subject of debate is whether such design optimality of these models would hold when auxiliary tasks are added – as in to serve as the backbone of a dynamic model.

Given this pretext, we identify several limitations of current practice for designing and deploying DyNNs onto heterogeneous MPSoCs:

- The backbone model architectures of DyNNs are not optimized for dynamic inference
- State-of-the-art NN design frameworks (e.g., NAS) do not characterize the runtime aspects of dynamic inference, potentially leading to suboptimal DyNN design
- The co-optimization synergy between DyNNs design, underlying heterogeneity of hardware, and DVFS configuration settings remain under-exploited.

2.1.1 Novel Contributions

Addressing these limitations, we present the following contributions in this Chapter:

- We present HADAS [21], a novel hardware-aware Dynamic Neural Architecture Search Framework for joint optimization of multi-exit DyNNs design and DVFS settings for efficient edge operation. HADAS can be integrated on top of modern pretrained supernets of existing state-of-the-art NAS frameworks.
- We present Map-and-Conquer [23], a framework for transforming static neural networks to DyNNs with multiple inference stages leveraging heterogeneous pipelining parallelism across the model *width* dimension.

2.2 Related Works

Dynamic Neural Networks. Dynamic Neural Networks serve as attractive solutions to scale computation according to the input complexity, providing latency speedup and energy gains. Incorporating dynamicity into NN inference has been widely studied for CNN architectures through early-exiting along the architecture's depth [144, 186] or width [225]. Early-exiting has also been successfully applied for emerging classes of models like Vision Transformers (ViT) [155, 224]. In the above works, the multi-exit networks have been manually designed based on heuristic choices of exits' positions, structure, and count conditioned on their respective backbone architecture [103]. Recent works [140, 225] have investigated the applicability of NAS techniques to automate the design of multi-exit networks, where the backbone and exits' design spaces can be jointly explored to reach superior DyNN architectures. However, a scalable, more generic solutions for designing DyNNs remaind lacking.

Computation mapping on MPSoCs. Recent MPSoCs contain diverse heterogeneous CUs that usually share system memory, making them more flexible for collaborative execution. Recent works have explored this specificity of MPSoC to optimize the execution of $\mathcal{NN}$. AxoNN and MEPHESTO [40, 132, 212] propose modeling strategies to characterize execution latency and energy consumption for computation mapping on the AGX Xavier MPSoC. Jedi [82] provides a framework built upon TensorRT to accelerate $\mathcal{NN}$ via model parallelism to maximize throughput for batched inference. [87, 90] proposes evolutionary-based scheduling for NN layers on heterogeneous MPSoCs with *DVFS* by exploiting both data and model parallelism to optimize the throughput. DistrEdge [76] provides a detailed analysis of different model parallelism schemes for distributed computing over edge devices. However, none of the prior works have considered the design of dynamic NN in the computation mapping problem for collaborative execution on MPSoCs.

Dynamic hardware reconfiguration: Dynamically scaling NNs results in different computational and energy footprints that require adapting the hardware configuration accordingly. In [52, 149], the hardware has been co-designed with the multi-exit networks using FPGAs, showcasing how further energy efficiency gains can be achieved through having specialized hardware for exits. Nevertheless, the considerable switching overheads of hardware configurations in FPGAs are not typically acceptable for runtime applications. A viable alternative came in the form of hardware reconfiguration through supported DVFS features, where the operational frequency can be scaled after exiting to preserve energy resources [108, 182].

2.3 HADAS: Hardware-Aware Dynamic Neural Architecture Search

We formulate the dynamic neural architecture search formulation, present the HADAS framework, and provide evaluation results on its efficacy compared to conventional approaches.

2.3.1 Problem Formulation

As the combined design space size for the DyNNs and hardware configurations can be enormous, we characterize three separate subspaces to manage the joint optimization of their parameters as follows: (*i*) ***The backbones*** **($\mathcal{B}$)**; which are models originally designed in a monolithic fashion for *static* inference with no adaptive behavior, (*ii*) ***The exits*** **($\mathcal{X}$)**; which are the dynamic components to be integrated onto a backbone, and (*iii*) ***The DVFS settings*** **($\mathcal{F}$)**; constituting the space of operational frequencies for the underlying hardware components. For the DyNNs, our reasons for designating $\mathcal{B}$ and $\mathcal{X}$ as separate subspaces are twofold: (*a*) To maintain the generality of the approach by having the $\mathcal{X}$ subspace indifferent to the "*type*" of candidate backbones in $\mathcal{B}$, and (*b*) To leverage the existing infrastructure of pretrained supernets from established NAS frameworks (as in [25, 194]) so as to provide high-caliber backbone models for the $\mathcal{B}$ subspace.

In order to rank candidate dynamic architectural designs, we denote $\mathcal{S}$ and $\mathcal{D}$ as generic performance objectives under *static* and *dynamic* deployments, respectively. Mainly, $\mathcal{S}$ represents the backbone evaluations when designated as a fixed standalone model (e.g., baseline energy), whereas $\mathcal{D}$ is for the evaluations of its dynamic variant after integrating the exits (e.g., average energy when effective mapping of inputs to exits). Hence, this implies a bi-level optimization problem with the $\mathcal{B}$ as the outer-level subspace and ($\mathcal{X}$, $\mathcal{F}$) as the inner-level:

$$b^* = \underset{b \in \mathcal{B}}{\arg\max}\ \psi[\mathcal{S}(b), \mathcal{D}(x^*, f^* \mid b)] \tag{2.1}$$

$$s.t.\ x^*, f^* = \underset{x \in \mathcal{X}, f \in \mathcal{F}}{\arg\max}\ \mathcal{D}(x, f \mid b) \tag{2.2}$$

where the global optimization objective to identify the ideal parameter combination (b^*, x^*, f^*) that maximizes a global function ψ combining the performance objectives of $\mathcal{S}$ and $\mathcal{D}$. In practice, the underlying optimization objectives are conflicting by nature – e.g., the larger, expensive models enjoy higher accuracy scores and vice versa. Thus, the problem can be approached as a multi-objective optimization searching for Pareto optimal solutions.

2.3.2 HADAS Framework

HADAS solves the bi-level optimization via a nested metaheuristic framework in Figure 2.1.

Outer Optimization Engine (OOE)

The OOE considers two primary tasks: (i) Searching through $\mathcal{B}$ to identify the best backbone candidates, and (ii) Ranking DyNNs according to their aggregate $\mathcal{S}$ and $\mathcal{D}$ evaluations.

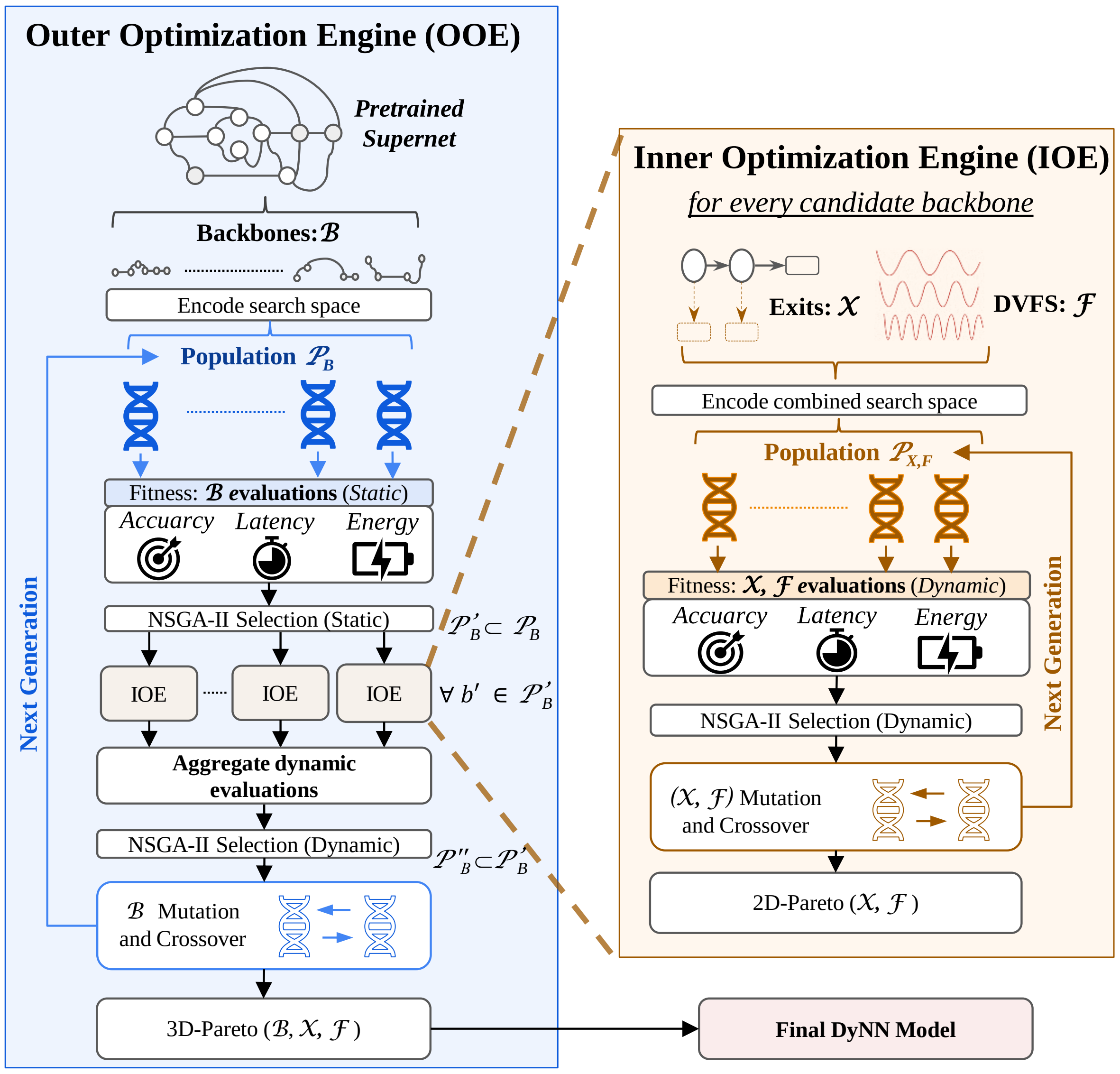

Outer Optimization Engine (OOE)
Pretrained Supernet
Backbones: $\mathcal{B}$
Encode search space
Population $\mathcal{P}_B$
Fitness: $\mathcal{B}$ evaluations (Static)
Accuracy
Latency
Energy
NSGA-II Selection (Static)
$\mathcal{P}'_B \subset \mathcal{P}_B$
IOE
IOE
IOE
$\forall\, b' \in \mathcal{P}'_B$
Aggregate dynamic evaluations
NSGA-II Selection (Dynamic)
$\mathcal{P}''_B \subset \mathcal{P}'_B$
$\mathcal{B}$ Mutation and Crossover
3D-Pareto ($\mathcal{B}, \mathcal{X}, \mathcal{F}$)
Next Generation
Inner Optimization Engine (IOE)
for every candidate backbone
Exits: $\mathcal{X}$
DVFS: $\mathcal{F}$
Encode combined search space
Population $\mathcal{P}_{X,F}$
Fitness: $\mathcal{X}, \mathcal{F}$ evaluations (Dynamic)
Accuracy
Latency
Energy
NSGA-II Selection (Dynamic)
($\mathcal{X}, \mathcal{F}$) Mutation and Crossover
2D-Pareto ($\mathcal{X}, \mathcal{F}$)
Next Generation
Final DyNN Model

Figure 2.1: HADAS co-optimization framework.

$\mathcal{B}$ Subspace. Modern NAS frameworks employ a Once-For-All (OFA) approach which entails first training a large over-parameterized *supernet* on a target task, prior to applying a search algorithm to identify the optimal subnet designs within. The enabling factor of OFA approaches is that all of the supernet's parameters are *shared* by its subnets, effectively rendering the *training* and *search* procedures as disjoint processes, which dramatically reduces the overall overheads within the NAS framework [25, 194]. From here, HADAS is built to leverage the pretrained supernets of existing NAS frameworks to construct the $\mathcal{B}$ subspace of backbones, where the search space can be *encoded* into discrete variables usable by the search algorithm, and each viable subnet (backbone) can be denoted as $b \in \mathcal{B}$.

$\mathcal{B}$ Evolutionary Search. With $\mathcal{B}$ defined, the dynamic architecture search initiates in the OOE through an evolutionary search algorithm (e.g., NSGA-II) that can navigate through $\mathcal{B}$ to sample promising backbone models. In particular, the evolutionary algorithm is set to run for a predefined number of generations G, generating with every generation, g, a population of backbones, $\mathcal{P}^g_{\mathcal{B}}$, from which the encoded pretrained subnets can be sampled. Afterwards, $\forall b \in P^g_{\mathcal{B}}$, a fitness evaluation under *static* conditions is performed as:

$$\mathcal{S}(b) = Fit(Acc_b, L_b, E_b) \tag{2.3}$$

where $\mathcal{S}(b)$ is a vector of the *static* performance evaluations with regards to the accuracy (Acc_b), latency (L_b), and energy (E_b), respectively. Estimates for L_b and E_b are obtained based on hardware measurements – as through a HW-in-the-loop setup (adopted here), lookup tables, or prediction models. At this stage, we remark that hardware evaluations are based on default HW settings, leaving the DVFS optimizations for the IOE. Based on the $\mathcal{S}$ scores, every $b \in P^g_{\mathcal{B}}$ is ranked using the NSGA-II non-dominated sorting algorithm. If a number of backbones shared the same rank, their diversity scores are used for re-ranking. This early selection procedure enables pruning the population to reach a smaller subset $\mathcal{P}^{g'}_{\mathcal{B}} \subset \mathcal{P}^g_{\mathcal{B}}$, where every $b' \in P^{g'}_{\mathcal{B}}$ is mapped to an IOE (detailed later) to obtain the overall dynamic architecture evaluations $\mathcal{D}(x^*, f^* \mid b')$.

Once an IOE concludes its procedures, a Pareto optimal set of exits placement and DVFS settings is returned to the OOE for every $b' \in P^{g'}_{\mathcal{B}}$. These Pareto sets are then collectively aggregated for a second selection algorithm that ranks backbones based on their combined $\mathcal{S}$ and $\mathcal{D}$ scores, leading to another population subset $\mathcal{P}^{g''}_{\mathcal{B}} \subset \mathcal{P}^{g'}_{\mathcal{B}}$. Lastly, $\mathcal{P}^{g''}_{\mathcal{B}}$ undergoes *mutation* and *crossover* operations to construct a new population $\mathcal{P}^{g+1}_{\mathcal{B}}$ for generation $g{+}1$.

This outer loop cycle repeats until generation G at which the Pareto optimal set (b^*, x^*, f^*) is returned as the final solution.

Inner Optimization Engine (IOE)

The IOE is invoked for every $b' \in \mathcal{P}_{\mathcal{B}}^{g'}$. Its primary responsibility is to search through the defined $\mathcal{X}$ and $\mathcal{F}$ subspaces to identify optimal pairings (x^*, $f^* \mid b'$) as follows:

$\mathcal{X}$ subspace. To define the exits' search space, we characterize the total *number* of exits and their *positions* as search parameters. In practice, present-day backbone structures (as those from AttentiveNAS) constitute M sequential computing neural blocks (i.e., an aggregation of interrelated layers) between which effective placement of the exits can be realized. We illustrate this in Figure 2.2 through how the $\mathcal{X}$ subspace is conditioned on a $b \in \mathcal{B}$. Specifically, we define a vector of indicators $[\mathcal{I}_1, \mathcal{I}_2, ..., \mathcal{I}_{M-1}]$ where $\mathcal{I}_i \in \{0, 1\}$ to indicate whether exit branch at position i is sampled for the corresponding instance. Regarding the composition of exit branches, we fix a simple structure across all potential exits positions for three reasons: (*i*) Re-usability as such a straightforward structure can act as a base module compatible with numerous backbone model architectures and classes, (*ii*) The smaller search space size of the exits leads to smaller search overheads – especially relevant when considering the additional subspaces as well, and (*iii*) Minimizing the training costs of the exits. Our exit structure constituted a single sequential computing block of a convolutional, batch normalization and activation layers, followed by a final classifier layer.

Exits Training. Once a b' is mapped to the IOE, every $x \in \mathcal{X}$ needs to be trained for a fair evaluation of the exit candidates. In this scheme, the weight parameters of b' are kept *frozen* independent of the exits' training procedure, where the rationale here is to avoid negatively influencing the performance of b' with regards to its static accuracy score (i.e., the backbone accuracy) – which can occur when the weights are optimized for more than

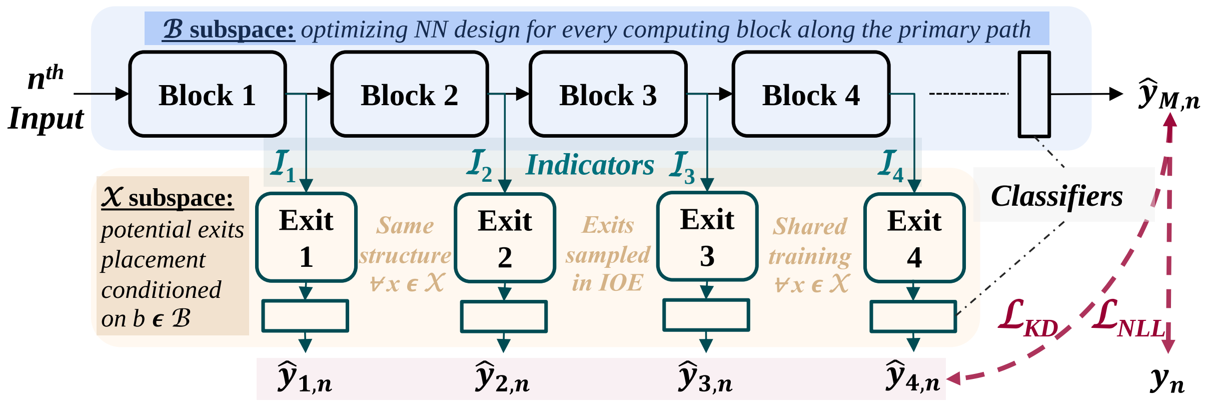

$\mathcal{B}$ subspace: optimizing NN design for every computing block along the primary path
nth Input
Block 1
Block 2
Block 3
Block 4
$\hat{y}_{M,n}$
I_1
I_2
Indicators
I_3
I_4
$\mathcal{X}$ subspace: potential exits placement conditioned on b ϵ $\mathcal{B}$
Exit 1
Same structure ∀x ϵ $\mathcal{X}$
Exit 2
Exits sampled in IOE
Exit 3
Shared training ∀x ϵ $\mathcal{X}$
Exit 4
Classifiers
$\mathcal{L}_{KD}$
$\mathcal{L}_{NLL}$
$\hat{y}_{1,n}$
$\hat{y}_{2,n}$
$\hat{y}_{3,n}$
$\hat{y}_{4,n}$
y_n

Figure 2.2: The combined $\mathcal{B}$ and $\mathcal{X}$ search spaces

one objective [186]. Combining this notion with the compact structure of the exits, the exits' training overheads can be kept to a minimum within the IOE, all while leveraging the representational power of b' across its various stages to attain the desired resource efficiency.

For the training loss function itself, we adopt a hybrid loss function ($\mathcal{L}_{total}$) combining the Negative log-likelihood ($\mathcal{L}_{NLL}$) and knowledge distillation ($\mathcal{L}_{KD}$) loss components to simultaneously train every $x \in \mathcal{X}$ as follows:

$$\mathcal{L} = \frac{1}{N}\sum_{n=1}^{N}[\frac{1}{M\text{-}1}\sum_{m=1}^{M-1}(\mathcal{L}_{NLL}(y_n, \hat{y}_{m,n}) + \mathcal{L}_{KD}(\hat{y}_{m,n}, \hat{y}_{M,n})] \tag{2.4}$$

where N is the total number of training samples and $M-1$ is the total possible number of exits. For the $\mathcal{L}_{NLL}$ term, it aggregates the losses from every exit at m when comparing its predicted outputs, $\hat{y}_{m,n}$, against the ground truth labels, y_n, for every sample n. Whereas the $\mathcal{L}_{KD}$ term aggregates the losses from comparing the error between every $\hat{y}_{m,n}$ and that of the final model classifier, $\hat{y}_{M,n}$. Due to space limitations, we illustrate how these loss components are defined in Figure 2.2, and refer interested readers to [150] for more details.

$\mathcal{F}$ subspace. The hardware search space entails the DVFS configurations for enhancing the DyNN's resource efficiency from the HW's perspective. Given how different computational workloads utilize the underlying hardware components differently, DyNN design candidates can attain maximal resource efficiency at different DVFS settings. In practice, edge devices constitute heterogeneous computing units that support DVFS features. Thus, depending on the underlying hardware, the operational frequencies of CPU, GPU, and External Memory Controllers (EMC) can be used to construct $\mathcal{F}$.

($\mathcal{X}$, $\mathcal{F}$) Evolutionary Search. Similar to the OOE, an IOE also operates an evolutionary NSGA-II algorithm to navigate the combined search spaces of $\mathcal{X}$ and $\mathcal{F}$. With each generation, a population $\mathcal{P}_{\mathcal{X},\mathcal{F}}$ is generated from the combined subspaces' encoding and provided for the dynamic fitness evaluation:

$$\mathcal{D}(x, f \mid b') = \frac{1}{\sum_{i=1}^{M\text{-}1} \mathcal{I}_i} \sum_{i=1}^{M\text{-}1} \mathcal{I}_i \cdot [score_i] \tag{2.5}$$

$$s.t.\;\; score_i = \mathcal{N}_i * \frac{E_{x_i,f}}{E_b} * \frac{L_{x_i,f}}{L_b} * (dissim_i)^{\gamma} \tag{2.6}$$

where equation (2.5) reflects the mean dynamic performance score of a sampled dynamic model (x, $f \mid b'$) through averaging scores for every sampled exit (recall $\mathcal{I}_i \in \{0, 1\}$). An exit's score is given by $score_i$ in equation 2.6, which constitutes: $\mathcal{N}_i$, the fraction of samples that can be correctly classified at exit i; $\frac{E_{x_i,f}}{E_b}$, as the normalized dynamic energy at exit x_i and DVFS settings f relative to the backbone energy consumption; $\frac{L_{x_i,f}}{L_b}$ is similarly the normalized dynamic latency term. $(dissim_i)^{\gamma}$ is a regularization term with a trade-off parameter γ measuring the dissimilarity of exit x_i and its preceding ones as:

$$dissim_i = 1 - \max\left(\mathcal{N}_{0:i-1}\right) \tag{2.7}$$

where x_i's score is regularized in proportion to the fraction of samples that can be already classified by its preceding exits. The rationale behind this metric is to: (i) avoid sampling exits of similar performance characterizations, and (ii) realize a compact decision space for the DyNN when deployed.

Based on the $\mathcal{D}$ scores, every $(x, f \mid b') \in P_{\mathcal{X},\mathcal{F}}$ is also ranked using the NSGA-II non-dominated sorting algorithm so as to realize subset $P'_{\mathcal{X},\mathcal{F}} \subset P_{\mathcal{X},\mathcal{F}}$ that would then undergo *mutation* and *crossover* for the following generation. This loop cycle continues until the final generation where a 2-D Pareto optimal set $(x^*,\ f^* \mid b')$ is returned to resume the OOE.

2.3.3 Runtime Controller

When a DyNN design is chosen for the final deployment, a runtime controller needs to be implemented to provide the effective input-to-exit mapping policies needed for dynamic inference. Concerning HADAS, its architectural optimizations are applied at the design stage of DyNNs under *ideal* mapping policies, that is, when every input is mapped to the first exit module x_i that can classify it correctly. This is evident through how the score of each exit in eq. (2.6) is scaled based on $\mathcal{N}_i$ – the *true* fraction of correctly classified samples. Thus, models from HADAS are compatible with any class of runtime controllers existing in the literature (e.g., entropy-based [69, 144, 186]).

2.3.4 Evaluation and Results

For evaluation, we implement HADAS on top of the AttentiveNAS framework [194]. To construct $\mathcal{B}$, we reuse their search space which contains more than 2.94×10^{11} neural networks generated by scaling different dimensions as stated in Table 2.1. Our experiments are conducted on the CIFAR-100 dataset where the pretrained supernet of AttentiveNAS has been fine-tuned accordingly. Backbones and baselines are all sampled from the same fine-tuned supernet. We dynamically generate the exits' search space $\mathcal{X}$ according to the supported depth (l) of the backbones in $\mathcal{B}$. In our case, potential exit positions occur at a layer-wise granularity starting from the fifth layer to the backbones' last layer (For AttentiveNAS [194], potential exit positions are set after their "MBConv" layers).

Table 2.1: Details on HADAS joint search spaces in our experiments

Decision variables	**Values**	**Cardinality**
Backbone Search Space ($\mathcal{B}$)		
Number of blocks (n_block)	7	1
Input resolution (res)	{192, 224, 256, 288}	4
Block depth (l)	{1, 2, 3, 4, 5, 6, 7, 8}	8
Block width (w)	[16, 1984]	16
Block kernel size (k)	{3, 5}	2
Block expand ratio (er)	{1, 4, 5, 6}	4
Exits Search Space ($\mathcal{X}$)		
Number of exits (nX)	$[1, (\sum_{i=1}^{nb} l_i) - 5]$	max(nX)
Exit positions (posX)	$[5, \sum_{i=1}^{nb} l_i)]$	$\binom{nx}{\sum_{i=1}^{nb} l_i}$
DVFS Search Space ($\mathcal{F}$)		
GPU frequency (AGX Volta GPU)	[0.1GHz, 1.4GHz]	14
CPU frequency (Carmel ARM v8.2 CPU)	[0.1GHz, 2.3GHz]	29
GPU frequency (TX2 Pascal GPU)	[0.1GHz, 1.4GHz]	13
CPU frequency (NVIDIA Denver CPU)	[0.3GHz, 2.1GHz]	12
EMC frequency (AGX SOC)	[0.2GHz, 2.1GHz]	9
EMC frequency (TX2 SOC)	[0.2GHz, 1.8GHz]	11

We evaluate our approach on 4 different hardware combinations from NVIDIA Edge devices: a) ***AGX Volta GPU***, b) ***Carmel ARM v8.2 CPU***, c) ***TX2 Pascal GPU***, and d) ***NVIDIA Denver CPU***. For each hardware setting, we leverage the supported DVFS configuration settings to generate $\mathcal{F}$ as in Table 2.1. Regarding the optimization process, we fix a budget of 450 iterations for the OOE and 3500 iterations for the IOE, where #iterations $= \mathcal{G} \times \mathcal{P}$. We use a cluster of 32 GPUs to train the exits for every sampled backbone, taking up to $\sim$ 8-10 GPU hours for each $\mathcal{G}$. In our experiments, we used a HW-in-the-loop setup for latency and energy measurements which pushed the overall search time of HADAS to $\sim$2-3 GPU days. Nevertheless, based on our analysis, HADAS's search overhead can be reduced to 1 GPU day if a proxy model replaced the HW-in-the-loop setup.

OOE Analysis: The top row of Figure 2.3 compares the static performance results from the OOE of HADAS against those of the top models from AttentiveNAS [194] (denoted as [**a0-a6**]). As shown, our obtained Pareto fronts (PF) generally dominate the baselines on the four hardware settings. Furthermore, HADAS can identify comparable backbones to the baselines with just a few evaluations. For instance, on the AGX Volta GPU, **a6** is dominated by another backbone from HADAS with an energy reduction of $\sim$ 33% under the

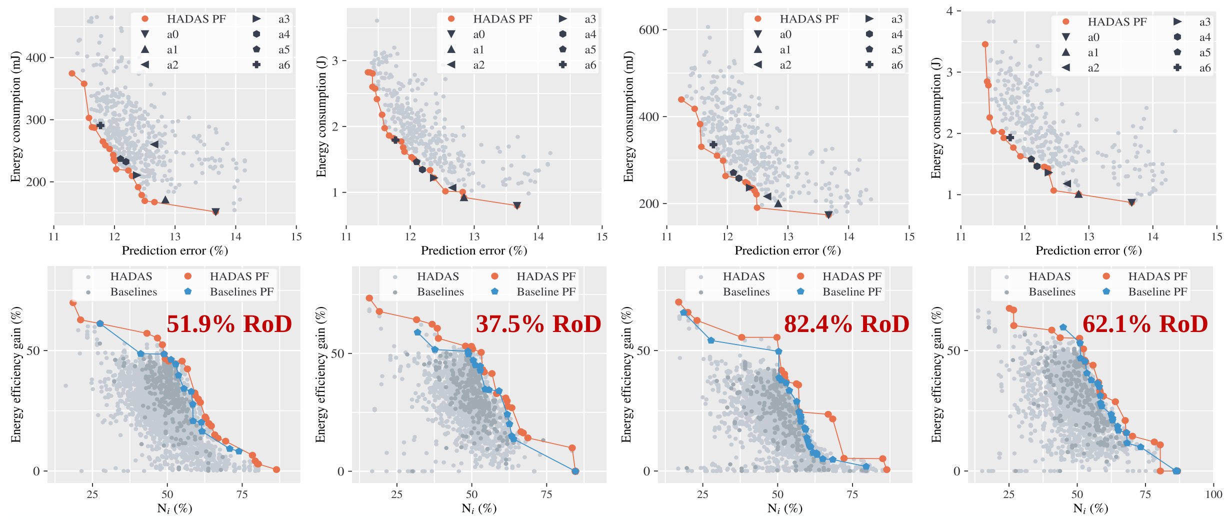


Figure 2.3: The top row gives the results of the outer optimization on 4 hardware settings of (*from left to right*): a) AGX Volta GPU, b) Carmel ARM v8.2 CPU, c) TX2 Pascal GPU, and d) NVIDIA Denver CPU. The bottom row shows the results of the inner optimization engine, with the same hardware settings. The points in the top row depict the static performance of the explored backbone neural networks in ($\mathcal{B}$) by the OOE, without early-exit or DVFS. The points in the bottom row represent the performance of the explored combinations of backbones, early-exits, and DVFS in ($\mathcal{B}, \mathcal{X}, \mathcal{F}$) by the IOE.

same accuracy level. Similarly, **a1** is dominated by another backbone from HADAS with an accuracy improvement of **2.34%** under the same energy gain.

IOE Analysis: The results of the IOE are shown in the bottom row of Figure 2.3. For a fair comparison, we fix the same optimization budget when running the IOE for the baselines and HADAS. The dynamic performance of the explored (b, x, f) combinations and the obtained Pareto fronts are given for both approaches, where the dynamic comparison metrics are the energy efficiency gains when early exiting and DVFS are supported, as well as the average of N_i values from equation (2.6). Across the four hardware settings, HADAS seemingly dominates the majority of the optimized baselines with an average ratio of dominance **58.4%** (detailed in the following paragraph). This can be attributed to HADAS's better understanding of the global search space, where it samples backbones that are more poised to benefit from the IOE optimizations with regard to early exiting and DVFS. This is also evident through how HADAS can sample dynamic parameters for its models that can

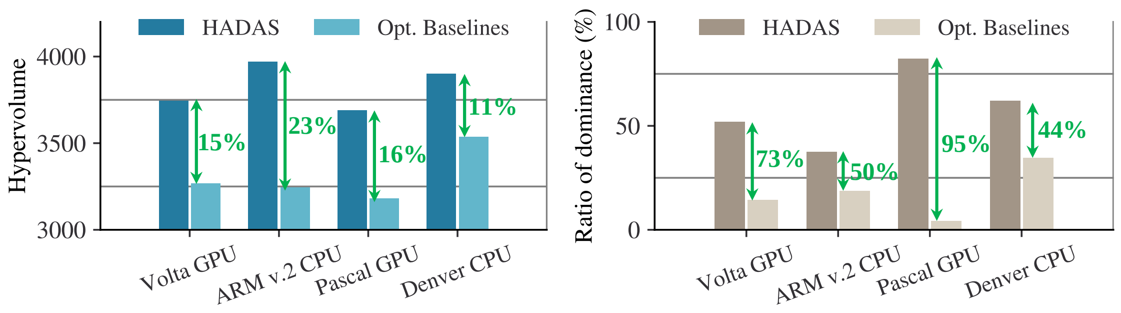


Figure 2.4: Comparing search efficacy for HADAS and the optimized baselines with regards to: a) hypervolume (*left*) and b) ratio of dominance (*right*)

Table 2.2: DyNNs Comparison using the TX2 Pascal GPU

Model	**Baseline Acc(%)**	**EEx Acc(%)**	**Baseline Ergy(mJ)**	**EEx Ergy(mJ)**	**EEx_DVFS Ergy(mJ)**
AttentiveNAS_a0	86.33	89.95	**173.78**	**119.83**	**116.14**
AttentiveNAS_a6	**88.23**	**93.02**	335.48	256.80	218.34
HADAS_b1	**87.34**	**93.16**	**212.44**	**119.84**	**93.78**
HADAS_b2	88.06	91.83	341.3	187.92	126.06
HADAS_b3	86.54	88.31	205.48	130.20	86.84
HADAS_b4	**88.40**	89.24	358.01	232.77	201.01

realize substantial energy or accuracy gains near the extremes of its Pareto frontier, which are not realizable by the optimized baselines. For instance on the Caramel ARM v8.2 CPU, energy gains reach **63%** for one of the extreme dynamic models on the Pareto frontier of HADAS, compared to **52%** for the extreme dynamic variant from the optimized baselines, under the same level of accuracy.

Hypervolume (HV) and Ratio of Dominance (RoD): we expand further on the IOE analysis and leverage *hypervolume (HV)* and *ratio of dominance (RoD)* as comparative evaluation metrics. The former metric measures the volume of the dominated portion of the objective space, whereas the latter measures the percentage of solutions found by HADAS that dominate the optimized baselines (and vice-versa). Figure 2.4 shows that HADAS consistently outperforms the optimized baselines with regards to both metrics across the 4 hardware platforms. Taking the Pascal GPU as an example, we find that the *HV* coverage and *RoD* are **16%** and **95%** more for HADAS over the optimized baselines, respectively.

DyNNs comparison: In Table 2.2, we compare the top DyNNs obtained by HADAS with

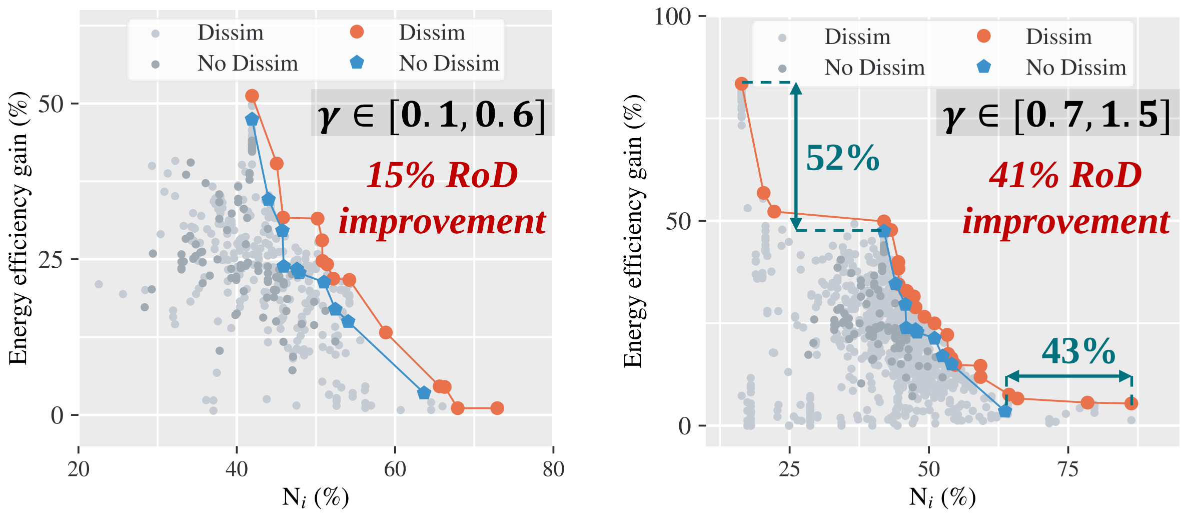


Figure 2.5: Inner optimization improvement by regularizing the exits scores with the dissimilarity function $(dissim)^\gamma$ over two ranges of γ values

two AttentiveNAS models: **a0**, the most energy-efficient baseline, and **a6**, the most accurate baseline. Models are compared with regards to their *static* (i.e., baseline accuracy and energy) and their *dynamic* performances (i.e., accuracy and energy with early exiting and DVFS). As shown, the optimal models from HADAS outperform the baselines of AttentiveNAS in both *static* and *dynamic* evaluations. For instance, **b1** from HADAS is **57%** and **19%** more energy-efficient than the **a6** and **a0**, respectively, while enjoying similar accuracy scores like the most accurate model **a6**.

Dissimilarity Ablation Study. We perform an ablation study to investigate the impact of the dissimilarity term ($dissim^\gamma$) in equation (2.6) through the performance of the explored models under each case. Specifically, we run the IOE for one backbone twice, with $dissim^\gamma$ not included and one when it is included. In Figure 2.5, we compare the results obtained with and without the dissimilarity with different values of γ. As shown, the inclusion of the dissimilarity term allows the optimization algorithm to focus more on exploring dissimilar early exits with a high contribution to the prediction accuracy, improving RoD by 41%.

2.4 Map-and-Conquer: Efficient Mapping of Dynamic Neural Nets onto Heterogeneous MPSoCs

We model how to conduct a static-to-dynamic transformation of a $\mathcal{NN}$, designating it as a multi-stage network along the width dimension (channels/attention heads), and characterize its performance overheads when deployed in a distributed manner on heterogeneous MPSoCs.

2.4.1 Dynamic Transformation of NNs on MPSoC

Consider an unaltered basic neural network, $\mathcal{NN}$, constituting a sequence of n layers:

$$\mathcal{NN} = \mathcal{L}^n \circ \mathcal{L}^{n-1} \circ ... \circ \mathcal{L}^1 \tag{2.8}$$

each computing layer, L^j, consists of weight parameter matrices whose count represents the 'width' of the layer. Without losing generality, we refer to these weight matrices here as 'channels', such as those in a convolutional $\mathcal{NN}$. Therefore, we can define the j^{th} layer as:

$$\mathcal{L}^j = \{C_1^j, C_2^j, \cdots, C_W^j\} \tag{2.9}$$

in which C_i^j represents the i^{th} channel in the j^{th} layer. Now, consider an SoC that comprises M computing units $\mathbb{CU} = \{\mathcal{CU}_1, \mathcal{CU}_2, ..., \mathcal{CU}_M\}$, the goal is to devise a strategy to partition every L^j into M subsets according to its width dimension (i.e., the channels), and thus, $\mathcal{L}^j$ is redefined as:

$$\mathcal{L}^j = \{l_1^j, l_2^j, ..., l_M^j\} \tag{2.10}$$

which enables every contiguous subset of channels, l_m^j, to be mapped onto one of the computing units, $\mathcal{CU}_m \in \mathbb{CU}$. In this sense, we define two operations to characterize this mapping problem: (i) ***Partitioning***; to divide layers and generate the subsets l_m^j, and (ii) ***Concatenation***; to reuse the generated intermediate features, F_m^j, in set of the immediate next layer in all subsequent stages, $\{l_{m+1:M}^{j+1}\}$. In accordance, we define two parameter matrices to characterize these operations:

$$\mathbb{P} = \begin{bmatrix} p_1^1 & \cdots & p_1^n \\ \vdots & \ddots & \vdots \\ p_M^1 & \cdots & p_M^n \end{bmatrix}, \mathbb{I} = \begin{bmatrix} I_1^1 & \cdots & I_1^n \\ \vdots & \ddots & \vdots \\ I_M^1 & \cdots & I_M^n \end{bmatrix} \tag{2.11}$$

where $\mathbb{P}$ is the *partitioning matrix* in which every p_i^j represents the fraction of channels in a layer L^j (equation 2.9) are to be assigned to l_i^j. $\mathbb{I}$ is an *indicator matrix* in which $I_i^j \in \{0,1\}$ indicates whether the intermediate features, F_i^j, are to be used in the $j+1$ layers in the following stages. Figure 2.6 provides an illustration for how these matrices govern the partitioning and concatenation operations of a neural network. As shown, each $\mathcal{CU}_m$ on the SoC can host a unique sequence of channel subsets, which we denote as a stage, S_i:

$$S_i = l_i^n \circ l_i^{n-1} \circ ... \circ l_i^1 \tag{2.12}$$

and ultimately, we obtain the following set of stages:

$$\mathbb{S} = \{S_1, S_2, ..., S_M\} \tag{2.13}$$

if we augment each stage S_i with an exit at its tail (e.g., a classifier layer), each stage can now act as a *separate* inference sub-model, to be invoked based on some established runtime criteria during deployment (e.g., input processing difficulty).

Lastly, we define an additional vector, $\mathbb{M}$, to parameterize the mapping of stages onto the

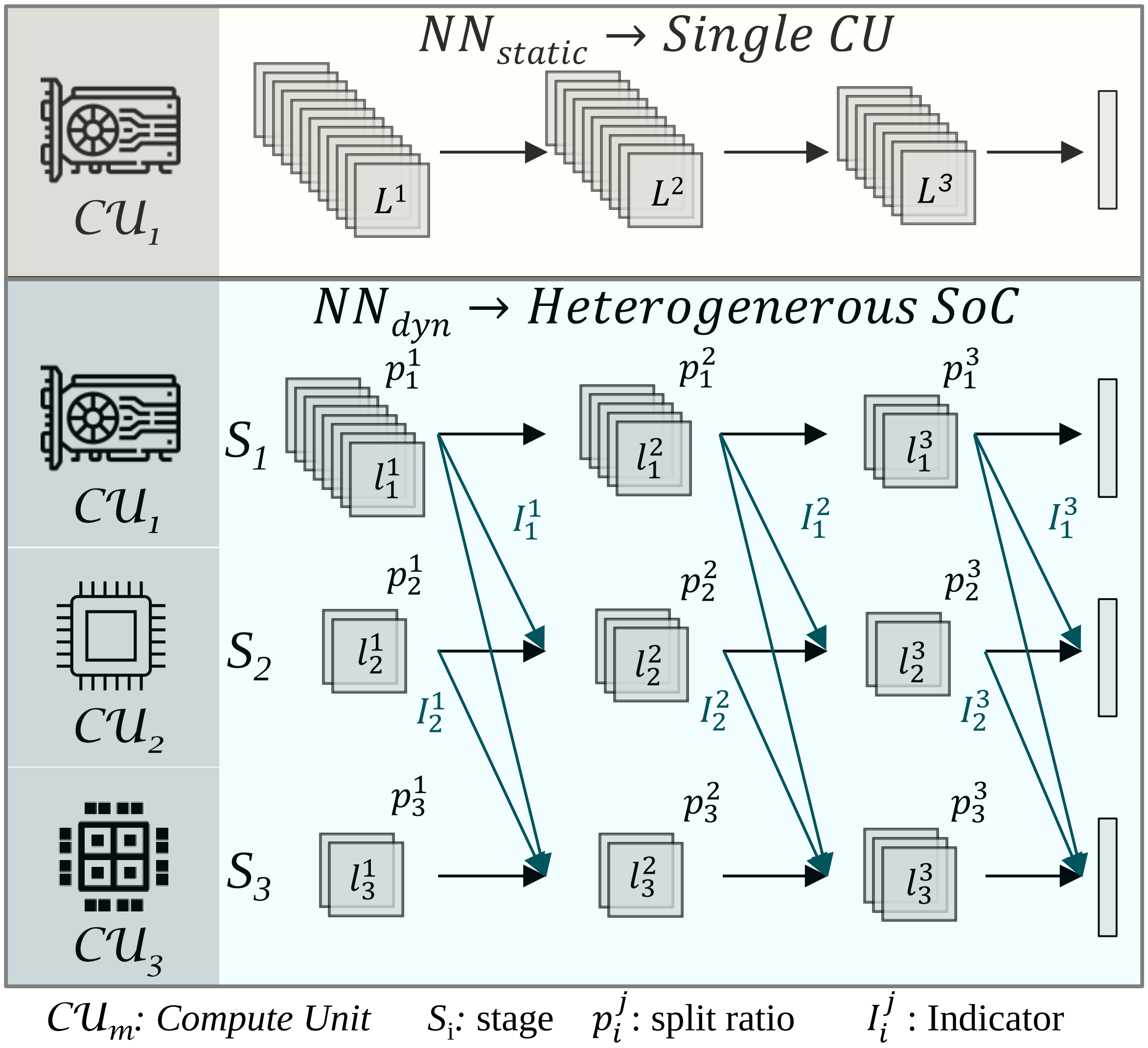


Figure 2.6: Transformation of NN_{static} into NN_{dyn} and mapping NN_{dyn} onto multiple $\mathcal{CU}$s

SoC: $S_i \rightarrow \mathcal{CU}_m \ \forall \ S_i \in \mathbb{S}, \mathcal{CU}_m \in \mathbb{CU}$. $\mathbb{M}$ can by given as:

$$\mathbb{M} = [\pi_1, \ldots, \pi_M] \ \ s.t. \ \pi_k \neq \pi_{k'} \ \forall \ 1 \leq k \leq k' \leq M \tag{2.14}$$

in which every entry π_k is one $\mathcal{CU}_m \in \mathbb{CU}$ to whom S_k is mapped. The condition is for enforcing that no two stages are mapped onto the same $\mathcal{CU}_m$.

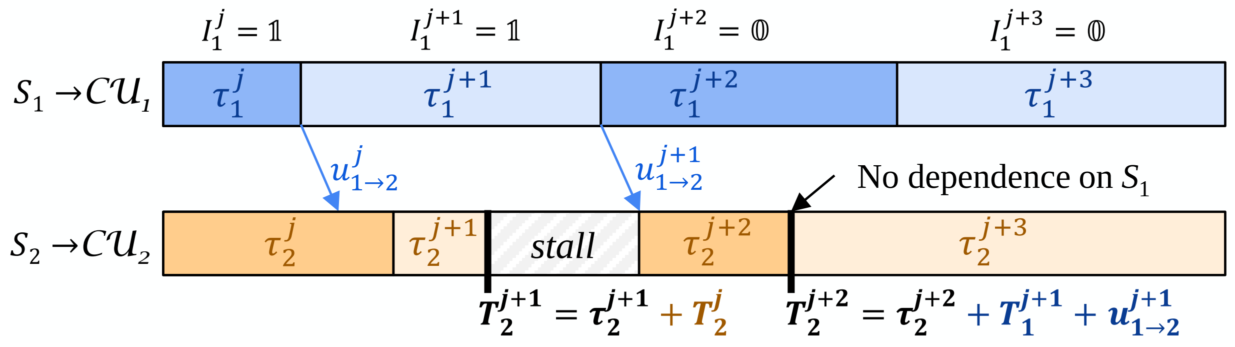


Figure 2.7: Concurrent execution of S_2 and S_1 considering timing dependencies

2.4.2 Distributed Performance Modelling for Dynamic Inference

Here, we model the dynamic inference execution overheads given the partitioned deployment of a model on a heterogeneous MPSoC with regards to *latency* and *energy consumption*. Given the scope of this work, we assume *ideal input mapping* in which the number of stages needed to process an input sample i is known apriori. In practice, input mappings can be determined using runtime controllers as those stated in [22].

Execution Latency. Let τ_i^j denotes the execution latency overhead of sublayer l_i^j in S_i. We first aim to derive an expression for the latency overhead of every stage, denoted by T_{S_i}. At this point, we highlight that stages are indexed by the order of their execution. For example, S_2 is only instantiated if S_1 is deemed insufficient to terminate the processing. Thus, there exists inter-stage dependencies of S_i on its predecessors $S_{1:i-1}$ (as indicated by I_i) whose overheads need to be accounted for, especially when stages are mapped onto different hardware units.

To avoid the demerits of a sequential execution model, we leverage the underlying separation of the compute units and propose a *concurrent* model of execution that considers these dependencies. Specifically, any sublayer l_i^j in an 'instantiated' S_i can immediately proceed to execute its inputs once all of its required input features, $\{(F_{1:i-1}^{j-1} \cdot I_{1:i-1}^{j-1}) \cup F_i^{j-1}\}$, are readily available within its local vicinity. From here, we can give the *cumulative* latency overhead at l_i^j by:

$$T_i^j = \tau_i^j + \max\{T_i^{j-1}, T_k^{j-1} + u_{k\to i}^{j-1} \mid I_k = \nVdash \forall\, 1 \leq k < i\} \tag{2.15}$$

where the second term captures the maximum cumulative latency experienced in a previous layer from all stages preceding S_i. Thus, T_i^j captures the cumulative latency estimate in stage i at j while accounting for inter-stage dependencies, while $u_{k\to i}^{j-1}$ is the data transmission overhead of the features F_k^{j-1} to the local buffer of the computing resource assigned to S_i (See Figure 2.7 for an illustrative example). Given n layers in S_i, the execution latency of S_i can be estimated:

$$T_{S_i} = T_i^n \tag{2.16}$$

Energy Consumption. For every $\mathcal{CU}_m \in \mathbb{CU}$, we first characterize its power consumption:

$$P_m = P_m^s + P_m^d(\vartheta_m) \approx \alpha + \beta \cdot \vartheta_m \tag{2.17}$$

P_m^s and P_m^d are the static and dynamic components, respectively. The latter is parameterized by the scaling factor ϑ_m based on the supported DVFS features on $\mathcal{CU}_m$, where α_m and β_m are constants. From here, the energy required to complete processing at sublayer l_i^j during inference is given by:

$$e_i^j = \tau_i^j \cdot P_m \tag{2.18}$$

and as such the total energy consumed by S_i is:

$$E_{S_i} = \sum_{j=1}^{n} e_i^j \tag{2.19}$$

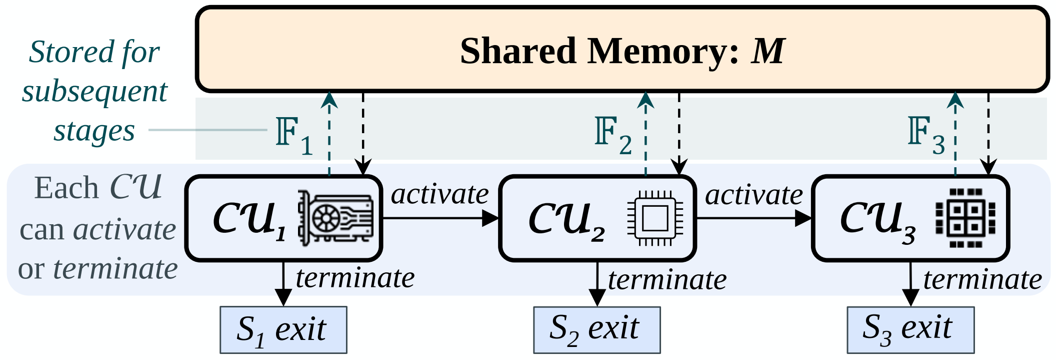


Figure 2.8: Illustration of data movement and feature storage on the MPSoC

Overall Characterization. Under the concurrent model of execution, the overall performance characterization is given by the following two equations:

$$T_{\mathbb{P},\mathbb{I},\mathbb{M},\vartheta} = \max\{T_{S_i}\ \forall\ S_i \in \mathbb{S}\} \tag{2.20}$$

$$E_{\mathbb{P},\mathbb{I},\mathbb{M},\vartheta} = \sum_{i=1}^{M'} E_{S_i}\ s.t.\ 1 \le i \le M' \le M \tag{2.21}$$

where for a dynamic inference on a MPSoC, described through the parameters choices of $(\mathbb{P}, \mathbb{I}, \mathbb{M}, \vartheta)$, its execution latency is the *maximum* from all its stages due to concurrency, whereas its energy consumption is the *aggregation* of energy consumed by the M' 'instantiated' stages to process an input sample.

2.4.3 Problem Formulation

Let $\Pi = (\mathbb{P}, \mathbb{I}, \mathbb{M}, \vartheta)$ combine all parameters that characterize a neural network's mapping onto an MPSoC. Our main optimization goal is to find the ideal parameters that can enhance a performance objective, $\mathcal{P}$, given a set of constraints:

$$\Pi^* = \min_{\Pi} \mathcal{P}(\Pi) \tag{2.22}$$

$$s.t.\ T_{\Pi^*} < T^{TRG},\ \ E_{\Pi^*} < E^{TRG},\ \ size_{\Pi^*}(\mathbb{F}, \mathbb{I}) < M$$

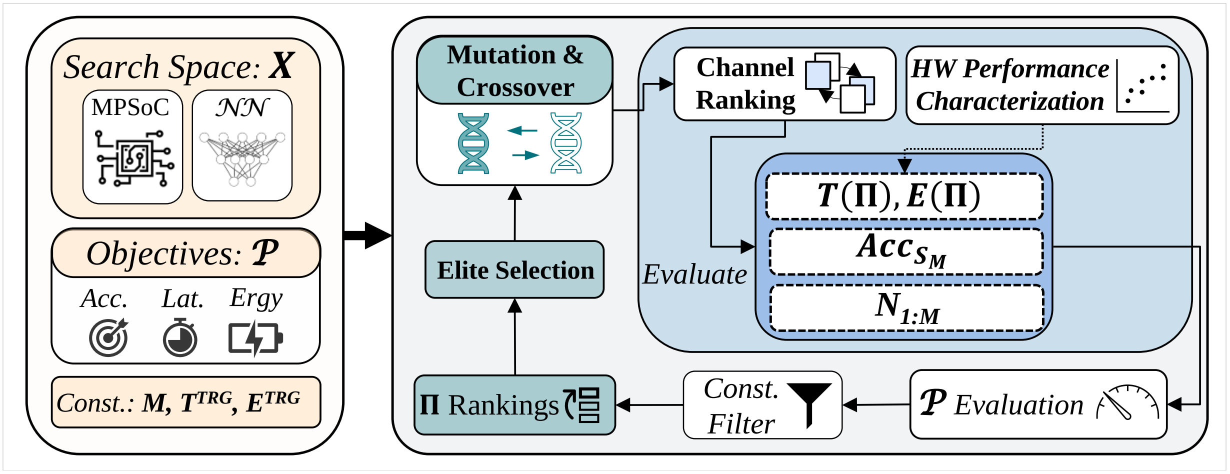


Figure 2.9: Overview of our proposed optimization framework

where T^{TRG} and E^{TRG} are the respective target latency and energy constraints as set by the practitioner. The constraint $size_{\Pi}(\mathbb{F}, \mathbb{I}) < M$ is to bound the size of the intermediate features that need to be made readily available for the duration of the inference (denoted as $\mathbb{F}$), for they are limited by the MPSoC's shared memory size, M (see Figure 2.8). $\mathcal{P}$ is kept generic and can be tuned to the designers' objectives.

2.4.4 Map-and-Conquer Framework

We propose the Map-and-Conquer optimization framework to solve the mapping problem. Figure 2.9 gives an overview of our framework, whose key components are detailed below.

Search Space. Here we describe how to generate a search space, X of mapping strategy parameters, namely the space of $(\mathbb{P}, \mathbb{I}, \mathbb{M}, \vartheta)$. Firstly, given a pretrained $\mathcal{NN}$ and an MPSoC with M CUs, we can generate X based on the $\mathcal{NN}$'s layer specifications and the MPSoC's underlying hardware composition. For the former, the attainable depth and width parameters of every layer $L^j \in \mathcal{NN}$ define the $(\mathbb{P}, \mathbb{I})$ parameter matrices. For the latter, $M = |\mathbb{CU}|$ specifies its mapping space and the total number of inference stages. Lastly, ϑ is specified through the hardware reconfiguration parameters $(DVFS)$. For instance, the mapping search space complexity of one layer from the *Visformer* [34] is $\mathcal{O}(1.5 \times 10^5) = \mathcal{O}(8^3 \times 3! \times 50)$, considering 8 channel partitioning ratios, $M = 3$, and $|\vartheta| = 50$.

Performance Objectives. Next, a performance objective needs to be designated as $\mathcal{P}$ for the main optimization function in equation (2.22), to be specifically used for the candidate mapping evaluation. For our case, we used the following expression for $\mathcal{P}$:

$$\mathcal{P} = (\frac{Acc_{base}}{Acc_{S_M}}) \times (\sum_{i=1}^{M} T_{S_i} \cdot N_i) \times (\sum_{i=1}^{M} E_{S_{1:i}} \cdot N_i) \tag{2.23}$$

In which Acc_{base} is the baseline accuracy of the pretrained $\mathcal{NN}$ model; Acc_{S_M} is the accuracy of the last stage of the dynamic version of $\mathcal{NN}$ as its base accuracy. The aforementioned terms are included to ensure that no accuracy drops ensue when a model's structure changes through the $\mathbb{I}$ matrix. N_i is the number of input samples -from the validation dataset- correctly classified at S_i, given that every prior stage misclassifies them. T_{S_i} is the latency experienced by the MPSoC at stage S_i based on equation (2.16); $E_{S_{1:i}}$ is the energy consumed as a result of executing i stages of the model – each E_i is evaluated as in equation (2.19).

Search Algorithm. We develop an evolutionary-based algorithm to effectively explore the search space. Following the workflow in Figure 2.9, every new search iteration entails a new population, say $X'_i \subset X$. Then for every configuration $\Pi \in X'$, its corresponding dynamic $\mathcal{NN}$ and hardware settings are evaluated using a predefined objective function, $\mathcal{P}$. Based on results, configurations that do not meet the search constraints (e.g., memory usage) are omitted, whereas the remaining ones are ranked according to $\mathcal{P}$, and a subset of elite configurations is taken for a mutation and crossover stage to obtain the new population X'_{i+1}. Once the search budget expires, a Pareto set in calculated from all the generated populations from which the ideal dynamic mapping strategy is extracted.

Channel Partitioning and Reordering. Before a candidate configuration $\Pi \in X'$ is evaluated on the objective function P, the $\mathcal{NN}$ should be partitioned according to the ratios in $\mathbb{P}$. Yet to maximize performance when partitioning, the width channels in each model layer are arranged according to their *degree of importance*. The logic being that given the sampled partitioning matrix $\mathbb{P}$ for a configuration Π, it would be beneficial to assign the most important channels in the layer to the earlier inference stages for dynamic inference. This would enable numerous samples to terminate processing prematurely if deemed feasible, which will consequently aid in enhancing the *dynamic inference* performance of the $\mathcal{NN}$ with regards to experienced latency and energy on the MPSoC. This reordering method is feasible as all channels within the same layer share the same dimensions. Channel ranking is widely used in pruning, and we follow the approach in [131] to estimate each channel's importance.

Performance Evaluation. Once a model is transformed to its dynamic version through $\mathbb{P}$ and $\mathbb{I}$, the hardware measurements needed for the performance evaluation of each $\mathcal{NN}$ in equation (2.23) need to be estimated for each input sample. One way to achieve this is through surrogate models, which are able to predict τ_i^j and e_i^j of each layer j mapped onto stage i (also CU i) based on input configurations while abiding by any inter-stage execution dependencies, and taking into account the computation cost and feature map communication overheads. Hence, a predictor (XGBoost [33] in our case) is first trained on a benchmarked dataset of diverse layer specifications, deployment hardware and DVFS settings. Afterwards, the predictor is deployed to characterize the performance of each model sampled within the population, providing estimates for its base latency, τ_i^j, and energy consumption, e_i^j. In our case, we use the TensorRT library to first evaluate performance overheads on a layer-wise granularity, construct the dataset, and then deploy the predictor for hardware evaluations.

2.4.5 Evaluation and Results

Our experiments are conducted on the MPSoC provided by NVIDIA: *Jetson AGX Xavier*. This platform embeds CPU, GPU, and DLA cores on the same chip, sharing the same system

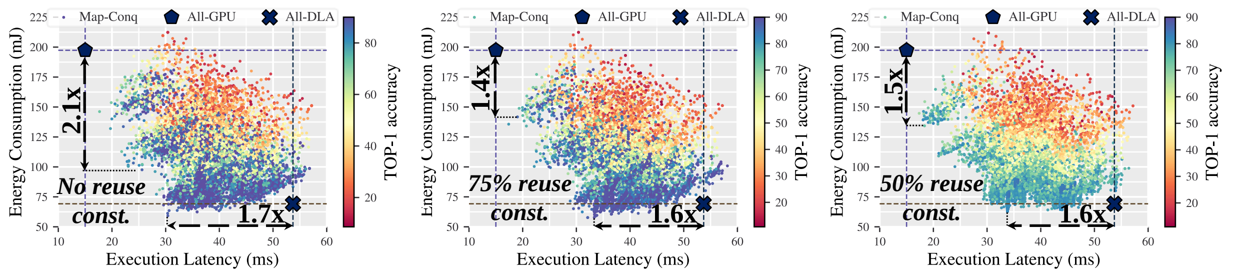


Figure 2.10: Results of three different search strategies: **Left)** No constraint is set on the *Fmap Reuse*. **Middle)** Under a constraint of reusing only less than 75% of feature maps. **Right)** Under a constraint of reusing only less than 50% of feature maps. All the results are reported for *Visformer* on the AGX Xavier MPSoC. In the three plots, we highlight the configurations that exhibit the highest latency-energy tradeoff while preserving less than 0.5% drop in accuracy

memory. To run the $\mathcal{NN}$ workloads on the DLA, we use TensorRT and ONNX to build inference engines from the PyTorch model. As $\mathcal{NN}s$, we use *Visformer* [34] as ViT-based architecture and *VGG19* [172] as CNN-based architecture to validate our approach for both cases. The dataset used for accuracy assessment is CIFAR100. Regarding the optimization framework, we run the optimization algorithm for 200 generations, each with a population size of 60, resulting in 12K overall evaluations. Furthermore, the evaluation step is performed on a cluster of 12 GPUs taking up to $\sim$ 10 GPU hours to run the entire optimization process.

Search Process Analysis. In this section, we analyze the results of the search process conducted by our framework under two main cases: 1) When no constraint is set to limit the feature map reuse between inference stages, 2) When only less than 75%, 50% of feature maps can be reused, respectively. In Figure 2.10, we show the optimization results for each case. Firstly, we observe that most of the explored configurations achieve a good tradeoff between DLA energy efficiency and GPU latency speedup. Furthermore, under the same baseline accuracy of *Visformer*, we notice an energy gain up to $\sim$ **2.1x** compared to the GPU-only mapping with latency $\leqq 30ms$. Similarly, a latency speedup up to $\sim$ **1.7x** compared to the DLA-only mapping, with comparable energy efficiency. Secondly, we can notice an accuracy drop of $\sim$ 6% when setting up hard constraints on the feature map reuse (See the *50% case*), underlining the importance of defining optimal inter-stages concatenation strategies

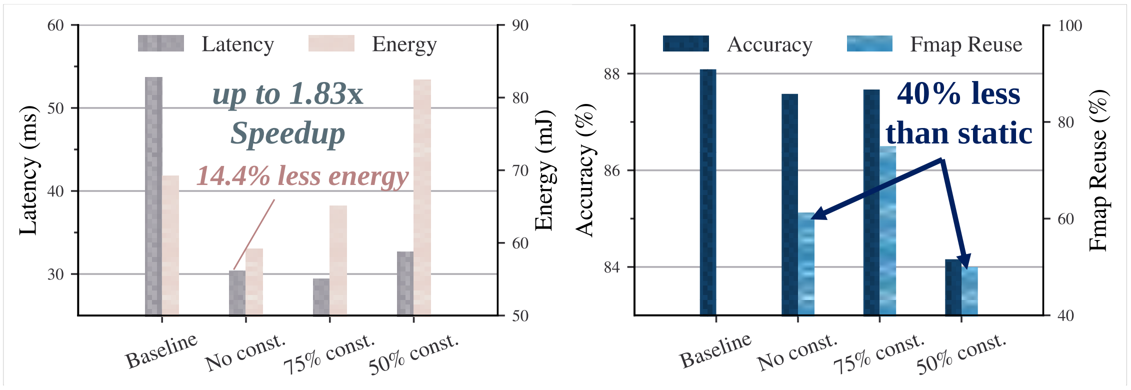


Figure 2.11: Comparison between the most energy-oriented models selected from the obtained Pareto sets by each search strategy and the baseline on DLA

in maintaining accuracy while minimizing inter-CUs dependencies.

Pareto Optimal Models Analysis. In this section, we delve further into the performance breakdown of the Pareto optimal models obtained from the three search strategies. We select the most energy-oriented models and compare them with the baseline *Visformer* mapped entirely on the DLA. Figure 2.11 and Table 2.3 detail the obtained results. By exploring neural network dynamicity and concurrency on heterogeneous CUs, our models achieve better latency-energy tradeoff, providing latency speedup of ∼ **1.83x** and up to ∼ **14.4%** of energy gain as shown in the left sub-figure. In addition, the correlation between feature maps reuse and accuracy is highlighted in the right sub-figure. Reducing the feature maps reuse across stages decreases the inter-CUs data transmission at the cost of accuracy drops. However, some models can achieve comparable accuracy to the baseline while only reusing **60%** of the necessary feature maps (See *No constr. and 75% constr.* cases)

Generalization to other architecture. To further demonstrate our approach's applicability, we evaluate our optimization framework on a typical CNN architecture, *VGG19*. Table 2.3 details the obtained results. Regarding the baseline performances, *VGG19* depicts a high energy consumption on GPU and slow execution latency on DLA. This is explained by its many weights and large feature maps, which entail high memory footprints for both CUs. Moreover, the large number of weights may exhibit a high degree of redundancy. Our

Table 2.3: Performances comparison for Map-and-Conquer models against the baselines

Opt. Strategy	**NN Implment.**	**TOP-1 Acc (%)**	**Avg. Enrg. (mJ)**	**Avg. Lat. (ms)**	**Fmap. reuse. (%)**
Visformer (ViT-based Architecture)					
None	GPU	**88.09**	197.35	**15.01**	-
	DLA		**69.22**	53.71	-
No Fmap Constr.	Ours-L	86.12	**108.44**	25.58	68.75
	Ours-E	**87.58**	**59.21**	**30.40**	**61.25**
75% Fmap Constr.	Ours-L	84.64	102.67	24.65	65.00
	Ours-E	**87.67**	**65.12**	**29.46**	75.00
50% Fmap Constr.	Ours-L	82.69	116.00	**24.51**	50.00
	Ours-E	84.16	82.44	32.70	**50.00**
VGG19 (CNN-based Architecture)					
None	GPU	**80.55**	630.11	**25.23**	-
	DLA		**164.89**	114.41	-
No Fmap Constr.	Ours-L	84.81	251.63	**25.67**	52.94
	Ours-E	84.63	153.97	**34.02**	70.58
75% Fmap Constr.	Ours-L	84.76	**247.34**	**26.07**	64.70
	Ours-E	82.64	**136.31**	37.22	**47.05**
50% Fmap Constr.	Ours-L	84.62	250.80	25.83	50.00
	Ours-E	82.53	**136.41**	37.24	**50.00**

approach has exploited these two properties of *VGG19* well, resulting in up to $\sim$ **4.62x** energy gain and $\sim$ **4.44x** latency speedup.

2.5 Concluding Remarks

We studied how the synergy of DyNNs, distributed computation mapping, and hardware reconfigurability can elevate performance of DNNs on heterogeneous MPSoCs. We developed two novel frameworks that showcased promising results in elevating performance efficiency, highlighting the merit of considering dynamic inference at design time, and motivating further exploration along this direction.

Chapter 3

A Mapping-Aware Graph Neural Architecture Search Framework for Heterogeneous MPSoC Deployment

We draw on our insights from the previous chapter and study how to design Graph Neural Networks (GNNs) for heterogeneous MPSoC deployment. The motivation for this study is threefold: (i) the rising trend of GNNs at the edge (e.g., vision in autonomous driving). (ii) GNNs' unique irregular sparse-dense computational flow. (iii) Commercial Heterogeneous MPSoC platforms being not optimized for GNN workloads. We add special emphasis on Vision GNNs (ViGs) case. The full study details and results are in our published work [137].

3.1 Introduction

Due to their inherent capacity in learning meaningful feature representations from non-Euclidean graph-structured data, the employment of Graph Neural Networks (GNNs) has extended beyond typical graph learning applications, e.g., molecular inference and social networks [204], to encompass the field of computer vision. By transforming an image structured as a regular grid of pixels into a graph, irregular and complex objects can be better captured by the more flexible graph-level features generated throughout the model architecture. As such, recent works employing GNNs to operate on this generalized form of image data have demonstrated remarkable successes across a variety of visual tasks, e.g., object detection and image classification [68, 198, 215, 216]. In fact, the application of GNNs has been further studied for more nuanced visual-based tasks in critical application settings, such as collision prediction in self-driving vehicles [121, 223].

Commercial SoC platforms, such as the Nvidia Xavier [1] and Tesla FSD [180], have successfully integrated a variety of proven hardware computing units (CUs) and industrial IPs on a single chip to balance the low-latency and energy efficiency requirements of compute-intensive workloads. Through such advanced platforms, conventional deep learning vision models (e.g., CNNs) can be run effectively in an edge computing setting to meet stringent application requirements (think real-time object detection for autonomous driving [109]). By extension, any consideration for applying GNNs for vision in the such deployment scenarios must ensure that the execution constraints are still satisfied.

However, this is challenging in the case of vision GNNs as they are characterized by an *irregular, multiphase sparse-dense* computational flow [61] compared to the regular, dense DNN workloads. Particularly, this irregularity emanates from the repeated sequence of *Aggregation* and *Combination* phases. The former employs a message-passing algorithm for feature exchange between graph vertices, exhibiting sparse kernels with random memory access patterns. The latter constitutes typical multi-layer perceptron (MLP) layer(s) for feature transformation, exhibiting dense kernels and regular access patterns. As such, the complication arises as neither the architecture of typical CUs (e.g., GPU) nor that of conventional accelerators (e.g., DLA) is designed to efficiently support this unique sequence.

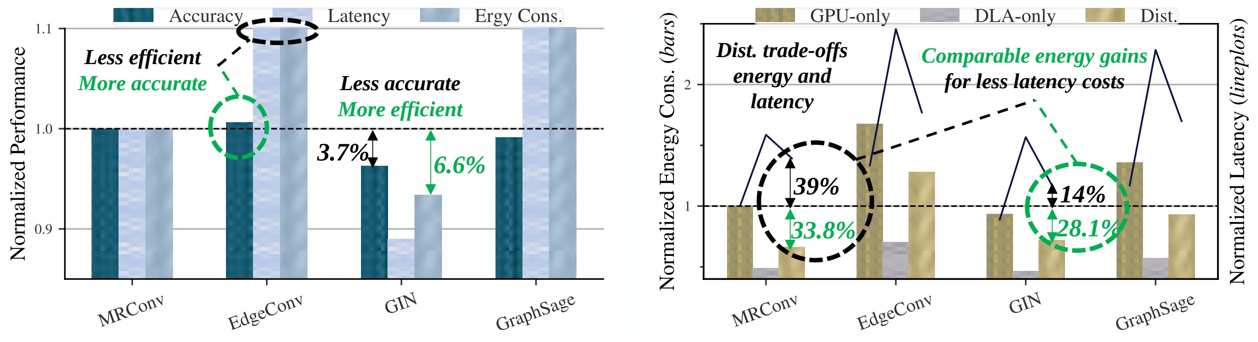


Figure 3.1: Comparing ViG model variants [68] with different graph learning operators when trained on the Oxford-Flowers dataset and deployed onto the NVIDIA Jetson AGX Xavier SoC. All values are normalized by the baseline performance evaluations incurred by the original ViG with MRConv layers when fully deployed onto the GPU only. The *left* figure shows how performance characteristics differ from one variant to the other regarding accuracy, latency, and energy consumption. The *right* figure illustrates how distributed mapping strategies across the GPU and DLA can yield different latency-energy trade-offs.

Naturally, considerable research works have dedicated efforts to design customized GNN accelerator architectures that can support the *multi-phased* computational flow [13, 30, 95, 177, 214, 218]. Unfortunately, these designs are not flexible enough to be consolidated into standard MPSoCs, for GNNs belong to a nascent, rapidly-evolving field, and SoCs silicon die area restrictions limit the number of specialized CUs that can be integrated. As vision GNN applications on the edge continue to proliferate, an alternative approach is needed.

3.1.1 Motivational Example

In Figure 3.1, we showcase the potential performance trade-offs as offered by the *architectural* and *mapping* optimization spaces for a vision GNN model when deployed onto a heterogeneous SoC. In this example, the backbone GNN architecture is the ViG-S [68], the target platform is the NVIDIA Xavier AGX SoC, and the models are trained on the Oxford-Flowers image dataset. Given how the ViG belongs to the Graph Convolutional Network (GCN) class of GNNs, we construct three (03) additional variants of the baseline ViG with different GCN operators. Specifically, the original ViG architecture employs the `Max-Relative` Graph Conv

(`MRConv`) graph operation throughout the entirety of its model, whereas the variants employ other GCN layer types, namely `EdgeConv`, `GIN`, and `GraphSage`. After training the ViG variants, we characterize their accuracy, latency, and energy consumption scores relative to the original `MRConv` ViG variant when deployed onto the NVIDIA platform. In the *left* Figure, we can observe some performance trade-offs from varying this *singular* GNN architectural setting, i.e., the GCN layer operator. For instance, the `EdgeConv` ViG variant can achieve slightly higher accuracy (0.69% more) than the `MRConv` one at the expense of a considerable increase in latency and energy consumption. Contrarily, the `GIN` operation is 6.6% more energy-efficient than `MRConv` at the expense of a 3.7% decrease in accuracy. Though there is no clear dominance for one variant over the other, this analysis sheds light on the potential performance trade-off gains from optimizing the architectural design parameters. These gains can be further compounded when considered alongside feasible deployment options. In these first experiments, only the GPU was used as the target computing unit.

In the *right* Figure, we showcase how additional performance trade-offs are attained considering the various deployment options for the ViG variants on the SoC. In this example, the considered options are *standalone* deployment on either the GPU or DLA components or *distributed* deployment across the two. We remark that the distributed deployment options follow the mapping strategies for GNN processing workloads provided by our optimization engine, detailed in a later Section. From the Figure, the straightforward observation is that for every ViG architecture, *standalone* GPU deployment is the option with the fastest execution speeds, *standalone* DLA deployment is the most energy-efficient alternative, and the distributed option compromises between the two. However, a more interesting perspective on mapping optimizations can be taken when considered part of a broader design problem. That is, combining both the *architectural* and *mapping* optimizations to achieve better performance trade-offs compared to performing optimizations for each design space in isolation. For instance, assume a designer's primary objective is to improve the ViG's energy efficiency while incurring minimal execution slowdown. From a pure resource efficiency perspective, a *distributed* mapping strategy for the `GIN` *architectural* variant can be more beneficial than directly distributing the original `MRConv` ViG workloads since the former achieves comparable energy efficiency gains to those of the latter (28.1% to 33.8%) at the expense of reduced latency costs (14% to 39%). Still, the caveat remains that the `GIN` variant is less accurate than the original ViG. Thus, the question becomes how to provide a generalized characterization of this architecture-mapping design space.

3.1.2 Novel Contributions

In light of these challenges, we make the following contributions thorugh this study.

- We study how vision GNNs can leverage distributed deployment across multiple CUs for performance efficiency when deployed onto a heterogeneous SoC.
- We present **MaGNAS**, a Mapping-aware Graph Neural Architecture Search Framework for *co-optimizing* the design of ViG architectures and their SoC mappings.
- MaGNAS first contributes a self-contained framework for designing ViG supernets to characterize their search space of GNN-based architectural design choices.
- To identify optimal *ViG architecture-mapping* pairs, MaGNAS solves a bilevel optimization problem via a two-tier evolutionary search algorithm of two optimization engines: an *outer* engine to optimize GNN architectural design choices; an *inner* engine to identify optimal mapping strategies for ViG workloads onto heterogeneous CUs.
- We conduct extensive experiments, in-depth analysis, and ablation studies using a real MPSoC platform (Nvidia AGX Xavier [1]) and hardware simulator (MAESTRO [98]) on four (04) state-of-the-art vision datasets. Our findings have demonstrated the superiority of MaGNAS in designing and mapping ViG models on these platforms.

3.2 Preliminaries and Related Works

3.2.1 A Primer on Vision Graph Neural Networks

We describe the main constituents of the ViG architecture [68], which pioneered a generic approach for graph-based image processing by modeling raw input images as graphs.

Graphing Image Data Structures. The ViG operates on images modeled as graphs of patches. A $W \times H \times C$ image is first partitioned into N patches of dimensions $W' \times H' \times C'$. Each patch's dimensions can be viewed as a single feature vector $x_i \in \mathbb{R}^D$ where $D = W' \times H' \times C'$. To construct the graph, a node v_i is assigned to each patch, forming an unordered set of N nodes $\mathcal{V} = \{v_1, v_2, \ldots, v_N\}$ associated with the corresponding set of feature vectors $X = \{x_1, x_2, \ldots, x_N\}$, where x_i can be called the *feature embedding* of vertex v_i. To build graph edges, K edges are constructed for each v_i based on the K nearest vertices in its neighborhood $\mathcal{N}(\mathcal{V})$, that is, for every $v_j \in \mathcal{N}(\mathcal{V})$, an edge e_{ji} is constructed from v_j to v_i. Finally, the full graph structure of the image is given by $\mathcal{G}(\mathcal{V}, \mathcal{E})$, which can be inputted into the ViG model for processing.

Graph Processing Layer. Describing a graph through its features, $\mathcal{G} = G(X)$ s.t. $X \in \mathbb{R}^{N \times D}$, a typical GCN layer on $\mathcal{G}$ can be represented by the following abstract formula:

$$\mathcal{G}' = Combine(Aggregate(\mathcal{G}, W_{agg}), W_{comb}) \tag{3.1}$$

where $\mathcal{G}$ is processed through an *aggregation* and a *combination* stages of the GCN layer. W_{agg} and W_{comb} resemble the respective learnable weights of each stage. The *aggregation* stage employs a feature exchange procedure in which every node v_i receives features $x_j \in \mathcal{N}(x_i) s.t. i \neq j$ from its neighboring nodes and aggregates them to provide x_i'. The *combination* stage involves further treatment of features x_i' (as through an MLP layer) to obtain refined representation x_i''. We remark that for each of the two stages, a variety of operations can be employed (e.g., aggregation through sum, max-relative, mean), which correspond to the variety of GCN layer types existing in the literature (e.g., `GraphSage`, `GIN`, etc.). Lastly, The resulting output feature set from both stages, X', is used to construct the output graph $\mathcal{G}' = G(X')$.

Grapher and FFN Modules. To enrich feature representation, graph processing layers can be interleaved with typical DNN layers in a GNN model. As such, the standard ViG architecture comprises a stack of two basic building blocks: *Grapher* and *Feed Forward Network (FFN)* given by:

$$L^{Grapher} = l^{post} \circ l^{comb} \circ l^{agg} \circ l^{pre}, \qquad L^{FFN} = l^{fc_2} \circ l^{fc_1} \tag{3.2}$$

The *Grapher* comprises at its core the GCN layer with its aggregation, l^{agg}, and combination, l^{comb}, operations, injected between two linear layers, namely *pre-processing*, (l^{pre}), and *post-processing*, l^{post}, layers, to promote feature diversity. The *FFN* block constitutes two fully connected layers that further elevate feature capacity, l^{fc_1} and l^{fc_2}. For every GCN or fully-connected layer in either module, non-linear activation and batch normalization operations are applied. From here, every *Grapher* can be followed by an optional *FFN* to form the ViG block, and the sequence of ViG blocks form the ViG backbone architecture.

3.2.2 Related Works

GNNs for vision. Through learning graph-level features, GNNs achieved remarkable performance on a variety of computer vision tasks, such as activity recognition and point clouds classification [102, 198, 215]. Scene graph generation [121, 209, 223] has emerged as a viable approach to generate a graph of objects and their relations from an image through cascading an object detector and a GCN model. The ViG [68] studied here relies on a standard GCN

Table 3.1: Comparison between related Graph Neural Architecture Search works and ours.

	[60]	[59]	[231]	[170]	[228]	[230]	MaGNAS
Training-in-the-loop NAS	✓	✓	✓	✓			
Once-for-all NAS					✓	✓	✓
Vision GNN							✓
Hardware Awareness					✓	✓	✓
GNN-Hardware co-design					✓		
Edge Computing Setting						✓	✓
Distributed Mapping							✓

backbone to generate and process graphs from raw images directly.

Hardware acceleration for GNNs Numerous works [13, 30, 95, 177, 214, 218] have proposed hybrid accelerator architectures comprising separate engines and specialized hardware components to effectively manage the non-uniform GNN dataflow on both an *intra-* and *inter-phase* level. More recent work [61] attempted to characterize the design space of dataflow choices to enable running GNNs on customary reconfigurable spatial accelerators to serve various GNN use cases.

Distributed Computing of GNNs. Distributing DNN workloads across the heterogeneous computing resources of CPU, GPU, DLAs, and FPGAs, is an active field of research [23, 40, 93, 151, 213], with different forms of GNN workload distribution (task-level, data-level, and pipelining) have been explored in the literature [31].

Graph Neural Architecture Search. Our work falls under the category of HW-aware NAS for GNNs as these two, with several distinctive features summarized in Table 3.1 as will be seen in the following discussions.

3.3 System Model and Problem Formulation

In this section, we model the mapping problem of GNN kernels onto heterogeneous SoC CUs. Then, we derive a formulation for the global design-mapping bi-optimization objective.

3.3.1 System Model for mapping GNNs onto Heterogeneous SoCs

GNN Workload Characterization. Let a standard GNN model architecture, α, be formally described as a sequence of n computing blocks as follows:

$$\alpha = L_n \circ L_{n-1} \circ \cdots \circ L_1,\ s.t.\ L_i \neq L_{i-1},\ L_i \in \{L^{FFN}, L^{Grapher}\},\ L^{FFN} \in \{L^{FFN}, \phi\}\ \forall 1 \leq i \leq n \tag{3.3}$$

where each GNN computing block L_i can either be the *Grapher* or *FFN* blocks as defined in the previous section, denoted by $L^{Grapher}$ and L^{FFN}, respectively. The condition ensures that each $L^{Grapher}$ block can be succeeded by an optional L^{FFN} block.

Let X_j be the input graph-level features for block $L_j \in \alpha$. Then, the output feature embedding vector, X_{j+1}, can be obtained as:

$$X_{j+1} = L_j(X_j)\ \ s.t.\ \ x_k^j \in \mathbb{R}^{D'}\ \forall\ x_k^j \in X_j \tag{3.4}$$

where the condition ensures that feature embedding dimensions remain consistent throughout each computing block within the GNN. That is the feature embedding for x_k^j (the k^{th} node within the graph representation at the j^{th} block) retains the same D' dimensions before and after being processed through block L_j. Note that D' can either be equivalent to D or a downsampled version of it as some architectures (e.g., Pyramid [68]) can include extra downsampling layers in-between computing blocks to promote abstract feature learning.

Let $\mathbb{CU} = \{\mathcal{CU}_1, \mathcal{CU}_2, \cdots, \mathcal{CU}_M\}$ be the set of available computing units within a heterogeneous MPSoC. Considering a *blockwise* granularity, we can define a mapping vector, m, to characterize the workload distribution for each GNN computational block as follows:

$$m = [\pi_1, \pi_2, \cdots, \pi_n],\ \ s.t.\ \ \pi_i \in \mathbb{CU}\ \forall\ 1 \leq i \leq n \mid support(\pi_i, L_i) == True \tag{3.5}$$

where each entry π_i in $\mathbb{M}$ describes the mapping assignment of L_i onto a computing unit $\mathcal{CU}_m \in \mathbb{CU}$ as long as this corresponding $\mathcal{CU}_m$ hardware supports running L_i.

Performance Modelling. For a mapping strategy m, the total latency and energy consumption overheads, T_{total} and E_{total}, experienced by a GNN model under distributed pipelined deployment can be modeled as the sum of the overheads incurred by its individual blocks:

$$T_{tot}(m) = \sum_{i=1}^{n} T_i(m), \quad s.t.\ \ T_i(m) = \tau_i^{comp} + \mathbb{I}[\pi_{i-1} \neq \pi_i] \cdot \tau_i^{in} + \mathbb{I}[\pi_i \neq \pi_{i+1}] \cdot \tau_i^{out} \quad (3.6)$$

$$E_{tot}(m) = \sum_{i=1}^{n} E_i(m), \quad s.t.\ \ E_i(m) = e_i^{comp} + \mathbb{I}[\pi_{i-1} \neq \pi_i] \cdot e_i^{in} + \mathbb{I}[\pi_i \neq \pi_{i+1}] \cdot e_i^{out} \quad (3.7)$$

where the τ_i^{comp} and e_i^{comp} are the respective computational latency and energy consumption experienced by L_i given its corresponding mapping, π_i. τ_i^{in} and τ_i^{out} are the latency overhead sustained when loading and writing back graph features from and to the *shared system memory* on the SoC, respectively. The indicator function $\mathbb{I}[\cdot]$ evaluates to 1 only when the associated condition is met; that is, no transmission overhead penalties are sustained between two consecutive layers when they are both assigned the same computing unit. For the energy formula, the same logic of notation applies for every layer L_i.

Mapping Problem Formulation. Define $P(m) = f(T_{tot}(m), E_{tot}(m))$ to be a combined evaluation function for a mapping configuration m. Let $\mathbb{M}$ be the set of feasible mapping configurations. Then, we can formulate the mapping objective function for an architecture α deployed on a heterogeneous SoC platform as follows:

$$m^* = \max_{m \in \mathbb{M}} P(m), \quad s.t.\ \ T_{tot} < T_{TRG},\ E_{tot} < E_{TRG} \quad (3.8)$$

where the goal is to identify an optimal mapping strategy, m^*, for α such that performance objective function P is maximized with respect to latency and energy under user-specified constraints on latency and energy consumption, T^{TRG} and E^{TRG}, respectively.

3.3.2 Nested Search Formulation

Given our observations in the motivational study, we further refine our formulation to an *architecture-mapping* bi-level co-optimization problem, where the goal is to identify the optimal set of design choices for the GNN architecture and its mapping strategy. At the top level, a *GNN architecture subspace ($\mathbb{A}$)* describes the set of architectural design choices associated with the GNN model. At the inner level, a *mapping subspace ($\mathbb{M}$)* specifying the possible distributed mapping options given the underlying CUs. Through this designation, mapping choices become conditioned on architectural choices, which promotes the generality of this approach. Formally, the nested optimization formulation can be given as follows:

$$\alpha^* = \max_{\alpha \in \mathbb{A}} \psi[Acc(\alpha), P(m^*|\alpha, \mathbb{CU})] \tag{3.9}$$

$$s.t.\ m^* = \max_{m \in \mathbb{M}} P(m|\alpha, \mathbb{CU}) \tag{3.10}$$

where the outer optimization equation targets identifying the optimal set of GNN architectural parameters, α^*, that yield the best scores on a combined function, ψ, of both the accuracy, $Acc(\cdot)$, and performance efficiency $P(\cdot)$. Evaluation of $P(\cdot)$ is based on the results from the inner optimization equation. That is, energy and latency performance evaluations used for scoring a candidate architecture, α, are for the architecture's optimal mapping, m^*.

3.4 MaGNAS Framework

We present **MaGNAS**, a mapping-aware Graph Neural Architecture Search framework for heterogeneous SoC deployment. **MaGNAS** solves the co-optimization problem through: (*i*) the construction and training of a ViG supernet as a viable GNN design space, (*ii*) developing a bi-level evolutionary algorithm for optimizing *architecture-mapping* pairings.

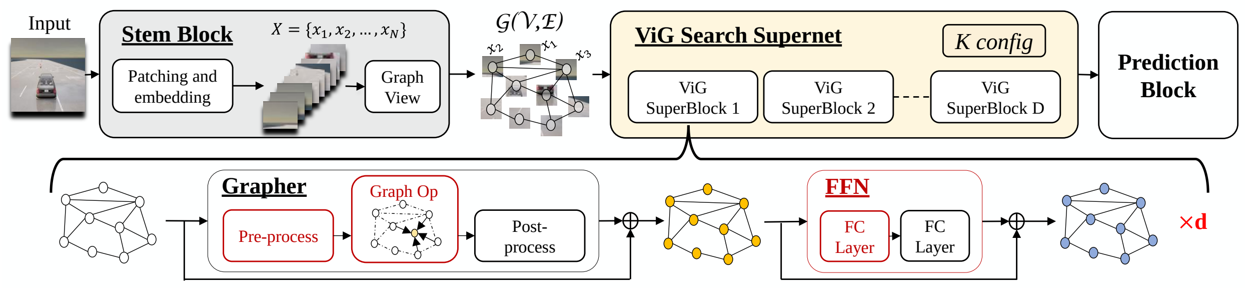


Figure 3.2: The ViG supernet implementation for MaGNAS co-search framework. The supernet comprises D ViG search super blocks, each of which constitutes a sequence of d_i Grapher and FFN computing modules. Architectural search parameters characterizing $\mathbb{A}$ subspace are highlighted in *red* and detailed in the text.

3.4.1 Supernet Construction and Training

We extend the ViG architecture introduced in Section 3.2.1 to construct a supernet of various design choices to characterize an architectural search space $\mathbb{A}$. Briefly, a *supernet* represents a network of networks that can be trained simultaneously to facilitate providing diverse model designs for different deployment scenarios [25]. In the context of ViGs, each subnet within a supernet is defined by a unique set of architectural parameter choices (e.g., choice of GNN layers, #layers, etc.). Additionally, supernets entertain the property of *weight-sharing*, meaning that during the supernet's training, weight updates for a candidate layer are applied and reused across all subnets that share that particular layer, which enables the simultaneous training of all subnets within it. Once the supernet is trained, a search algorithm can be employed to identify an ideal subnet that meets the target specifications. The ViG supernet is illustrated in Figure 3.2, where the choice of architectural search parameters for $\mathbb{A}$ is based on observations from both related works [59, 68, 218, 220] as well as from our initial experiments. The supernet construction is detailed in the following.

ViG Superblocks. The backbone ViG-S architecture in [68] comprises 16 computing blocks, each comprising a stack of a *Grapher* and an *FFN* module. On the one hand, characterizing $\mathbb{A}$ on a per-layer or a per-block basis can lead to an explosion in the search space, given the number and cardinality of various search parameters. Conversely, associating the parameters of $\mathbb{A}$ with the entire backbone restricts fine-grained architectural optimizations, not fully exploiting the power of diversified architectural settings at different model stages. As a compromise, we propose *ViG superblocks* to characterize $\mathbb{A}$, where each i^{th} superblock constitutes a collection of d_i ViG blocks sharing the same design choices. The merits of the ViG superblocks are twofold: (i) they balance the trade-off between architectural diversity and search space complexity; (ii) They facilitate effective management of the depth parameter through d_i while preserving key architectural features.

$\mathbb{A}$ **search parameters.** For each superblock i, we specify the following parameters to construct our architectural search space $\mathbb{A}$:

- *The depth, d_i*, to indicate how many ViG blocks exist in the i^{th} superblock i.
- *Grapher pre-processing* as a binary decision variable to indicate whether a pre-processing layer exists before every graph processing layer.
- *Graph Op* to specify the graph operation employed throughout the i^{th} superblock.
- *FFN module* as a binary decision variable to indicate whether FFN modules should exist in this superblock.
- *FC hidden layer dimension* to specify the size of the intermediate features in the FFN.

Supernet Training. We train the supernet for our target task using a combination of Cross-Entropy and knowledge distillation loss functions, where for the latter, we employ a pretrained model as a teacher for more representative training on soft labels' training [22,222]. This training is performed from scratch due to: (i) The ViG is a relatively new GNN architecture and the availability of pretrained weights is still limited.(ii) loading the exact pretrained model weights from the original ViG backbone [68] can introduce a bias towards certain design choices during training.

To train the supernet, we sample and train a set of subnets at each iteration. The choice of subnets is realized through 3 separate samplers following the Sandwich sampling rule [222]:

- *Maximum Sampler:* sample the largest subnet from $\mathbb{A}$, that is, the one with the maximum depth and width (i.e., hidden dimension features).
- *Minimum Sampler:* sample the smallest subnet from $\mathbb{A}$.
- *Balanced Sampler:* sample a number of random subnets of different architectures.

This scheme enables improving the performance of all subnets within the search space simultaneously by pushing the upper and lower performance bounds with every iteration. Furthermore, given how numerous GNN architectures leverage a homogeneous structure, that is, one where the choice of the *Graph OP* is kept consistent throughout the entire architecture, we modify the Maximum/Minimum samplers so that they sample architectures of maximal/minimal sizes, but constituting a randomly selected *Graph Op* repeated throughout the model. This ensures training fairness by pushing the upper and lower boundaries of architectures of different graph operations and avoids bias towards specific implementations.

3.4.2 Nested Evolutionary Search: Outer Optimization Engine

In order to solve the bi-level *architecture-mapping* optimization problem formulated in equations (3.9) and (3.10), we construct the two-tier evolutionary search framework illustrated in Figure 3.3 to identify optimal architecture-mapping pairings. Specifically, an evolutionary search works by creating a population of candidate solutions from a search space, evaluating each one, and propagating the top-performing solutions to the gene pool of subsequent generations. These solutions can then endure and undergo the genetic operations of mutation and crossover to contribute new derivative solutions for the following generations. Thus with

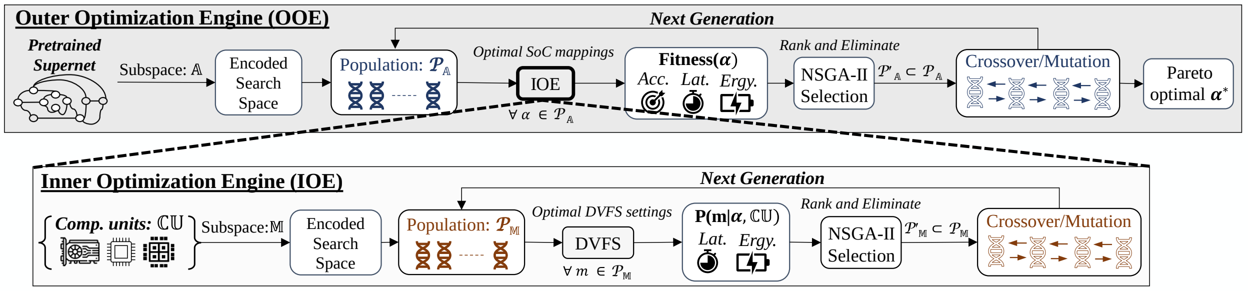

Outer Optimization Engine (OOE)
Next Generation
Pretrained Supernet
Subspace: $\mathbb{A}$
Encoded Search Space
Population: $\mathcal{P}_{\mathbb{A}}$
Optimal SoC mappings
IOE
$\forall \alpha \in \mathcal{P}_{\mathbb{A}}$
Fitness(α)
Acc. Lat. Ergy.
Rank and Eliminate
NSGA-II Selection
$\mathcal{P}'_{\mathbb{A}} \subset \mathcal{P}_{\mathbb{A}}$
Crossover/Mutation
Pareto optimal α^*
Inner Optimization Engine (IOE)
Next Generation
Comp. units: $\mathbb{CU}$
Subspace: $\mathbb{M}$
Encoded Search Space
Population: $\mathcal{P}_{\mathbb{M}}$
Optimal DVFS settings
DVFS
$\forall m \in \mathcal{P}_{\mathbb{M}}$
P(m|α, $\mathbb{CU}$)
Lat. Ergy.
Rank and Eliminate
NSGA-II Selection
$\mathcal{P}'_{\mathbb{M}} \subset \mathcal{P}_{\mathbb{M}}$
Crossover/Mutation

Figure 3.3: MaGNAS two-tier evolutionary search framework

each evolution, only new non-dominated solutions from the current population are added, and the newly-dominated ones in the archive are removed.

We first describe the Outer Optimization Engine (OOE), which employs a higher-level evolutionary algorithm whose purpose is to: (i) search through the supernet to identify the most-promising GNN subnets and (ii) rank candidate subnets

Subspace $\mathbb{A}$ Description. By adopting a Once-For-All (OFA) NAS approach [25], the *training* and *search* stages within MaGNAS are decoupled, significantly reducing the search process overheads as once the supernet has been trained, its search subspace, $\mathbb{A}$, can be reused for the search to identify beneficial subnets. Accordingly, subspace $\mathbb{A}$ in the search stage is encoded as a sequence of 04 discrete vectors, each representing the architectural parameters for each ViG superblock listed in 3.4.1, facilitating the sampling of subnets as GNN architectural design candidates, $\alpha \in \mathbb{A}$.

OOE Evolutionary Search. The next step is to employ a search algorithm to solve the optimization objective in (3.9) by searching for optimal GNN architectural implementations, α^*. Here, we implemented the NSGA-II evolutionary search algorithm to navigate through $\mathbb{A}$ and explore the subspace of viable design choices. Typically, the search algorithm is run for a pre-specified number of *generations*, where a new population of candidate architectural designs, $\mathcal{P}^g_{\mathbb{A}}$, is sampled with every generation, g. Then, $\forall \alpha \in \mathcal{P}^g_{\mathbb{A}}$, a *fitness* evaluation function, $F(\cdot)$, is applied as follows:

$$F(\alpha) = f(Acc_\alpha, T_\alpha, E_\alpha) \tag{3.11}$$

which scores every α based on its target task accuracy, latency, and energy consumption on the target platform denoted by Acc_α, T_α, and E_α, respectively. Acc_α evaluation can be obtained directly by evaluating the α model predictive performance on the test dataset, whereas estimates of T_α, and E_α are provided by the inner optimization engine based on evaluations of the ideal mapping strategy, m^* (which will be detailed in the following subsection). Though we used for $F(\cdot)$ a weighted product function of the objective evaluations in our implementation, we kept its definition here abstract for generality. According to the fitness evaluation scores, every $\alpha \in \mathcal{P}^g_{\mathbb{A}}$ is ranked via the NSGA-II non-dominated sorting algorithm. Based on the rankings, an elimination process is initiated afterward to yield a population subset $\mathcal{P}'^g_{\mathbb{A}} \subset \mathcal{P}^g_{\mathbb{A}}$. Subset $\mathcal{P}'^g_{\mathbb{A}}$ then undergoes *mutation* and *crossover* operations to provide a new population $\mathcal{P}^{g+1}_{\mathbb{A}}$ for the following generation $g + 1$. This iterative search continues until the search budget expires. At the last iteration, a Pareto-optimal set, $\{\alpha^*|m^*\}$, is provided. In our experiments, we sample 100 architectures for $\mathcal{P}^g_{\mathbb{A}}$ out of a total $|\mathbb{A}| \approx 2^{29}$ candidates. After fitness evaluations, we select a subset of 30% from the top-ranked candidates as $\mathcal{P}'^g_{\mathbb{A}}$.

3.4.3 Nested Evolutionary Search: Inner Optimization Engine

To estimate T_α and E_α $\forall \alpha \in \mathcal{P}^g_{\mathbb{A}}$, we develop an Inner Optimization Engine (IOE) to specify an ideal mapping strategy of α onto underlying SoC ($\alpha \rightarrow \mathbb{CU}$) and evaluate its performance.

Subspace $\mathbb{M}$ Description. The mapping configuration, m, defined in equation (3.5) reflects the encoded discrete vector within the IOE search space that characterizes potential mapping options for each *Grapher* and *FFN* modules from α. We also extend the specification of m

in the IOE to incorporate two further mapping options for the *stem* and *prediction* modules.

IOE Evolutionary Search. Given how the mapping decision space is at least $|\mathbb{CU}|^n$ (see equation (3.3)), a brute-force search to determine the ideal mapping, m^*, can be costly. As such, we implement another NSGA-II evolutionary algorithm in the inner optimization level to effectively explore mapping choices within $\mathbb{M}$ and identify the best candidates. Particularly, a population of mapping configurations, denoted by $\mathcal{P}_{\mathbb{M}}^g$, is sampled every generation g by the search algorithm. Then for every $m \in \mathbb{M}$, a fitness evaluation function $P(\cdot)$ is applied as given in the below formula:

$$P(m|\alpha, \mathbb{CU}) = (\frac{E_\alpha^m}{max\{E_\alpha^{\mathcal{CU}}\}})^{\gamma_1} \times (\frac{L_\alpha^m}{max\{L_\alpha^{\mathcal{CU}}\}})^{\gamma_2} \quad \forall \mathcal{CU} \in \mathbb{CU} \tag{3.12}$$

where E_α^m and L_α^m are the respective energy and latency sustained by α when its components are deployed onto the underlying hardware following a mapping strategy m. Each of these values is then normalized by the best *standalone* deployment option from $\mathbb{CU}$, denoted here by E_α^{CU} and L_α^{CU}, respectively. The reasons for this normalization are twofold: (i) To ensure fairness when comparing various mapping options for α; (ii) To enforce achieving comparable, if not improved, performance scores over those obtained by the canonical standalone deployment options. γ_1 and γ_2 are user-specified tunable hyperparameter values to enable prioritizing one performance objective or the other. For our experiments, we constructed low-cost, accessible hardware lookup tables for the target CUs.

Accordingly, another non-dominated sorting algorithm is instantiated to rank mapping configurations, retaining the top-ranked configurations to provide population subset $\mathcal{P'}_{\mathbb{M}}^g \subset \mathcal{P}_{\mathbb{M}}^g$. Afterwards, subset $\mathcal{P'}_{\mathbb{M}}^g$ undergoes mutation and crossover to provide $\mathcal{P}_{\mathbb{M}}^{g+1}$ as the new population for the next generation. Once the search budget expires, $E_\alpha^{m^*}$ and $L_\alpha^{m^*}$ are returned as evaluations for the best configuration, m^*, to be used for E_α and T_α in the OOE, respectively.

Performance Characterization. Generally, estimates of E_α^m and L_α^m for every $m \in \mathcal{P}_{\mathbb{M}}^g$

can be provided through a multitude of approaches (e.g., predictive models). As was shown in equation (3.4), the dimensional consistency of graph features offered throughout the ViG backbone has led to a tractable space of evaluation possibilities, enabling the construction of low-cost lookup tables to directly retrieve performance estimates of various architecture-mapping configurations. Simply put, the lookup tables are indexed by the architectural parameters of a computing block, L_i, and the CU to whom it is mapped. By invoking the tables for every block in α given m, the performance overheads of each block can be aggregated to estimate the total E_{α}^{m} and L_{α}^{m}. Although lookup tables work for our case, proxy prediction models can be more feasible for a different GNN architecture in which the graph features dimensions change as a result of inconsistent graph structures.

DVFS Search Support. We also include the option to supplement $\mathbb{M}$ subspace with the configuration setting choices of dynamic voltage and frequency scaling (DVFS) features. We specify a DVFS search block in the IOE as a *third* optional optimization level contingent upon the choices of m and α. This is convenient as the search space of the DVFS is small compared to $\mathbb{A}$ and $\mathbb{M}$. Formally, if we denote a single set of DVFS configuration settings as ϑ and the overall DVFS search space as Ψ, then the DVFS search objective is given as:

$$\vartheta^* = \max_{\vartheta \in \Psi} P(m|\alpha, \mathbb{CU}, \vartheta) \tag{3.13}$$

where the performance evaluation of m becomes also contingent upon the choice of $\vartheta \in \Psi$.

3.5 Experiments

We conduct extensive experiments using a real MPSoC platform and hardware simulation on four(04) state-of-the-art image classification datasets to assess the merit of MaGNAS in designing ViG architectures and mapping them onto heterogeneous CUs.

Table 3.2: Search space parameters for GNN architectures.

Decision variables	**Values**	**Cardinality**
	Supernet Search Space ($\mathbb{A}$)	
Superblock depth (d)	{2, 3, 4}	3
Graph Op	{`Max-Relative`, `EdgeConv`, `GraphSAGE`, `GIN`}	4
Skip pre-process (fc_use)	{False, True}	2
Skip post-process (ffn_use)	{False, True}	2
FFN hidden features (w)	{96, 192, 320}	3
	Mapping Search Space ($\mathbb{M}$) for NVIDIA Xavier AGX	
Computing units	{GPU, DLA}	2
Mapping granularity	{Stem, Grapher, FFN, Cls}	$\mathcal{O}(1.7\times10^{12})$
	DVFS Settings Search space (Ψ) for NVIDIA Xavier AGX	
CPU clock frequency	{1728MHz, 2265MHz}	2
GPU clock frequency	{520MHz, 900MHz, 1377MHz}	3
EMC clock frequency	{1065MHz, 2133MHz}	2
DLA clock frequency	{1050MHz, 1395MHz}	2

3.5.1 Experimental Setup

Supernet Design. We build our supernet on top of the ViG-S variant [68] with 16 computing blocks, each a *Grapher* and an *FFN* block. We group every four (04) computing blocks into a *ViG superblock*, and assign to each K nearest neighbor values of 12, 16, 20, and 24, respectively, which enables aggregation of features from farther nodes with each superblock. To support dynamic width and depth configurations, we transform each ViG superblock into a *slimmable* neural network following [221]. To support varying graph operations, we specify a dynamic graph processing layer in the *Grapher* with four concurrent branches reflecting different GCN operational choices for *Graph Op*: 1) `EdgeConv` [198], 2) `GIN` [210], 3) `GraphSAGE` [67], and 4) `Max-Relative` GraphConv [106].

Datasets and Training. We employ four (04) image classification datasets of CIFAR-10, CIFAR-100, Tiny-Imagenet, and Oxford-Flowers. To transform the images to graphs, images are first scaled to 224×224×3 resolution, and transformed through the *Stem* block into a graph of nodes $N = 196$, each of dimension $D = 14 \times 14 \times 320$. The supernet training for each dataset is run for 150, 150, 250, and 250 for each respective dataset in the order in which they were stated. The training is performed using an Adam optimizer with a momentum of 0.9, weight decay of 0.05, and dropout set to 0.2. We use cosine as a learning rate scheduler with an initial LR of 0.003 and batch size of 320 on a 20 Nvidia RTX 2080 Ti GPU cluster.

Evolutionary Search Settings. Table 3.2 lists the search sub-spaces of $\mathbb{A}$, $\mathbb{M}$, and Ψ designated within our optimization framework. For the optimization process, we fix the population size to 100 and 200 and the number of generations to 50, and 10 for the OOE and IOE, respectively. Combining the OOE and IOE, we explored $\sim 1.6 \times 10^6$ candidates of GNN architectures and deployment settings on an Nvidia Xavier AGX platform.

Hardware experimental settings. We evaluate our approach using two hardware setups presenting a variety of computing units and architectural features: (i) NVIDIA Jetson AGX Xavier [1], as an MPSoC platform; (ii) MAESTRO [98, 99], a hardware simulator tool. The Jetson AGX Xavier comprises a Volta GPU and an energy-efficient DLA, which use for our $\mathbb{CU}$. Whereas using MAESTRO, we simulate a use-case of an SoC with three (03) heterogeneous CUs. We use the native dataflows in MAESTRO of *kcp_ws*, *ykp_os*, and *dpt* for our 3 CUs, which for simplicity, we denote by **DSA-k**, **DSA-y**, and **DSA-d**. W

GNN architectures baselines: These include the original isotropic ViG-S model in [68] as well as its variants by altering *Graph Op* (i.e., the GCN operation) where the *Graph Op* remains consistent across all the layers. Specifically, we identify the baselines by their recurring *Graph Op* operation: 1) **b0**: ViG-S/Max-Relative, 2) **b1**: ViG-S/EdgeConv, 3) **b2**: ViG-S/GIN, and 4) **b3**: ViG-S/GraphSage. For the scalability analysis of the IOE, we also consider the PyramidViG-M as the alternative ViG backbone that sustains graph features dimensional reductions as the network deepens.

HW-mapping baselines: We consider the default *standalone* deployment options – i.e., the full mapping of an entire ViG model to a singular CU (e.g., to the GPU only). We also consider hybrid mapping strategies in which inter-CU transitions are limited, as pro-

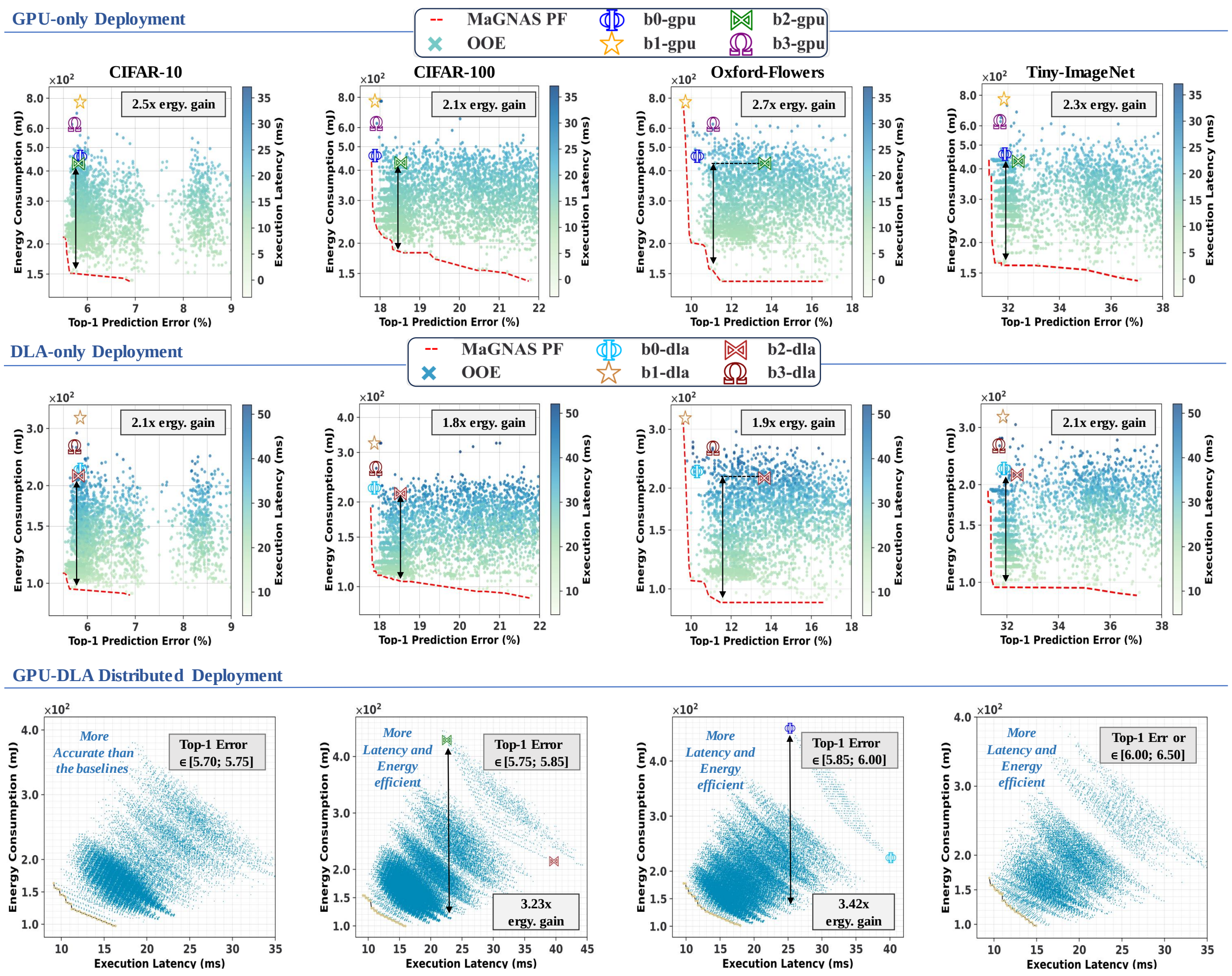


Figure 3.4: The first two rows show the performance of the explored GNNs in ($\mathbb{A}$) by the OOE on four datasets (*from left to right*: a) CIFAR-10, b) CIFAR-100, c) Oxford-Flowers, and d) Tiny-ImageNet. The Hardware metrics (i.e., latency and energy) are shown for *GPU-only* deployment in the first row and for *DLA-only* deployment in the second row. The third row shows the IOE results on CIFAR-10 grouped by prediction error intervals.

posed in [40]. We also use the aforementioned PyramidViG-M GIN-variant for our hardware scalability experiments using MAESTRO.

3.5.2 OOE Results: GNN Architecture Optimization

In Figure 3.4, the first two rows depict the explored GNN architectures from $\mathbb{A}$ by the OOE on the four (04) datasets given *standalone* mapping strategies on GPU-only (*top row*) and

DLA-only (*middle row*). Compared to the baselines defined above, our obtained Pareto-optimal GNN architectures generally dominate all baselines on the four image classification datasets with regard to the three performance metrics of accuracy, latency, and energy consumption. Specifically, the OOE can identify GNN architectures that achieve up to $\sim$**3.6**$\times$ latency speedup than baselines when deployed onto the GPU; can realize up to $\sim$**2.8**$\times$ more energy efficiency gains compared to the baselines when deployed onto the DLA – all while maintaining comparable accuracy scores. As will be emphasized in the subsequent Section 3.5.4, the reasons for this dominance by the OOE's GNN architectures is attributed to the allowed diversification of *Graph Op* across the different ViG superblocks, which has facilitated achieving better accuracy-performance trade-offs. Moreover, skipping the *FFN* and the *Grapher's* FC pre-processing layers offers attractive design choices to avoid unnecessary computation, especially when the set of features is limited and can be already captured by the basic layers of the *Grapher* modules – which is the case for the simpler datasets (e.g., CIFAR-10). Our OOE recognized this property and leveraged its knowledge to concentrate its search on identifying GNN architectural parameters that achieve the best accuracy levels with the minimal number of *FFN* and FC pre-processing layers.

3.5.3 IOE Results: Hardware Mapping Optimization

The bottom row of Figure 3.4 shows the optimization results when exploring mapping strategies from $\mathbb{M}$ for the *top-performing* GNN architectures (as ranked by equation 3.11) provided to the IOE. The results are reported for CIFAR-10 and grouped by TOP-1 error intervals in each sub-figure. A similar trend has also been observed in the other datasets. At each top-1 error interval, we can observe that the IOE explored various mapping strategies, as illustrated by the latency-energy trade-offs. The bulk of these trade-offs are captured within the range of performance values between the GPU-only and DLA-only mapping options values.

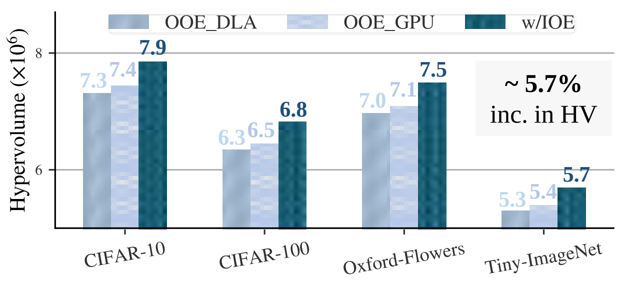
Table 3.3: Detailed performance results, GNN architectural parameters, and mapping strategies of our Pareto optimal models (**a0-a3**). The original ViG-S and its variants (**b0-b3**) on the four datasets on the NVIDIA Jetson Xavier AGX SoC platform. '**G**' and '**D**' in the latency and energy columns indicate GPU and DLA, respectively.

Datasets	**GNN Models**	**TOP-1 Acc (%)**	**Graph-Ops (M, E, G, S)**	**FFN-use (%)**	**FC pre-use (%)**	**Latency (ms)**	**Energy (mJ)**	**GPU-use (%)**	**DLA-use (%)**
All-datasets	Φ Baseline-b0	**C10:** 94.15, **C100:** 82.13 **F:** 89.71, **Ti:** 68.12	M-M-M-M	100	100	**G:** 25.28 **D:** 40.11	**G:** 459.44 **D:** 224.41	-	-
	★ Baseline-b1	**C10:** 94.15 **C100:** 82.13 **F:** 90.29, **Ti:** 68.15	E-E-E-E	100	100	**G:** 33.74 **D:** 62.11	**G:** 770.36 **D:** 323.70	-	-
	⋈ Baseline-b2	**C10:** 94.20, **C100:** 81.49 **F:** 86.37, **Ti:** 67.62	G-G-G-G	100	100	**G:** 22.49 **D:** 39.62	**G:** 429.07 **D:** 214.35	-	-
	Ω Baseline-b3	**C10:** 94.27, **C100:** 82.10 **F:** 88.92, **Ti:** 68.32	S-S-S-S	100	100	**G:** 29.57 **D:** 57.77	**G:** 623.76 **D:** 263.48	-	-
CIFAR-10 **(C10)**	◯ Ours-a0	94.25	G-G-G-G	25	25	16.02	97.0	09	91
	◯ Ours-a1	**94.46**	G-G-G-G	100	0	19.49	118.00	17	83
	◯ Ours-a2	94.32	G-M-G-G	25	0	**11.19**	121.14	75	25
	◯ Ours-a3	94.32	G-M-G-G	25	0	14.18	**105.11**	33	67
CIFAR-100 **(C100)**	◯ Ours-a0	82.13	S-G-S-G	100	25	17.72	180.56	50	50
	◯ Ours-a1	**82.17**	S-S-S-S	100	75	34.72	271.62	30	70
	◯ Ours-a2	81.63	G-G-G-G	50	50	**15.06**	**131.81**	50	50
	◯ Ours-a3	82.13	S-G-S-G	100	25	17.29	197.80	55	45
Oxford-Flowers **(F)**	◯ Ours-a0	**89.90**	M-G-M-M	75	75	14.37	153.54	69	31
	◯ Ours-a1	88.43	G-G-G-G	0	50	**9.60**	119.07	90	10
	◯ Ours-a2	88.43	G-G-G-G	0	50	12.30	**105.88**	40	60
	◯ Ours-a3	89.02	M-G-G-G	25	25	12.82	116.63	50	50
Tiny-ImageNet **(Ti)**	◯ Ours-a0	68.40	M-G-G-G	25	0	**13.07**	114.89	50	50
	◯ Ours-a1	68.40	M-G-G-G	25	0	15.47	**102.06**	17	83
	◯ Ours-a2	**68.51**	M-G-G-G	75	25	16.37	122.56	38	62
	◯ Ours-a3	**68.51**	M-G-G-G	75	25	17.87	115.78	19	81

Furthermore, as both *GNNs* and *mappings* are considered together in the IOE design space, superior energy gains can be realized through more compact GNN architectures. For instance, as illustrated in the third sub-Figure, an energy gain up to ∼**3.42**× can be attained compared to the **b2-gpu** while preserving comparable latency and accuracy levels by opting for another GNN architecture and distributed mapping. Upon comparing the curve lines, we can observe that GNN architectures that outperformed the baselines in the OOE (i.e., in the standalone deployment options shown by the extremes) typically maintain their dominance within the IOE and proves that rank is preserved across GNN architectures and mapping schemes in this joint search space.

3.5.4 Analysis of Pareto Search and Models

Results Discussion. In Table 3.3, we provide a detailed analysis of performances, architectural parameters, and mapping strategies of the ViG baselines [**b0-b3**] and a selection



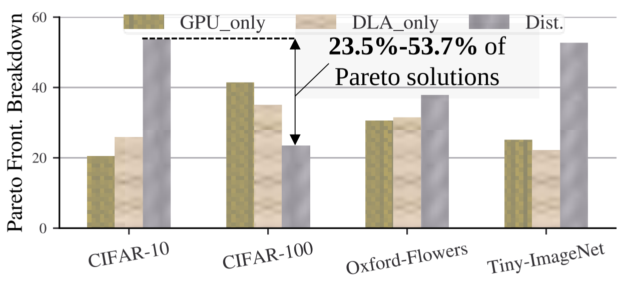


Figure 3.5: *Left*: Hypervolume analysis when including the IOE against those of the standalone OOE for the DLA and GPU. *Right*: Breakdown of the combined Pareto Fronts constituents on the basis of mapping options.

of our final Pareto optimal models from the two-tier search [**a0-a3**] for each dataset. As shown, although our models maintain comparable accuracy scores to the baselines, they generally achieve better speedups and energy efficiency results. To be more precise, our models achieve on average ~**1.57**× and ~**2.49**× latency speedups; ~**3.38**× and ~**1.65**× more energy efficiency when compared against the original ViG baseline fully-deployed onto the GPU and DLA, respectively. This dominance is primarily attributed to 3 factors: (*i*) the enabled diversification of Graph Op parameter throughout the ViG superblocks, which enables interleaving both *powerful* and *resource-efficient* operators within a model architecture. For instance, examining the Oxford-Flowers results in the Table, model **a0** interleaves both `Max-Relative` and `GIN` operators. The former contributes to the model's representational capacity and compensates for the inadequacy of `GIN` operators in capturing long-range dependencies from the graph nodes features, ultimately leading the model to surpass baseline **b0**'s accuracy score (89.9% to 89.71%). On the other hand, the employment of `GIN` operator – alongside other factors – leads **a0** to achieve superior latency and energy efficiency scores. (*ii*) The additional varying architectural parameters from $\mathbb{A}$ (e.g., FFN-use) enable tuning the model's size and learning capacity to the task and dataset complexity. (*iii*) The distributed mapping strategies, as indicated by the *GPU-use* and *DLA-use* columns in Table 3.3, further balance the latency-energy trade-offs by effectively utilizing different CUs.

Hypervolume and Pareto Composition Analysis. To appraise the efficiency of our nested evolutionary search algorithm in identifying meaningful and mapping configurations, we compare its Hypervolume [166] against those of baseline OOE searches conducted on the *standalone* deployment options on the GPU and DLA. Succinctly, the Hypervolume measures the volume of the dominated area in the objective space by the estimated Pareto fronts. In Figure 3.5 (*left*), we can observe that the nested search (*w/IOE*) improves the Hypervolume scores over the baseline *OOE_GPU* search by ∼**5.7**% on average across the four (04) datasets, indicating the IOE's merit in extending the dominated area in the search space. In Figure 3.5 (*right*), we complement the Hypervolume analysis with a breakdown of the Pareto front composition with regard to the mapping strategies. Specifically, we consider the non-dominated solutions by combining Pareto fronts obtained at every generation. As seen, the *distributed* mapping options constitute 23.5%-53.7% of the solutions on the Pareto front, indicating their value in elevating resource efficiency for the various models.

Analysis of GNN workload distribution. We showcase how different GNN workload assignments across the GPU and DLA influence the latency-energy tradeoffs. In Table 3.4, we select one of the Pareto-optimal models, **Ours-a3** on CIFAR-100, and compare three mapping configurations: (*i*) Standalone options in which the model is fully deployed on either GPU or DLA. (*ii*) Constrained transition options (as introduced in [40]) where the number of allowable inter-CU transitions is limited to those that offer the best tradeoffs in order to mitigate data transmission overheads (i.e., the write-back and initial cold cache misses). (*iii*) Ours (IOE) are the mapping options provided through our IOE with unconstrained inter-CU transitions.

We ensure a fair comparison by focusing on one objective (energy) while fixing the other (latency). As such, for each constrained transition option, we use two (02) Pareto optimal solutions whose latency values are closest to our solution – i.e., solutions with latency closest to 17.29 ms. From the reported results in Table 3.4, we can observe that with our unconstrained mapping strategy, a single inference sustains 197.8 mJ on average, which is more

Table 3.4: Details and comparison of the GNN workload Assignment. 'G' and 'D' indicate GPU and DLA assignment, respectively. Note that each Grapher block is first succeeded by a corresponding FFN block.

Mapping option	**Stem**	**Grapher**	**FFN**	**Cls**	**#transit**	**Lat.**	**Enrg.**
DLA-only	D	D-D-D-D-D-D-D-D	D-D-D-D-D-D-D-D	D	0	25.56	**121.74**
GPU-only	G	G-G-G-G-G-G-G-G	G-G-G-G-G-G-G-G	G	0	**13.42**	273.22
constr-transit1	D	D-G-G-G-G-G-G-G	D-G-G-G-G-G-G-G	G	1	16.31	232.60
constr-transit1	G	G-G-G-G-G-D-D-D	G-G-G-G-G-D-D-D	D	1	17.42	226.79
constr-transit2	D	D-G-G-G-G-G-G-D	D-G-G-G-G-G-G-D	D	2	17.58	220.23
constr-transit2	G	G-G-D-D-D-G-G-G	G-G-D-D-D-G-G-G	G	2	17.11	227.15
Ours (IOE)	D	G-G-G-G-G-G-G-G	G-D-D-D-D-G-D-D	D	12	**17.29**	**197.8**

efficient than the best energy numbers, 226.79 mJ and 220.23 mJ, experienced by each of the other distributed mapping baselines, 'constr-transit1' and 'constr-transit2', respectively. The reasons for this improvement can be attributed to the following: (*i*) graph feature sizes are relatively small throughout the ViG models, leading to low inter-CU transmission overhead penalties to be experienced on the Xavier SoC. As Such, our IOE optimization strategy was able to exploit this property to identify more efficient mapping configurations with a larger number of transitions. (*ii*) Each computing block type within the ViG exhibits different affinities towards the underlying CUs. Thus, our IOE optimization strategy leveraged the other property of unconstrained transitions to map as many Grapher blocks to the GPU as feasible and as many FFN blocks to the DLA before transmission increase.

3.5.5 Generality and Scalability

We assess the scalability and generality of the IOE on the levels of: (*i*) ***the ViG architectural backbone***; where the supernet's backbone is implemented as a pyramid variant with dimensional reductions across the ViG superblocks, and (*ii*) ***the hardware CUs***; by simulating a case with 03 heterogeneous CUs.

On the ViG architectural level. We first compare the mapping results from the IOE between the isotropic (ViG-S) and pyramid (PyramidViG-M) variants on the Nvidia SoC. As we analyze the effectiveness of the inner EA, we fix the GNN from the OOE for both variants

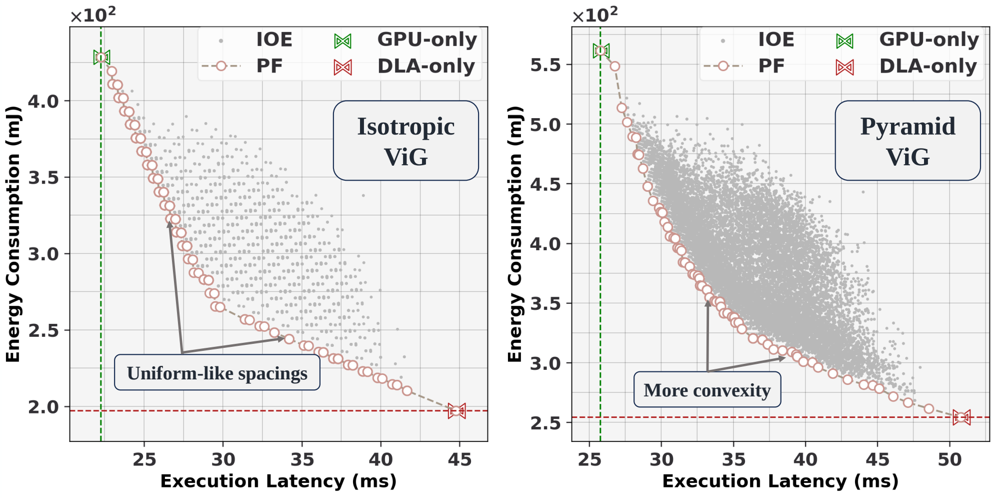


Figure 3.6: The results of the IOE EA optimization on the Isotropic Vision GNN (*left*) and Pyramid Vision GNN (*right*).

by setting the design parameters, $\mathbb{A}$, in Table 1 (i.e., d=4, *Graph Op*=`GIN`, fc_use=False, ffn_use=False, w=192), and specify an optimization budget of 2×10^4 evaluations.

As depicted in Figure 3.6, we can observe in the left subfigure that for the isotropic ViG, the explored mapping options follow well-defined spaced patterns between the two mapping extremes of *GPU-only* and *DLA-only*, offering almost uniform linear trade-offs between the energy efficiency and execution latency across various mapping options on the Pareto front. This results from the Grapher and FFN blocks being replicated throughout an isotropic architecture. As such, the performance evaluation of the different mapping options becomes predominantly influenced by the percentage of Grapher/FFN blocks assigned to each CU, irrespective of their order. However, for the PyramidViG on the right, this property does not hold as each Grapher/FFN block entertains different dimensions of their input and output features depending on its position, leading to varying performance characterizations, and a stronger convexity in the Pareto front reflecting a richer mapping space.

On the hardware CU level. Using the PyramidViG-M, we investigate how MaGNAS scales when the search space is further compounded with an increasing number of viable CUs. We simulate such use-case using MAESTRO tool [99] to specify 3 DSAs of diverse dataflows

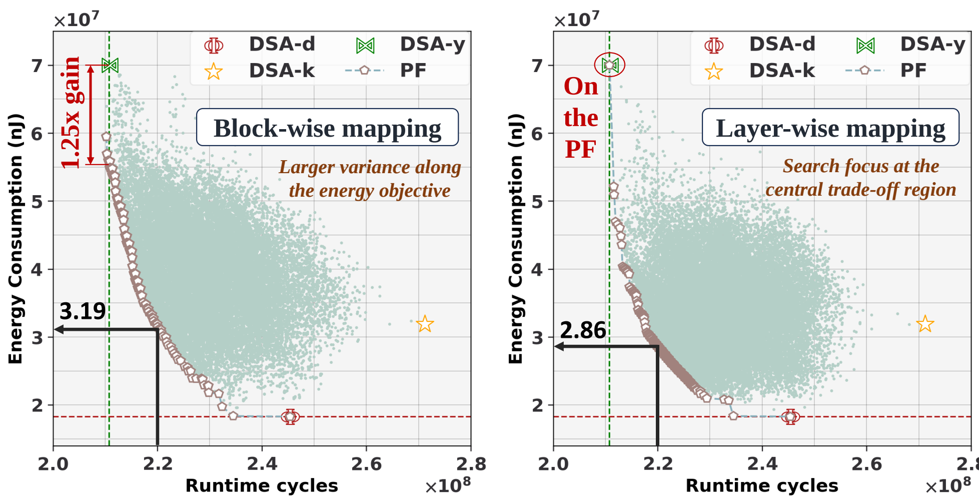


Figure 3.7: The results of the IOE optimization on MAESTRO [99] with: i) *Block-wise mapping granularity* (*left*) and ii) *Layer-wise mapping granularity* (*right*).

for CU heterogeneity. We also perform an additional search experiment on the *layerwise* granularity to assess further how the EA in the IOE scales when the number of mappable options dramatically increase. To provide context, the mapping space of the PyramidViG-M is $\mathcal{O}(1.72\times10^{12})$ in the *blockwise* using 2 CUs; $\mathcal{O}(1.67\times10^{16})$ in the *blockwise* using 3 CUs; and $\mathcal{O}(1.67\times10^{23})$ in the *layerwise* 3 CUs case, indicating a rising problem complexity.

In Figure 3.7, we demonstrate how the inner EA scales effectively as the search space is expanded from the *blockwise* to the *layerwise* mapping granularity. We specify a fixed optimization budget of 6×10^4 evaluations for both. In the blockwise case (*left*), the EA focuses on exploring more mapping solutions at the energy consumption extremes due to coarse-grained characterization of the Grapher block, leading it to identify a distributed mapping configuration that achieves 1.25× energy gains over DSA-y for the same latency. Contrarily for the *layerwise* search, where the EA was capable of recognizing benefits from distributing the *aggregation* and *combination* across different DSAs, leading it to concentrate the search more at the centralized latency-energy trade-off region. For example, at execution latency of $\sim 2.2\times10^8$ cycles, the layerwise search by the IOE was able to identify a mapping option that incurs 28.6 mJ compared to 31.9 mJ from the blockwise search.

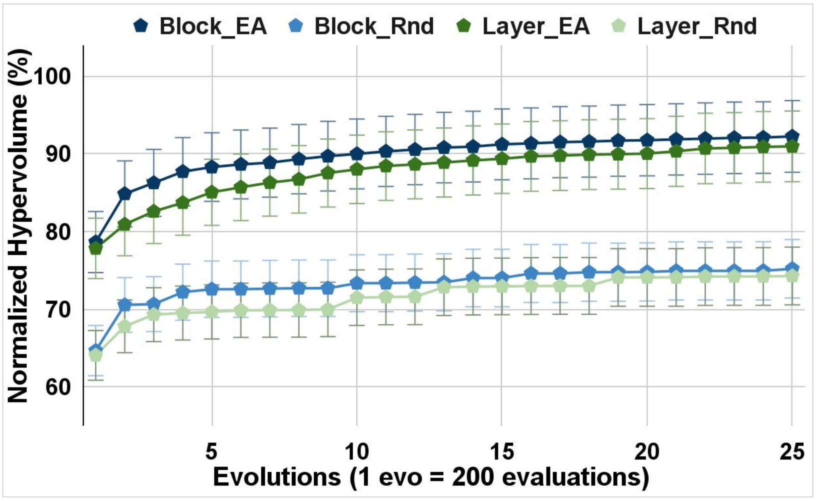


Figure 3.8: Evolutionary Vs. Random Search

On the power of evolution. We further analyze the hypervolume improvement when using an EA compared to a random search. We fix an optimization budget of 5000 evaluations for each and showcase the results in Figure 3.8 at different evolution stages for the mapping onto 3 CUs experiment. Normalized by a maximum achievable value from our previous results, we observe that the normalized hypervolume in the Figure reaches ∼92% improvement for the EA compared to ∼75% for the random search. We also notice that both *blockwise* and *layerwise* converge to proximate values despite the larger gap at the earlier evolutions (i.e., generations), further indicating the EA's capacity to scale.

3.6 Summary and Concluding Remarks

We have demonstrated through MaGNAS the merit of HW/SW co-design for the emerging ViG class of DNNs. Given the nascence of ViGs, the design, characterization and training of their supernets can only improve as the application of ViGs continue to proliferate. This is also seen through the push reconfigurable spatial accelerators to support new dataflows for the irregular graph computational sequences. Our approach is complementary to these efforts and remains generalizable to the broader applications of GNNs.

Chapter 4

Scheduling Multi-Model Workloads on 2.5D Chiplet-based Architectures

In this chapter, we study the scheduling optimization problem of multi-model AI models onto heterogeneous 2.5D multi-chip modules (MCMs). Multi-model workloads emerged toady as a result of modern trends in AI workloads (multi-tenancy, AR/VR systems) which constitute multiple, diverse models requesting simultaneous service. Whereas the 2.5D chiplet integration is gaining traction as a key technology in the post-Moore era from the edge to the cloud. Hence, we extend our HW/SW co-design studies to these technologies, characterizing this nascent scheduling space, and proposing a corresponding novel scheduling methodology.

4.1 Introduction

Recent artificial intelligence (AI) inference workloads have increased their scale in both of the model size (e.g., large language models [24, 187]) and the number of models deployed together (e.g., augmented and virtual reality; AR/VR [101]), which constructs multi-model workloads with heavier models than those in the past. Such trends led to heavy demands on compute capabilities in AI hardware from edge to cloud devices. As an approach to scale up the hardware for AI and increase the compute capability, chiplet-based multi-chip module (MCM) package has emerged as a promising solution [143, 167, 183, 199]. Such MCM packages facilitate the scaling of AI hardware based on their composability and cost-effectiveness, unlike monolithic designs, which are often constrained by fabrication yields, power, heat, and other engineering costs such as verification [133].

Researchers have actively explored the MCM for AI, focusing on the dataflow mapping (i.e., loop ordering, parallelization, and tiling) and workload orchestration onto chiplets considering the network-on-package (NoP) and other communication constraints [143,167,183,199]. For example, Simba [167] proposed a scalable MCM inference architecture that enables chiplets to either act as standalone inference engines or collaborate as groups for a layer. Although such works have successfully delivered promising performance and energy efficiency than monolithic designs, they mostly focused on *single-model* workloads targeting *homogeneous* chiplets. Unlike single-model workloads, multi-model workloads introduce major challenges to such homogeneous MCMs because of the ML operator heterogeneity (e.g., operator types and tensor sizes) and resulting diverse dataflow preferences [100]. Also, multi-model workloads often involve model level dependency and concurrency [101], which adds complex considerations to the scheduling problem.

Therefore, considering the new trend with multi-model AI workloads such as multi-tenancy [70, 104, 201] and AR/VR [101], we propose to explore heterogeneous chiplet-based MCM with AI accelerator chiplets with various dataflows to address the workload heterogeneity and concurrency. We consider inter-layer pipelining to enhance in-package data reuse and reduce offchip traffic. We formulate the scheduling problem and develop effective heuristics to navigate the huge scheduling space, whose problem scale is as big as O(10^{18}) on a 6x6 chiplet MCM AI accelerator system even running a single model (BERT-L).

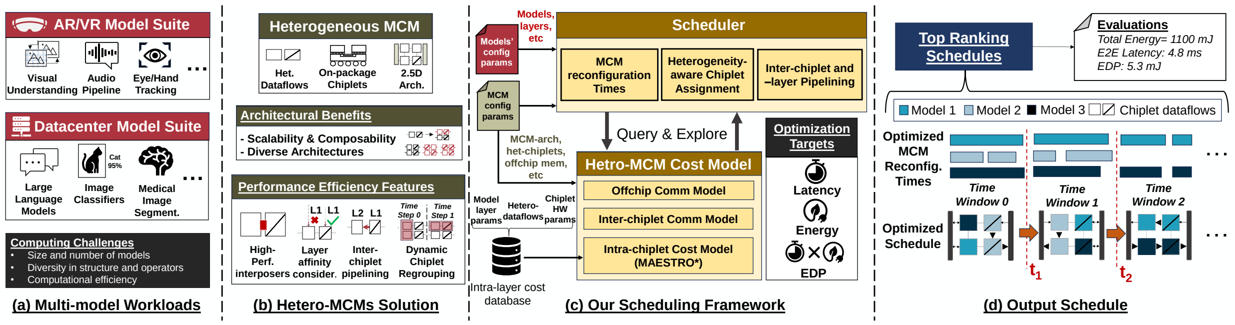


Figure 4.1: Proposed scheduling framework for multi-model workloads on 2.5D MCMs.

We evaluate five MCMs including three heterogeneous MCMs on ten multi-model scenarios: the first five scenarios are curated using MLPerf [158] to represent datacenter multi-tenancy scenarios. The models are selected based on recent datacenter model usage trends [70,83] and the trend of large language model adoptions (e.g., GPT-L [153]), future-proofing emerging AI workloads such as AI assistant [125]. The other five scenarios are curated for AR/VR usage scenarios from XRBench as a practical use case for edge multi-model workloads [101]. The evaluation results show the effectiveness of heterogeneous MCM combined with our scheduling method. Compared to the homogeneous MCM [167] running NVDLA [135] and Shi-diannao [45] style dataflows, heterogeneous MCM, on average, achieved 35.3% and 31.4% less energy-delay product (EDP) in each domain, respectively.

In brief, we summarize our contributions as follows:

- We propose to explore heterogeneous dataflow MCM for emerging AI workloads with multiple models running concurrently for the first time.
- We formulate the MCM AI accelerator scheduling problem into a multi-tiered optimization problem to address intractably large scheduling space.
- Based on the formulation, we develop a scheduler that thoroughly considers heterogeneous MCM and multi-model workloads. The scheduler employs advanced scheduling techniques, such as inter-layer pipelining, dynamic chiplet regrouping utilizing latest

representation such as the resource allocation tree [27].

- We codify our scheduling method and integrate it with a heterogeneous MCM AI accelerator cost model. We extend MAESTRO [98,99] to model the latency and energy of MCM accelerators.

- We analyze the costs and benefits of heterogeneous dataflow MCM using future-proof multi-model workloads motivated by recent industry use cases and present the importance of the scheduling problem.

4.2 Background and Preliminaries

We discuss examples of multi-model AI workloads and chiplet-based MCM AI accelerators.

4.2.1 Multi-model AI Workloads

The success of AI algorithms in individual tasks (e.g., hand tracking, depth estimation, and speech recognition) led to the emergence of multi-model AI workloads, which include multi-tenant workloads at data centers [70, 104, 201] and real-time multi-model workloads such as AR/VR [101]. We summarize example multi-model AI workloads from industrial use cases in Table 4.2. The models in such workloads are diverse in terms of the tasks and input modalities. For example, an industrial data center multi-tenant AI workload suite [70] includes a face recognition model based on support vector machine, recommendation models based on multi-layer perceptron, and a speech recognition model based on recurrent neural network (RNN). More recent workloads in data center AI workload include large language models [126], which adds more heterogeneity to the multi-model AI workloads. As discussed in prior works [100, 101], such multi-model workloads involve high heterogeneity in AI operators (or layers), which is one of the major challenges to accelerators that specialize the architecture and dataflow for a specific set of workloads.

4.2.2 MCM AI Accelerators

Multi-chip Modules (MCM) comprise small functional dies (chiplets) that are packaged together to build a larger system. Chiplets are interconnected via on-package links typically through silicon interposer or organic substrates to create a network-on-package (NoP) [15, 89, 193]. A typical chiplet, in the context of a DNN accelerator, comprises off-chip memory, a global shared memory, and an array of processing elements connected via a Network-on-Chip (NoC) [29]. Advantages of the chiplet-based MCM architecture include the modularity and scalability to systems of varied scales simply by adjusting the number of chiplets placed on the package as well as low verification cost [133]. Based on such benefits, many chiplet-based MCMs have been developed for scalable DL inference [28, 143, 167, 181]. Such MCM accelerators successfully scaled up the systems up to 256 chiplets with 1 million processing engines [143]. However, the effectiveness of a chiplet-based MCM system heavily depends on the careful distribution of computation amongst the different chiplets while balancing the added NoP/NoC communication costs.

4.2.3 Scheduling space

As discussed in previous works [27, 100, 147], scheduling AI workloads on an accelerator can be considered as assigning a set of computations in various granularity (e.g., layer or compute tile) to each compute unit and ordering the computation. That is, the scheduling process is spatially and temporarily partitioning a workload onto a target accelerator architecture [27]. However, with that formulation, the scheduling space of multi-model workloads onto shared MCM accelerators is intractably large and high-dimensional, as discussed in section 4.1.

One approach to address the complexity is formulating the problem into multi-level decision problem where each decision subspace is a tractable problem [29, 100].We adopt a similar approach and formulate the multi-model workload scheduling on MCM as multiple-level decision problem, as shown in Figure 4.2. We discuss details of our problem formulation in detail and performance modeling methodology next.

4.3 System Modeling and Problem Formulation

To develop a systematic approach to navigate complex scheduling space, we formulate the scheduling problem of multi-model workloads on a heterogeneous MCM AI accelerator.

4.3.1 Base Formulation

To formulate the MCM scehduling problem, we first define multi-model workload scenario (Sc) and MCM hardware (H).

We formulate the workload in the granularity of layers in each model. Therefore, we formulate a multi-model workload scenario (Sc) as the collection of layers in the models included in the scenario. Letting the number of models included in Sc as $|Sc|$ and the number of layers included in a model m as $|m|$, we define Sc as follows:

Definition 4.1. ***Multi-model Workload Scenario (Sc)***

$$Sc = \{layer_{i,j} | 0 < i \leq |Sc|, 0 < j \leq |m_i|\}$$

where $layer_{(i,j)}$ refers to the j-th layer of model i in Sc.

AI accelerator chiplets consist of a PE array, memory, and on-chip interconnection among memory and PEs. In addition to them, we also include the dataflow in the formulation to model heterogeneous chiplet MCM AI accelerator. Accordingly, we define an AI accelerator chiplet (c) as follows:

Definition 4.2. ***AI Accelerator Chiplet (c)***

$$c = \{df, N_{PE}, BW_{noc}, BW_{mem}, Sz_{mem}\}$$

In 4.2, df refers to the dataflow, N_{PE} is the number of PEs, BW_{noc} is the NoC bandwidth, BW_{mem} is the chiplet-level shared memory bandwidth, and Sz_{mem} is the memory size in c.

Based on the definition of the chiplet, we formulate the MCM accelerator as the set of chiplets ($C = \{c_1, c_2, ..., c_{N_{cpl}}\}$), NoP, and off-chip interface as follows:

Definition 4.3. ***MCM AI Accelerator (H)***

$$H = \{C, BW_{offchip}, BW_{nop}\}$$

We assume the 2D mesh topology for NoP like Simba [167], and chiplets on two sides (left and right) of the packages have off-chip interfaces.

4.3.2 Workload Partitioning Space

To reduce the complexity of the scheduling problem, we adopt a multi-level scheduling method, which splits the end-to-end workload defined in the layer granularity into coarse-grained layer groups, termed as the *time window*. Figure 4.2 shows an example of the time window that contains six layers from Model A and five layers from Model B.

A time window (tw) is defined by the start time and the duration (T_S and T_{tw}) and a set of assigned layers to the time window, as shown in 4.4.

Definition 4.4. ***Time Window (tw)***
For a target workload scenario Sc, a time window tw is defined as follows:

$$tw(Sc) = (T_s, T_{tw}, L)$$

where $L = \{l | l \in Sc\}$

The time window describes a set of layers to be executed on an MCM AI accelerator package, which is used for describing package level scheduling. For each chiplet, we define a finer-grained group of layers within a time window. We term the sub-set of layers within a time window as *segment*.

Definition 4.5. ***Segment (sg)***
For a time window $tw(Sc)$ and its layers $L(tw(Sc))$, the segment $sg(tw(Sc))$ is defined as follows:

$$sg(tw(Sc)) = \{l | l \in L(tw(Sc))\}$$

To develop a systematic optimization algorithm for layer segmentation within each time window, we need to define the conditions of valid layer segments. We define the condition as follows:

Theorem 4.1. ***The validity of segments in a time window***
For a time window $tw(Sc)$ and its layers $L(tw(Sc))$, let the set of all segments for $tw(Sc)$ be SG, then SG is valid if the following condition is satisfied:

$$\bigcup_{sg \in SG} sg = L(tw(Sc)) \land \forall sg_i \neq sg_j \in SG, sg_i \cap sg_j = \emptyset$$

Theorem 4.1 states two conditions (1) the set of segments needs to cover all the layers in their time window for completing assigned layer computations for the time window and (2) all segments are exclusive to prevent redundant computing. The same idea extends to the time window as follows:

Theorem 4.2. ***The validity of time window partitioning***
For a multi-model workload Sc, its layers $L(Sc)$, and the set of time windows $TW(Sc)$, $TW(Sc)$ is valid if the following condition is satisfied:

$$\bigcup_{tw \in TW(Sc)} tw = L(Sc) \land \forall tw_i \neq tw_j \in TW(Sc), tw_i \cap tw_j = \emptyset$$

Both Theorem 4.1 and Theorem 4.2 indicate that the time windows and segments need to be partitions of the workload and time window layer, respectively. Combining all definitions in this section, we formulate the workload partitioning space into the time window and segment as follows:

Definition 4.6. ***Workload Partitioning Space***
For a multi-model workload Sc, the time window partitioning space ($Sp_{tw}(Sc)$) and the layer segmentation space for a time window ($Sp_{sg}(tw)$) are defined as follows:

$$Sp_{tw}(Sc) = \mathbb{P}(L(Sc))$$
$$Sp_{sg}(tw) = \mathbb{P}(L(tw))$$

where $\mathbb{P}(A)$ refers to all possible partitioning of a set A

4.3.3 Scheduling Space

A segment contains layers to be executed on a chiplet. Therefore, spatial (i.e., which segment runs on which chiplet) and temporal mappings (i.e., execution order of segments on each chiplet) of segments construct the scheduling space within each time window when segments are determined. Therefore, the scheduling space within a time window $tw(Sc)$ can be defined:

Definition 4.7. ***Scheduling Space in a Time Window*** ***(***SS_{TW}***)***
For a given time window $tw(Sc)$ *and a target MCM accelerator hardware* H*, the scheduling space within the time window (*$SS_{TW}(tw(Sc), H)$ *is defined as follows:*

$$\begin{aligned} &SS_{TW}(tw(Sc), SG, H) \\ &=\{(sg, c, j) | sg \in SG \land c \in C_H \land j \in \mathbb{N} \land valid(sg, tw(Sc))\} \end{aligned}$$

where C_H *refers to the set of chiplets in* H *and val(sg, tw(*Sc*)) indicates the validity of sg for* $tw(Sc)$

Each entry in SS_{TW} describes the spatial and temporal mapping of a segment (sg). Spatial mapping can be defined as the target chiplet to execute sg. Accordingly, a target chiplet (c) is specified for sg. The temporal mapping is defined as the execution order. Therefore, a natural number j is used to represent the execution order. Note that the execution order is defined separately on each chiplet. Based on 4.7, we can define the entire scheduling space as the collection of that in each time window.

Definition 4.8. ***MCM Scheduling Space for a Multi-model Workload*** ***(***$SS_{Sc}(H)$***)***
For an MCM AI accelerator (H*) and a multi-model workload (Sc), the scheduling space*

$(SS_{Sc}(H))$ is defined as follows:

$$\begin{aligned} SS_{Sc}(H) = &\{(TW, SG_{TW}, (SS_{TW}(tw, SG_{TW}(tw), H))| \\ &TW \subset Sp_{tw}(Sc) \land SG_{TW}(tw) \subset Sp_{sg}(tw) \\ &\land tw \in TW\} \end{aligned}$$

where SG_{TW} refers to the set of layer segments for each time window in TW

4.8 defines the entire scheduling space of an MCM AI accelerator for a multi-model workload as the cross-product of all possible time window partitioning, layer segmentation for each time window, and corresponding scheduling space within each time window.

4.3.4 Scheduling Problem

Based on 4.8, we define a schedule instance as the collection of spatial and temporal mapping for given valid time windows (TW) and segments for each time window (SG_{TW}).

Definition 4.9. ***MCM Schedule***
A schedule instance $(sched(Sc, H))$ is defined as follows:

$$\begin{aligned} sched(Sc, TW, SG_{TW}, H) = &\{(TW, SG_{TW}, s) | valid(TW, Sc) \\ &\land \forall tw \in TW : valid(SG_{TW}(tw), tw) \\ &\land s \in SS_{TW}(tw, SG_{TW}(tw), H)\} \end{aligned}$$

where SG_{TW} refers to the set of layer segments for each time window in TW

Using 4.9, we formulate the scheduling problem as a minimization problem of an optimization metric of choice (e.g., latency and energy), as follows:

Definition 4.10. ***MCM Scheduling Problem***

$$\underset{TW, SG_{TW}, Sched}{\text{argmin}} \; OptMetric(TW, SG_{TW}, Sched, H)$$

where $Sched = sched(Sc, TW, SG_{TW}, H)$

The optimization metric can be chosen by users depending on the use case. In our scheduler, we adopt a comprehensive and customizable score that thoroughly consider all of latency, energy, and energy-delay product (EDP), allowing users to configure their own optimization metrics, which can the mentioned frequently used metrics or a user-defined function that takes a schedule instance and generates a custom metric.

4.3.5 Latency Modeling

To develop a scheduler based on the scheduling problem formulation in subsection 4.3.3, we need to be able to evaluate each schedule on target MCM AI accelerator hardware. For that, we extend MAESTRO [98] to the chiplet domain and model the latency of MCM AI accelerators concurrently executing multi-model workloads on a shared MCM system in a bottom-up fashion. We discuss our latency evaluation methodology in detail, focusing on our extension for the MCM and multi-model workloads.

Layer Latency. The latency incurred by an individual layer, l, mapped onto an accelerator chiplet is defined as:

$$Lat(l) = Lat^{ip\_com}(l) + Lat^{comp}(l) + Lat^{op\_com}(l)$$

$Lat^{comp}(l)$ being the layer computation cost dependent on the AI accelerator chiplet parameters 4.2; $Lat^{ip\_com}(l)$ is latency incurred from loading the layer operands (input activations and weights), and $Lat^{op\_com}(l)$ from transmitting the output activation to a subsequent layer. As for Lat^{com}, it is defined as:

$$Lat^{com} = \begin{cases} 0, \text{if same chiplet} \\ \frac{Sz_{data}}{BW_{nop}} + n_{hops} \times Lat_{hop} + \delta, \text{if same package} \\ \frac{Sz_{data}}{BW_{mem}} + n_{hops} \text{ x } Lat_{hop} + Lat_{mem} + \delta, \text{if offhcip} \end{cases}$$

where assuming sufficient memory for double-buffering on each chiplet accelerator, communication costs become incurred when transmitting data to/from another chiplet on package or the offchip memory. The first term $\frac{Sz_{data}}{BW}$ reflects transmission latency; the second term is captures propagation latency across n_{hops} between the source and destination; δ is an additional latency term for potential NoP traffic conflicts; Lat_{mem} is the cost from read/write access of data at the offchip memory.

Time Window Latency. We first model a layer segment's latency in a time window as:

$$Lat(sg) = \sum_{n=1}^{N} Lat^{comp}(l_n) + Lat^{ip\_com}(sg) + Lat^{op\_com}(sg)$$

The first term represents the sum of individual layer computational latencies; Lat^{ip_com} is the initial external data transfer loading costs of necessary off-chiplet input activation and parameter weights; Lat^{op_com} is the transmission latency from transmitting segment output data to the next segment or writing back to memory.

Based on the segment's latency, the time window latency can be defined as:

$$Lat(tw) = \max_{SG_m \subset SG} \begin{cases} \sum_{sg_k \in SG_m} Lat(sg_k), \text{if end-to-end} \\ \max\limits_{sg_k \in SG_m} Lat(sg_k), \quad \text{if pipelining} \end{cases}$$

where SG_m represents the set of segments in a time window associated with a model m.

The latency of SG_m can either be the maximum out of all m segments if pipelining or their summation in the base case. The latency of the entire window is the max out of all $SG_m \subset SG$.

Overall Latency. The overall Scenario latency can then be estimated as the aggregate across all time windows:

$$Lat(Sc) = \sum_{tw_j \in TW} Lat(tw_j)$$

4.3.6 Energy Modeling

Albeit similar to latency, Energy costs are always aggregated. The base communication energy cost is defined as:

$$E^{com} = \begin{cases} 0, & \text{if same chiplet} \\ Sz_{data} \times E_{tx\_bit} \times n_{hops}, & \text{if same package} \\ Sz_{data} \times E_{tx\_bit} \times n_{hops} + E_{mem}, & \text{if offchip} \end{cases}$$

where the energy incurred from moving data across the package is equal to the product of data size, number of hops, and the per-bit transmission energy (E_{tx_bit}). In case of an offchip data transmission, the cost of memory access E_{mem} is added.

The overall energy consumption across the entire time windows can be computed as the sum of energies of its constituent components as follows:

$$E(l) = E^{ip\_com}(l) + E^{comp}(l) + E^{op\_com}(l)$$
$$E(tw) = \sum_{sg_k \in SG} E(sg_k), \quad E(Sc) = \sum_{tw_j \in TW} E(tw_j)$$

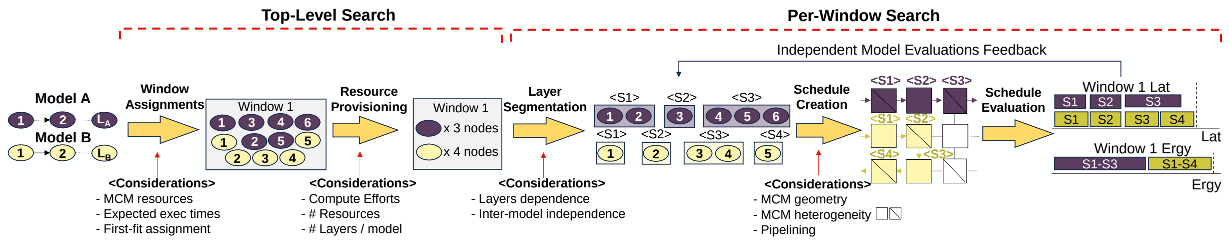


Figure 4.2: An overview of our scheduling algorithm.

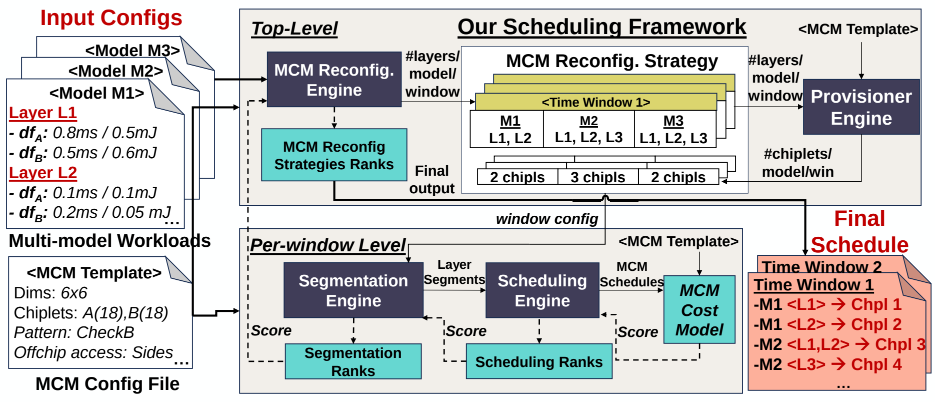


Figure 4.3: Our proposed multi-model scheduling framework on heterog. MCM.

4.4 Scheduling Framework

We discuss our scheduling framework for multi-model workloads on heterogeneous MCMs based on the hierarchical search space characterization and problem formulation in section 4.3. As illustrated in Figure 4.2, our scheduling algorithm is a two-level approach: top-level and per-window searches. Top-level search is responsible for selecting layers in each model to be scheduled within a time window and determining the initial number of chiplet nodes for each model. Per-window search explores the spatial and temporal partitioning (i.e. tiles or layer segments) of the layers in each model at the chiplet granularity. To explore the chiplet granularity tiling space, we generate valid inter-chiplet-pipelined schedules utilizing a scheduling tree structure inspired by the RA Tree [27]. Each schedule is evaluated using our custom heterogeneous MCM cost model which provides feedback to the chiplet level tiling (”layer segmentation” in Figure 4.2) with expected metrics (latency, energy, EDP, etc.).

We codify our scheduling algorithm into a software framework, as illustrated in Figure 4.3. As inputs, our scheduling framework receives (1) description files of the multi-model workloads (layer parameters, topology, dependencies, etc.) and (2) a description file of the MCM hardware specification, such as the number of chiplets, the shape, and dataflow organization of the chiplet arrays, NoP bandwidth, on-chiplet memory size, and so on. As outputs, our scheduling framework reports an optimized schedule with expected metrics such as latency, energy, EDP, or other user-defined metrics as a combination of latency and energy. Our framework consists of four software *engines*, which handle each step of the scheduler discussed in this section. Each engine is responsible for each step of our two-level scheduling method as illustrated in Figure 4.2. We discuss each engine and our cost model next.

4.4.1 MCM Reconfiguration Engine ***(MCM-Reconfig)***

The *MCM-Reconfig* engine at the top-level step receives the multi-model workload descriptions with layer information in each model, layer dependency, and expected latency and energy of each layer on each chiplet class offline-analyzed by MAESTRO [98]. The *MCM-Reconfig* engine is responsible for the window assignment in Figure 4.2, which (1) generates candidate time window partitioning strategies via sampling a set of discrete points in time reflecting the boundary points between execution windows and (2) assigns layers from models to each time window. As the final assignment of layers to chiplets is not yet known, the decisions in *MCM-Reconfig* engine are based on expected execution times. Formally, given $|DF|$ dataflow style classes, the expected execution latency for a layer l is:

$$\mathbb{E}(Lat(l)) = \sum_{i=1}^{|DF|} \frac{n_{df_i}}{|\mathcal{C}|} \times Lat(l \to i) \tag{4.1}$$

where n_{df_i} indicates the number of class i chiplets integrated onto the MCM having $|C|$ chiplets in total; $Lat_{l \to i}$ is layer l latency when scheduled on the class i chiplet, which is retrieved offline from latency database generated by MAESTRO [98, 99]. The average execution time information is utilized in *MCM-Reconfig* for window assignment in Figure 4.2.

Time Windows Characterization. *MCM-Reconfig* engine first specifies the number of windows, which is the coarse-grained scheduling granularity in our scheduling algorithm. We define n_{splits} as a user-defined parameter to characterize the number of time windows and explore proper cut points for each model. For example, in Figure 4.2, the model A has a cut after layer 6, which led to having layers 1-6 in Window 1. The worst-case latency experienced by a model in the multi-model workload is set as the time horizon which we partition into periodic time windows. Based on preliminary analysis, we set n_{splits}=4 (5 time windows) as our default unless otherwise stated. Ablation is performed in the evaluation Section 4.5.

Greedy Layer Packing Algorithm. Multi-model workload introduces a challenge: the time window boundary determined by the cut points of one model might not be aligned with other models. Therefore, we adopt a *first-fit greedy-packing* heuristic where layers are assigned to an execution window if their execution time is expected to be within a time window (see Algorithm 1), even if the start and finish time is not aligned with those of a time window. Any layer with execution time lies across two time windows is deferred to the next time window. This approach not only solves the time window - layer execution time misalignment problem, but also facilitates (i) running low-latency layers in earlier windows, preventing starvation of small workloads blocked by heavy workloads. (ii) Dynamically controlling number of time windows by skipping trivial windows without any workloads.

4.4.2 Provisioner Engine *(PROV)*

The *PROV* engine is responsible for providing an initial estimate on the number of chiplet needed by each model workload in every time window given a candidate partitioning strategy. This assignment is agnostic to the underlying chiplets' resources or dataflow, and hence we

Algorithm 1 Greedy Layer Packing Algorithm

```
Input: $M$ (workloads), $\mathcal{T}$, $\mathcal{C}$, $DF$
Output: $L2W$ (Layer(s) to windows assignments)
 1: Function LayerAssignment($M$, $\mathcal{C}$, $\mathcal{T}$)
 2:  for $m \in M$ do
 3:    $exec\_win = ()$
 4:    $win\_idx, used\_cycles = 0, 0$
 5:    for $l \in m$ do
 6:      $\mathbb{E}(Lat(l)) = \sum_{i=1}^{|DF|} \frac{n_{df_i}}{|\mathcal{C}|} \times Lat(l \to i)$
 7:      while True do
 8:        if $win\_idx == |\mathcal{T}|$ then
 9:          $Slack = None$
10:        else
11:          $Slack = \rho[win\_idx] - used\_cycles$
12:        if $Slack == None$ or $\mathbb{E}(Lat(l)) <= Slack$ then
13:          $exec\_win$ += $(l, )$
14:          $used\_cycles$ += $\mathbb{E}(Lat(l))$
15:          Break
16:        else
17:          $L2W[win\_idx][m] = exec\_win$
18:          $used\_cycles = \mathcal{T}[win\_idx]$
19:          $exec\_win = ()$
20:          $win\_idx$ += 1
21:    $L2W[win\_idx][m] = exec\_win$
```

refer to chiplet resources in this state as *nodes.*

We implement our $PROV$ engine for nodes' distribution across various model workloads using a set of rules. The rules are based on expected latency, energy, EDP, or user-defined metric for each corresponding window. This computational effort is associated with a specified performance optimization goal, denoted as $\mathcal{P}$ where $\mathcal{P} \in \{Lat, Ergy, EDP\}$, Following a uniform distribution rule, the number of nodes N_i allocated to the i^{th} model is:

$$N_i = round(\frac{\mathbb{E}(\mathcal{P}_i)}{\sum_j(\mathbb{E}(\mathcal{P}_j)} \times |\mathcal{C}|) \tag{4.2}$$

where $\mathbb{E}(\mathcal{P}_i)$ represents the expected value of the performance optimization goal for model, computed in a manner similar to the expected execution latency formula in (4.1), whereas the sum term in the denominator represents the sum of all expected values for every model workload in the current time window. Though in principle other allocation strategies can be implemented for the Provisioner, the benefits of having a rule-based Provisioner are twofold:

- The Provisioner becomes specialized in warranting a *fair spatial distribution* of resources per window across the various model workloads, leaving temporal allocation tasks to be handled through the other engines.
- Circumventing an additional top-level search space specification with a high degree of uncertainty, acting instead as a regulator mid-way throughout the framework without aggravating the search complexity.

To ensure the progression of all model workloads assigned to a window, we enforce the allocation of at least one resource per model per window to account for rounding errors in Equation 4.2 when a model workload assigned to the window has negligible computational overhead compared to its peers. The reallocation process iteratively reassigns nodes from models with max number of resources until the constraint is satisfied.

4.4.3 Segmentation Engine *(SEG)*

As the first module in the per-window level, the *SEG* module is instantiated for every time window, receiving topologically sorted sets of layers from each model to be further partitioned into segments. Segmentation is the process of partitioning a set of layers into smaller subsets of layers (i.e., segments or tiles) that can be mapped to a computing resource for exclusive execution throughout the duration of a time window. Different segmentation choices reflect various trade-off points between the *layer-sequential* and *layer-pipelining* execution features: the former controls the granularity of layers within each segment to be co-located for sequential execution on a single chiplet resource; the latter exploits inter-layer and -chiplet pipelining opportunities between various segments.

Segmentation Search Space. As per our formulation in Section 4.3, a segmentation candidate is represented by a sequence of splitting points, where candidate splitting points are specified after each layer for each model's set of layers provided to the *SEG* Given $|L_i|$ and $|N_i|$ as the respective number of layers and number of assigned nodes (from *PROV*) for a model workload m_i, the max number of segments that can be generated for m_i becomes upper bounded by N_i. Hence, the overall segmentation space complexity is $\mathcal{O}(\Pi_i\ |L_i| \times |N_i|)$, with the Π_i indicating the combinatorial space across all models. To aid in managing the rising multi-model segmentation space complexity, we introduce the following heuristics.

Heuristic 1. Product to summation reduction. We enable *SEG* to navigate the segmentation search space with reduced complexity through a two-step process: (1) *SEG* leverages the independence of segments from different models to initially explore the segmentation subspace for each model *separately* (2) Segmentation point candidates from the top-k configurations for each model are used to construct a smaller search space for the combinatorial co-exploration of the segmentation space. Through this heuristic, the search space complexity can be reduced from $\mathcal{O}(\Pi_i\ |L_i| \times |N_i|)$ to $\mathcal{O}(max(|L_i| \times |N_i|))$.

Heuristic 2. Node allocation constraint. We designate an additional user-specified constraint to restrict the number of nodes provisioned to a model within a window based on its number of layers. The constraint is beneficial in cases where there is mismatch between the multi-models' distribution of computational efforts across their respective layers. For instance, a scenario may arise where a single compute-intensive layer may be assigned to same window alongside dozens of layers from smaller models, leading to an unnecessary explosion in the segmentation search space. *PROV* is responsible for enforcing this constraint.

4.4.4 Scheduling Engine (*SCHED*)

The innermost Scheduling engine (*SCHED*) is responsible for generating the final mapping of layer segments to the physical chiplet accelerators on the target MCM.

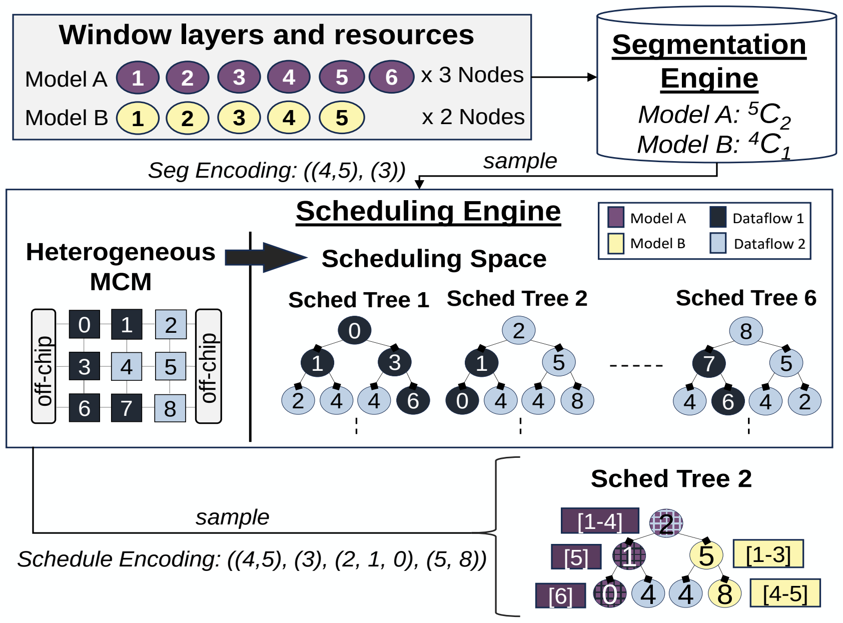


Figure 4.4: Schedules creation through the *SEG* and *SCHED* engines.

Scheduling Search Space. As illustrated in Figure 4.4, the scheduling search space for the mapping of M model workloads onto $\mathcal{C}$ chiplets can be represented as a forest of scheduling trees. Throughout this sub-section, we use three terms to describe different parts of the scheduling search space: (i) *forest*; as the entire collection of search trees. (ii) *tree*; characterizing a single scheduling tree modeling with all the M models involved. (iii) *subtree*; representing a subset part of each tree exclusively associated with a model $m_i \in M$.

Scheduling Tree Composition. every node in a scheduling tree corresponds to a unique chiplet resource on the MCM showcasing its distinctive heterogeneous features (i.e., dataflow). Each chiplet is assigned a unique identifier based on a row-major order traversal across the MCM grid. Tree edges are constructed based on each chiplet's XY neighbors connected directly through an interposer. Though a node j can be replicated throughout the tree, it can only be visited once, indicating its exclusive occupancy by a model.

Trees Distinction. within each tree, the root nodes of the subtrees specify different chiplets as potential starting positions for candidate model schedules. This is particularly relevant when the underlying pattern of heterogeneity is non-uniform, causing scheduling options to be dependent on the starting position (see Figure 4.4). Thus, the scheduling space coverage starts by selecting a tree, represented by a permutation sequence of subtrees' root nodes – e.g., permutation sequence [i,j,k] indicates exploring scheduling candidates for a tree with scheduling candidates starting at chiplet positions i, j, and k for a 3-model workload. The depth of model i's subtree is determined by its number of provisioned resources N_i.

Candidate Schedules Generation. Through traversing each subtree, we can obtain candidate execution schedules for each model by assigning segments orderly to the subtree's nodes. Starting from the root node of the first model's subtree, a constrained depth first search (DFS) is performed generating a candidate schedule path once the full subtree depth (N_i) has been reached. This traversal is repeated for each subsequent subtree, constrained on the preceding subtree's prior visited nodes. Traversal paths from each subtree are then aggregated to form the overall scheduling candidate.

Encoding and Search Algorithm. As shown in Figure 4.4, use a $2 \times |M|$-length tuple to represent the final scheduling encoding, where the first M entries reflect segmentation decisions for each model m_i, and the latter $|M|$ entries reflect schedule mappings of segments to chiplets for each workload. This form of encoding facilitates supports having different search algorithms for each engine. We tested both brute-force and evolutionary algorithms as will be shown in section 4.5.

Schedules Starting Positions. We constrain the number of scheduling trees to chiplet nodes that satisfy either of the following two conditions: (i) chiplets that maintain a direct link to an offchip DRAM memory interface (as in the right and left-most chiplets of the MCM in Figure 4.4). (ii) ending chiplet positions from the preceding window, which enable leveraging data locality across time windows.

Search Space Complexity. Given $|M|$ as the number of models in a given window, $|T|$

Table 4.1: MCM microarchitecture parameters from [143, 167]. All numbers are scaled to 28 nm process node technology.

Offchip Memory	DRAM latency	200 ns
	DRAM energy	14.8 pJ/bit
	DRAM bandwidth	64 GB/s
Package	NoP intercon. latency	35 ns/hop
	NoP intercon. energy	2.04 pJ/bit
	NoP intercon. bandwidth	100 GB/s/Chiplet

the number of scheduling trees in the search space, d is a traversal path's degree of freedom, and N_{max} representing the max number of resources allocated to any model in this window. The scheduling search complexity can be given by $\mathcal{O}(|M| \times |T| \times d^{N_{max}})$.

4.4.5 Cost Model and Scoring

We implement a cost model for evaluating scheduling candidates on different performance efficiency metrics.

Cost Model. The overall cost model constitutes three distinctive cost model components: *offchip communication model*, *inter-chiplet communication model*, and *intra-chiplet cost model*. We follow our latency and energy modeling characterization in subsection 4.3.5 and subsection 4.3.6, and use the architectural parameters provided in [143, 167] for the offchip and inter-chiplet communication costs as shown in Table 4.1. For the intra-chiplet cost model, we utilize the open-source accelerator cost model, MAESTRO [98, 99], to evaluate intra-chiplet performance based on a chiplet's underlying dataflow and hardware parameters.

Scoring. Scores are estimated based on latency, energy, or EDP metrics following Section 4.3 modeling. The *SCHED* aggreages scores for each model's schedule, and returns the top performing configuration to the *SEG* engine to rank segmentation strategies. Top segmentation strategies in each window are aggregated to score the overall scheduling strategy at *MCM-Reconfig* (see the scoring flow in Figure 4.3).

Table 4.2: Our Multi-model workload scenarios for datacenter and AR/VR use-cases inspired by MLPerf [129, 158] and XRbench [101] benchmarks. 'sl' indicates sequence length

Use-Case	Scenario	Models	Batch Size
Datacenter (MLPerf) [129, 158]	(1) LMs	GPT-L [153] (sl=128)	1
		BERT-L [43] (sl=128)	3
	(2) LMs + Image (light)	GPT-L [153] (sl=128)	1
		BERT-L [43] (sl=128)	3
		ResNet-50 [71] ($224\times224\times3$)	1
	(3) LMs + Image (heavy)	GPT-L [153] (sl=128)	1
		BERT-L [43] (sl=128)	3
		ResNet-50 [71] ($224\times224\times3$)	32
	(4) LMs + Segmentation + Image (heavy)	GPT-L [153] (sl=128)	8
		BERT-L [43] (sl=128)	24
		U-Net [161] ($512\times512\times1$)	1
		ResNet-50 [71] ($224\times224\times3$)	32
	(5) LMs + Segmentation + Image (heavy)	GPT-L [153] (sl=128)	8
		BERT-L [43] (sl=128)	24
		BERT-base [43] (sl=128)	24
		U-Net [161] ($512\times512\times1$)	1
		ResNet-50 [71] ($224\times224\times3$)	32
		GoogleNet [179] ($224\times224\times3$)	32
AR/VR (XRBench) [101]	(6) AR Assistant	D2GO [124] (Object Det.)	10
		PlaneRCNN [111] (Plane Det.)	15
		MiDaS [154] (Depth Est.)	30
		Emformer [171] (Speech Rec.)	3
		HRViT [51] (Semantic Seg.)	10
	(7) AR Gaming	PlaneRCNN [111] (Plane Det.)	15
		Hand S/P [62] (Hand Track.)	45
		MiDaS [154] (Depth Est.)	30
	(8) Outdoors	D2GO [124] (Object Det.)	30
		Emformer [171] (Speech Rec.)	3
	(9) Social	EyeCod [219] (Gaze Est.)	60
		Hand S/P [62] (Hand Track.)	30
		Sp2Dense [117] (Depth Ref.)	30
	(10) VR Gaming	EyeCod [219] (Gaze Est.)	60
		Hand S/P [62] (Hand Track.)	45

4.5 Evaluation

4.5.1 Experimental Settings

Multi-Model Workloads. Our evaluations are performed on multi-model workload scenarios based on models from (i) MLPerf inference benchmark [129, 158] for the datacenter multi-tenancy setting; (2) XRBench [101] for AR/VR workloads. The full list of scenarios is provided in Table 4.2 covering a wide range of use-cases with varying degrees of diversity and complexity. We implement our MLPerf following datacenter usage trends in [70,83,148].

MCM System. We follow Simba's on-package chiplet arrangement to implement our MCM experimental templates [167]. Simba comprises a total of 36 chiplets connected through a Mesh topology and arranged as four 3×3 groups of chiplets. We implement (1) 3×3, and (2) 6×6 MCM templates for our experiments. For each, we adopt XY routing for on-pacakge data movement, and integrate further memory interfaces on the sides of the outer chiplets, providing direct links to the offchip DRAM via double-sided memory channels as in [58]. We consider 4096 PEs/chiplet and 256 PEs/chiplet for the datacenter and AR/VR settings, respectively. We set the L2 shared memory size in each chiplet to 10 MB, inspired by the on-chip memory size in a recent mobile accelerator [152].

Baselines and Heterogeneity Patterns. We choose Shi-diannao [45] and NVDLA [135] dataflow styles for our accelerator chiplets. We accordingly implement two baselines:

- *Standalone.* Each model in a multi-model workload is assigned a single chiplet for execution, all chiplets posses the same dataflow.
- *Simba-like Pipelining.* In each time window, Model workloads can be assigned to more than one chiplet to leverage pipelining benefits. All chiplets posses the same dataflow.

We implement several patterns for the heterogeneous on-package integration of Shi-diannao and NVDLA chiplet accelerators illustrated in Figure 4.5, we test heterogeneous checkerboard, sides, and cross patterns.

Optimization Targets. We perform our search space exploration experiments to target optimizing a single metric at a time, coining the terms Latency Search, Energy Search, and EDP Search. EDP Search is our default experiment.

Search Algorithms and Evaluation. We adopt a brute-force search for all experiments

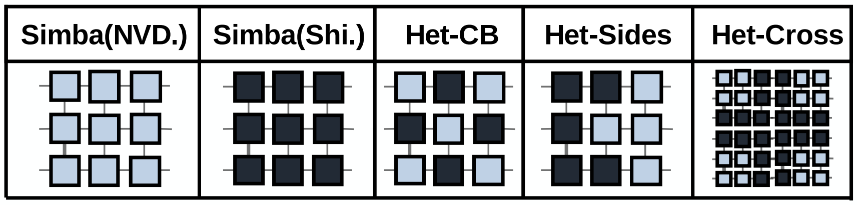


Figure 4.5: The evaluated MCM chiplet organizations.

entailing the 3×3 MCM template. For the scaled 6×6 experiment, we implement an evolutionary algorithm for the SEG module as a meta-heuristic approach to navigate the rising complexity. We set the population size and max number of generations to 10 and 4, respectively. The evaluation criteria follows the scoring function based on hierarchical latency and energy models derived in Section 4.3.

4.5.2 Search Space Exploration Analysis

We compare the 3×3 MCM brute-force search experiments for the heterogeneous and baseline configurations across the different optimization targets for the datacenter and AR/VR usage scenarios. All evaluations are normalized by the standalone NVDLA baseline. We illustrate our results in Figures 4.6 and 4.7 for Scenarios 3 and 4 from the datacenter use cases; the AR Gaming, Outdoors, and VR gaming scenarios from the AR/VR use-cases.

Pipelining Speedups. We observe that the combination of pipelining and chiplet heterogeneity lead to Pareto-optimal operating points that offer performance improvement opportunities. In Figure 4.6, we see that pipelining individually can offer speedups over standalone baselines. For example, Scenario 3 (Latency Search) – top-left most sub-Figure – shows Simba (NVDLA) realizing configurations achieving up to $4\times$ speedup over the standalone NVDLA. This results from potential multiple chiplet assignments per window to each model, speeding up compute-intensive layers of GPT-L,BERT-L, and the 32-batch ResNet-50.

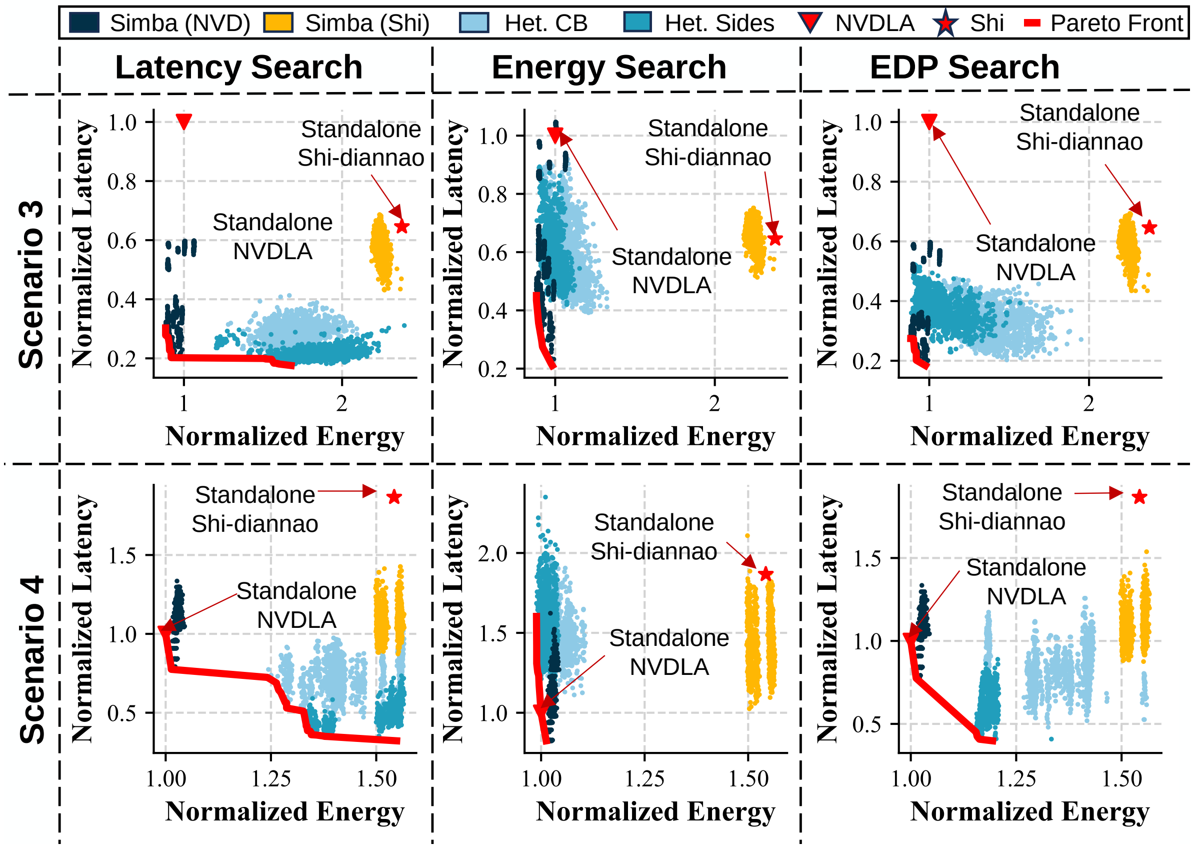


Figure 4.6: Comparing brute force scheduling space exploration processes across various MCM configurations for various search targets for scenarios 4 and 5 from Table 4.2. *standalone Shi-diannao* and *standalone NVDLA* indicate cases where execution is performed on a single chiplet supporting said dataflow.

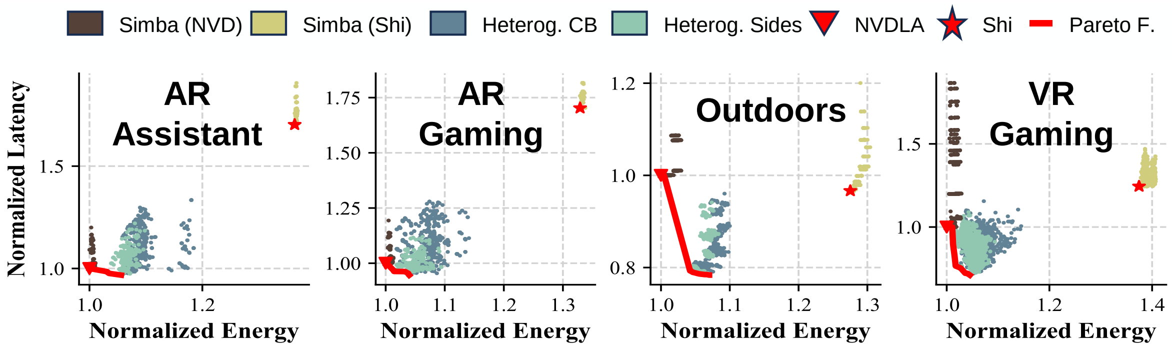


Figure 4.7: Pareto optimal results on the EDP search experiments for the labeled XRBench usage scenarios. Results normalized by standalone NVDLA

Heterogeneous Integration Synergy. As the density of multi-model workloads increase (Scenario 4 from Figure 4.6), we observe the effectiveness of homogeneous pipelining drops due to the increased competition for chiplet resources from heavier workloads. Heterogeneous MCM solutions become viable in such cases as they add another dimension for boosting performance through heterogeneous pipelining schedules, which compensate for the rising complexity through considering the varying affinities of diverse model layers, improving both latency and energy efficiency as seen in their respective search experiments. This benefits of heterogeneous pipelining also hold for the AR/VR scenarios as seen by up to 1.25× execution speedups in Figure 4.7.

Model Suite Diversity. The degree of diversity within the multi-model use-case influences the overall performance improvement. In Scenario 3 (Figure 4.6 top), GPT-L and BERT-L were dominant transformer-based workloads with strong affinities towards NVDLA style, contributing to the homogeneous Simba (NVDLA) solutions dominating the Pareto frontier. In Scenario 4 (Figure 4.6 bot) and the AR/VR (Figure 4.7), the more workload diversity leads more heterogeneous pipelining solutions to dominate the frontier.

Target Optimization. As illustrated in Figure 4.6, the dominance of MCM configurations is also dependent on the target optimization metric. For instance in Figure 4.6 Scenario 4, the Standalone NVDLA baseline is the most energy-efficient solution as it does not incur NoP data movement costs from pipelining. However for Energy Search experiment, Het-Sides configuration identifies scheduling solutions that become the most energy efficient ones by leveraging heterogeneity to overcome extra NoP costs.

4.5.3 Top Performing Schedules Comparison

Next, we compare the top-performing scheduling configurations for each MCM configuration.

NoP and Inter-chiplet Pipelining. We take barplot A1 in Figure 4.8 as an example

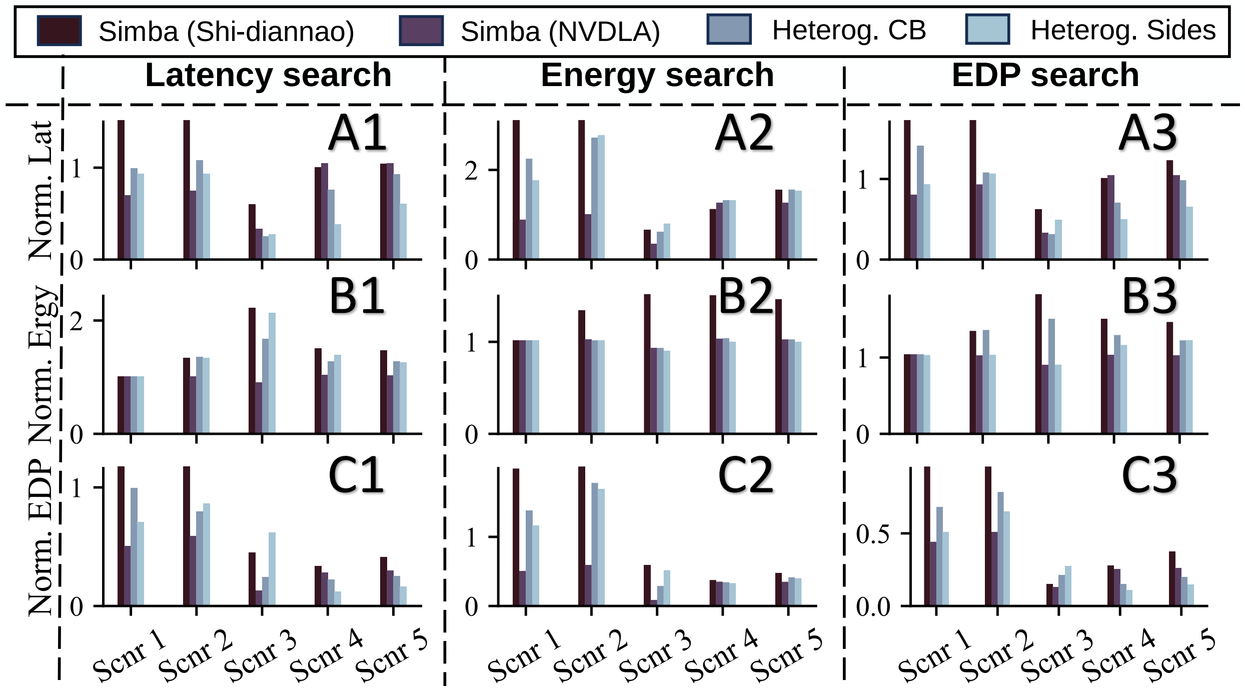


Figure 4.8: Comparing latency, energy, and EDP evaluations for the top-scoring candidates from every search experiment with different optimization criteria across every scenario in Table 4.2. Values normalized by NVDLA standalone.

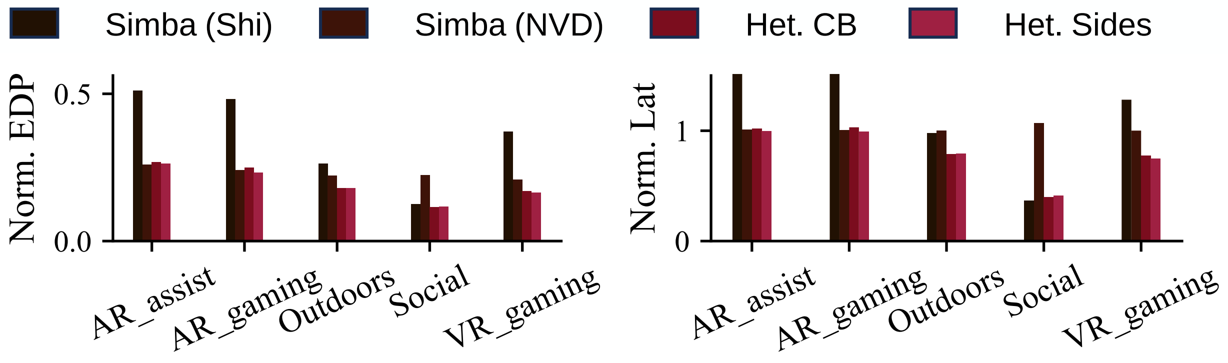


Figure 4.9: EDP and Latency comparison for the AR/VR usage scenario on various MCM templates for the EDP search experiment.

and and zoom in on the Simba (NVDLA) configuration (2nd category) in each scenario. In Scenarios 1-3, the small number of models and limited diversity lead the benefits of inter-chiplet pipelining to outweigh the added NoP costs, enhancing throughput and achieving latency speedups over the standalone NVDLA baseline reaching 1.4×, 1.4×, 3.3×, respectively. Scenarios 4 and 5 sustain larger NoP overheads due to the increased traffic contention from more workloads, causing Simba (NVDLA) 1.05× slowdown compared to the baseline.

Heterogeneity Pattern Choices. Across the heterogeneous MCM scheduling options in Figure 4.8 (Het-CB and Het-Sides), we notice that in the majority of cases, Het-Sides outperforms Het-CB. The reason being is that Het-Sides presents workloads with inter-chiplet pipelining options that can either be homogeneous or heterogeneous based on the chiplets heterogeneous arrangement . This is especially beneficial in cases where are sequences of layers that can benefit from pipelining while sharing the same dataflow affinities, unlike Het-CB which can only offer the heterogeneous pipelining option.

Scenario and Optimization Target. In all matching criteria plots (A1, B2, and C3), Het-Sides configuration at the most compute-intensive and diverse scenarios 4 and 5 consistently outperforms all baselines. For example, Scenario 4 EDP in barplot C3 is reduced by factors of 2.3× and 2.6× compared to Simba (NVDLA) and Simba (Shi-diannao), respectively, while being 9.25× less than the standalone NVDLA. We also show in Figure 4.9 the matching EDP barplot for the AR/VR experiments. We observe that Het-Sides option remains the most efficient option compared to the Simba baselines, achieving on average 5.2% improvement over the standalone NVDLA.

Het-Sides Top Scheduling Strategy. In Figure 4.10, we illustrate the overall scheduling strategy for the top-scoring Het-Sides solution from the EDP search in Scenario 4. The Figure depicts the per-window inner schedules and the progression of accumulative latency for processing the workloads packed into each window. The distinguishing feat from this top scheduling strategy is the non-uniformity of time windows resulting from the greedy-packing

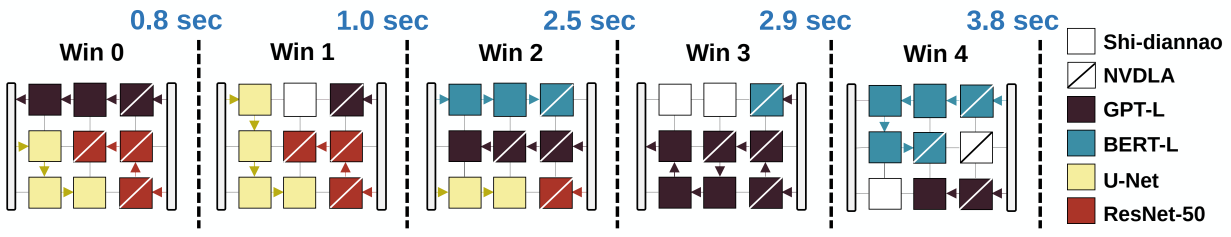


Figure 4.10: The top-scoring partitioning strategy for the Het-Sides scenario 4 experiments. Each window showcases the top-performing schedule within, and the mapping of models onto chiplets. Time boundaries between successive windows are computed over 500 MHz.

Table 4.3: End-to-end latency breakdown in seconds for the top partitioning candidate in Figure 4.10. 'Ideal' indicates individual model latencies unconditioned on window times.

	W0	**W1**	**W2**	**W3**	**W4**	**ideal**	**tot**	**#layers**
GPT-L	0.23	0.21	1.02	0.28	0.23	1.97	3.1	120
BERT-L	0	0	1.47	0.4	0.90	2.77	3.76	60
U-Net	0.21	0.14	0.46	0	0	0.8	1.45	23
ResNet	0.78	0.17	0.11	0	0	1.1	1.1	66
Window	0.78	0.21	1.47	0.4	0.9	-	3.8	269
#layers	60	30	131	25	23	-	-	

heuristic, where smaller workloads (ResNet-50) are assigned to the earlier windows at the expense of larger workloads (e.g., from BERT-L) being delayed to subsequent windows. This facilitates (i) optimizing the schedules of smaller workloads at a finer level of granularity; (ii) avoiding starvation of smaller workloads. Starting from window 2, GPT-L and BERT workloads dominate the schedule, having their segmentation and mapping strategies optimized to minimize the experienced EDP in each window. In Table 4.3, we breakdown how the latency of each window is estimated alongside their assigned number of layers.

4.5.4 Ablation on Windowing and Chiplets Scaling

Ablation Study on Time Partitioning. Using Scenario 4 and Het-Sides EDP Search experiments, we study how performance changes when varying n_{splits} and repeating the experiment in Figure 4.11. We observe the rate of EDP improvements starts to plateau after

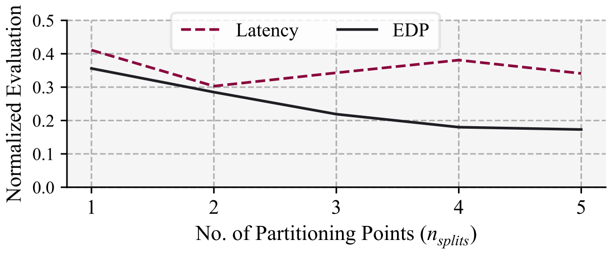


Figure 4.11: Sweep of n_{splits} for the 3×3 Het-Sides EDP search showing top candidates.

n_{splits}=4, where EDP of the top candidate is reduced from n_{splits}=4 to n_{splits}=5 by 1.04×.

Ablation on Greedy Packing Algorithm. Using Scenario 4 and Het-Sides configuration, We tested the efficacy of our first-fit greedy layer packing algorithm against a uniform packing baseline, which uniformly distributes layers from each model across time windows in a uniform fashion. The Greedy layer packing algorithm was superior, improving execution speedups and energy efficiency by 21.8% and 8.6%, respectively.

Scalability. We assess the scalability of our search framework using the full 6×6 Simba MCM system, where we implement an evolutionary algorithm for the *SEG* considering the rising problem complexity from the inclusion of more chiplets. We perform the search for our default experimental settings and Scenario 4 given $n_{splits} \in \{2, 3\}$. We employ a Heterog-Cross pattern and compare it against the Simba baselines in Figure 4.12. We find that the Heterog-Cross top-scoring schedule outperforms those from Simba baselines in all cases across all metrics. At n_{splits}=3, Heterog-Cross leads to 2.3× and 1.9 × reduction in EDP; 2.1× and 1.8× reduction in latency; over Simba (Shi-diannao) and Simba (NVDLA), respectively.

4.6 Related Works

Scheduler for Accelerators. Table 4.4 compares our work against prior scheduling works. As shown, the related works can be categorized into two groups: one which has considered

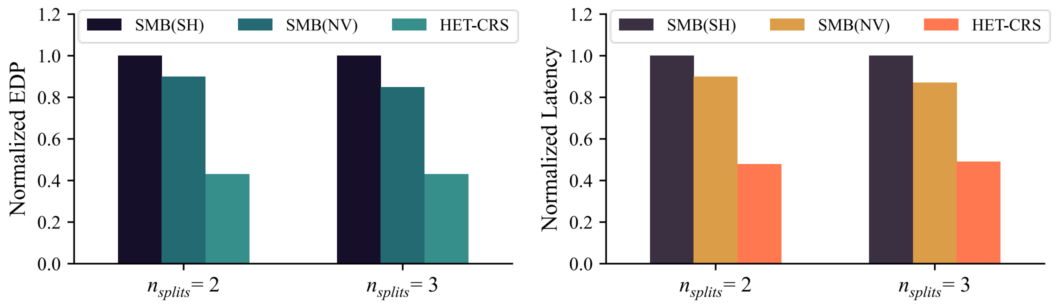


Figure 4.12: Comparing EDP and Latency at $n_{splits} \in \{2, 3\}$ for the 6×6 MCM for target EDP optimizaiton and an evolutionary search algorithm.

aspects of inter-layer pipelining and chiplet-based systems [27,28,58,167,183], while the other category of works focused on multi-model workloads on heterogeneous platforms [63, 92, 94, 100, 114]. Only this work addressed MCM, multi-model workloads, inter-layer pipelining, and heterogeneous dataflow.

Multi-chiplet Modules. Several works have proposed to address the performance scalability challenge for high-performance computing and DNN acceleration via MCM integration [12, 80, 143, 167, 183]. Simba [167] is one notable workload which pioneered a scalable deep learning inference accelerator employing MCM integration leveraging non-uniform work partitioning, communication-aware data placement, and cross-layer pipelining.

Intra- and Inter-layer Parallelism. Numerous works have explored intra-layer parallelism to maximize performance efficiency and resource utilization by partitioning DNN layers into smaller parallelizable tiles [74, 75, 79, 116, 147, 203, 206]. Other works have studied the inter-layer scheduling space to compensate for workloads characterized by low degrees of parallelism [27, 58, 91, 118, 229].

Table 4.4: Comparison against prior related scheduling works.

Work	**Chiplet-based Systems**	**Multi-Models**	**Inter-Layer Pipelining**	**Heterog-Aware**
Simba [167]	✓		✓	
Tangram [58]			✓	
NN-baton [183]	✓			
SET [27]			✓	
Gemini [28]	✓		✓	
Herald [100]		✓		✓
MAGMA [92]		✓		✓
Planaria [63]		✓		✓
Veltair [114]		✓		
MoCA [94]		✓		
This Work	✓	✓	✓	✓

4.7 Summary

In this work, we explored the scheduling space of a new class of MCM accelerator architecture, heterogeneous MCM AI accelerator, targeting multi-model AI workloads. We identify that the scheduling problem is intractably large but multi-level problem formulation and heuristics we proposed are effective for the extended scheduling problem. The results also show that heterogeneous MCM accelerator is beneficial for multi-model workloads, which motivates further exploration.

Chapter 5

Efficient Split Computing for Edge-AI Autonomous Systems applications

We shift gears and study the emerging split-computing paradigm and how it can be effectively implemented for Edge-AI applications like autonomous systems. The theory behind split computing is to partition a target DNN across two platforms: (i) the user end device with limited computational resources and compute capabilities. (ii) a compute-capable server with abundant resources. We propose new techniques to maximize the effectiveness of split computing at design and run times with an emphasis on the autonomous driving use case.

5.1 Introduction

Advances in the theory and application of Neural Networks (NNs), particularly Deep NNs (DNNs), have spurred revolutionary progress on a number of AI tasks, including perception, motion planning and control. As as result, DNNs have provided a feasible engineering solution to supplant formerly human-only tasks, most ambitiously in autonomous systems.

However, state-of-the-art platforms, like autonomous driving systems (ADS), require the use of very large DNN architectures to solve essential perception and control tasks, which generally involve processing the output of tens of cameras, LiDARs and other sensors.

As a result, contemporary ADSs are only possible with significant computational resources deployed on the vehicle itself, since their DNNs must process such high-bandwidth sensors in closed loop, in real time. The practical energy impact of high-capacity on-vehicle compute is understudied, but current research suggests that it is profound: e.g., up to a 15% reduction in a vehicle's range [109, 130, 189, 191, 192].

At the same time, advances in semiconductor design and packaging have made possible cheap, low-power silicon; and advances in wireless networking have made high-bandwidth, low-latency radio links possible even in challenging multi-user environments. Together, these advances have led to increasingly ubiquitous, cheap, wirelessly-accessible computational resources near the *edge* of conventional hard-wired infrastructure. In particular, it is now possible to achieve reliable, millisecond-latency wireless connections between connected ADSs and nearby edge computing [14, 112, 119].

The ubiquity of edge compute thus suggests a natural way to reduce the *local* energy consumption on ADS vehicles (and the broader autonomous systems): viz., by wirelessly *offloading* onerous perception and control DNN computations to abundant nearby edge compute infrastructure. Figure 5.1 illustrates how this edge computing paradigm can be adopted in the field of vehicular computing via advanced wireless technologies, where edge devices represent connected vehicles (CVs) while vehicular edge servers (VECs) are stationed at Road Side Units (RSUs) [14].

More Recently, *split-computing* has been introduced as a nuanced form of edge computing where rather than offloading the input sensory data to the edge, DNN computations can be split between the edge device (ADS) and server to balance computation-communication

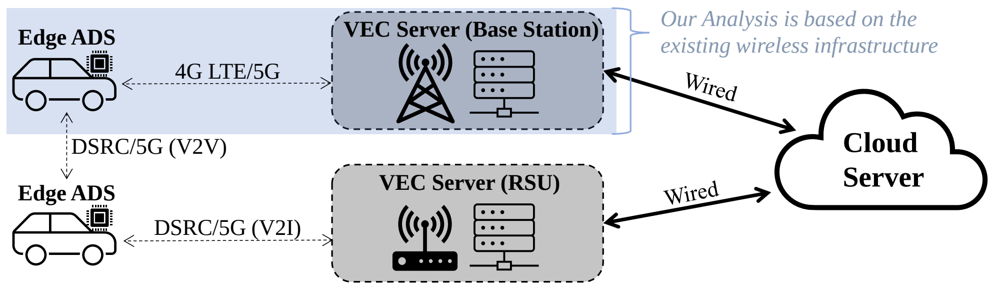


Figure 5.1: A Practical Example of the Edge Computing Paradigm for Vehicular Applications

overheads. Particularly, the concept emanates from the observation that data sizes can shrink as DNN computation proceeds, yielding split layers with lower transmission overheads [88, 142]. Even more so, recent research works have proposed the artificial construction of a *bottleneck* layer early within the model architecture, enforcing feature data compression prior to wireless transmission without degrading the model's utility [49, 119, 123].

5.1.1 Current Limitations of Split Computing

Despite the promise of split computing, the volatile nature of wireless connection (path propagation loss, network congestion) and the lack of a systematic approach to account for it remain a serious challenge for latency-critical applications like autonomous systems. In particular, we identified the following limitations.

- The lack of a systematic design approach for split-computing models for autonomous systems, where the design problem can be seen as that of a multi-branch neural networks distributed across two platforms while possessing stringent latency constraints.
- Scaling considerations of offloading optimizations for onerous edge systems like ADS given bandwidth restrictions and shared execution deadlines (e.g., processing 8 camera inputs from Tesla Autopilot [5]).
- The need for a learning-based adaptive controller for smooth adjustment of edge com-

puting modes and offloading granularity based on experienced runtime variations (wireless conditions, scene complexity) while meeting latency and robustness constraints.

5.1.2 Novel Contributions

In light of the aforementioned challenges, we make the following contributions.

1. We present a novel framework to design multi-branch split-computing model architectures for autonomous systems using knowledge distillation and NAS, scalable to both single and multi-sensor DNNs.
2. We propose a deep reinforcement learning (DRL) solution that leverages spatio-temporal correlations in the sensory input data streams to adjust offloading granularity at runtime according to the seen context and wireless conditions.
3. We perform a myriad of experiments using various autonomous driving datasets and evaluations on the industry standard ADS platform (Nvidia Drive PX2) which demonstrate the merits of our methods.

5.2 Related Works

DNN Split Computing: To identify optimal offloading points within DNN architectures, [88, 142] analyzed the expected computation and communication costs for each potential offloading layer. For a considerable number of architectures, either direct raw inputs offloading or pure local execution represented the most efficient option. Therefore, works in [49, 122, 123] proposed to modify a DNN's structure to include an early optimal offloading layer that shrinks the size of transmissible data, minimizing the costs of both computation and communication. This split-computing concept was applied for end-to-end control in autonomous vehicles [119].

Vehicular Edge Computing (VEC): Numerous research efforts have targeted system-wide resource optimization for VEC through optimal task offloading and scheduling strategies given a variety of servers, vehicles, and tasks [202, 226]. Typically, such strategies are complemented with runtime solutions that can tune the operation according to variations in the deployment environment, such as the network connectivity conditions [39]. Nonetheless, delayed responses from edge servers are not tolerated in autonomous driving application as the safety of the road, vehicles, and passengers [14] can be compromised. Hence, [195] proposed a customized communication protocol for a stable and fast offloading of autonomous driving tasks. Even more so, the authors in [119] proposed a fail-safe routine to re-invoke local computation if responses are delayed beyond a certain threshold to account for the uncertainty of wireless links.

Mutli-Sensor Perception: To maximize information extraction from a driving scene, data is collected from a diverse set of sensors, e.g., cameras, lidar, and radar, to promote perception robustness. Mainly, There are two primary schemes for processing these multi-sensory inputs: early fusion [164, 217] and late fusion [120, 208]. The former combines all sensory features to a single feature at an early point in the ADS pipeline, but is susceptible to sensing noise. Conversely, the latter offers more resilience at the expense of more redundancy across the sensor pipelines.

5.3 Design of Split Computing Models

We study the structure of bottlenecks, how to train them using knowledge distillation, and how to leverage NAS to realize optimized multi-branch architectures for edge computing.

5.3.1 Bottleneck Design

Bottleneck Structure. One prominent approach for achieving efficient split-computation between the local platform and the edge server is the application of in-model compression to obtain optimal offloading points. Formally, a DNN model $\mathcal{M}$ can be split into two parts: a head $\mathcal{M}_H$ and tail $\mathcal{M}_T$ to be deployed on the local and edge server platforms, respectively. The direct approach to select the splitting layer, ℓ, has been to identify the layer at which the output $z_\ell = \mathcal{M}_H(x)$ becomes smaller in size than the input x to decrease transmission overhead. Oftentimes, this criterion is only met at the latter layers for many DNN architectures, which leads to increased local computation [88,142]. Instead, recent split computing works proposed the notion of in-model compression through a *bottleneck* [49], in which a modified model version $\mathcal{M}'$ would comprise 3 sections: $\mathcal{M}_E$, $\mathcal{M}_D$, and $\mathcal{M}_T$. Submodels $\mathcal{M}_E$ and $\mathcal{M}_D$ represent a specialized form of an encoder-decoder architecture replacing the original $\mathcal{M}_H$. From here, $\mathcal{M}_E$ would serve as the new head $\mathcal{M}'_H$ while the concatenation of $\mathcal{M}_D$ and $\mathcal{M}_T$ would be deployed on the edge server. Conceptually, $\mathcal{M}_E$ is introduced to obtain the compressed form $z'_\ell = \mathcal{M}_E(x)$ prematurely in the network to realize an early optimal offloading point (the bottleneck) within $\mathcal{M}'$. $\mathcal{M}_D$ on the other hand serves two purposes: (i) ensuring that $z' = \mathcal{M}_D(\mathcal{M}_E(x))$ maintains the same spatial dimensions as the original input to $\mathcal{M}_T$, and (ii) minimizing the loss incurred by $\mathcal{M}'$ due to the proposed structural modifications of $\mathcal{M}_E$ and $\mathcal{M}_D$.

Bottleneck Training. Rather than retraining the entire modified model from scratch, we apply knowledge distillation [123] to train $\mathcal{M}_E$ and $\mathcal{M}_D$ through minimizing a loss function, e.g., mean squared error (L_{MSE}), between $\mathcal{M}_D$ outputs and those from the original parts, $\mathcal{M}_{orig}$. Figure 5.2 illustrates this for our ResNet-18 with the loss component for a single input x given by:

$$L_{SE} = ||\mathcal{M}_{orig}(x) - \mathcal{M}_D(\mathcal{M}_E(x))||_2^2 \tag{5.1}$$

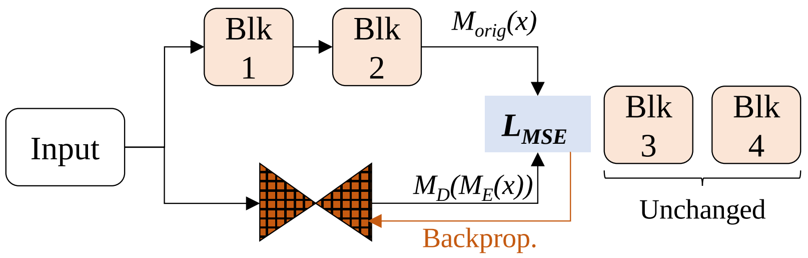


Figure 5.2: Our ResNet-18 feature extractors undergo Knowledge Distillation to train $\mathcal{M}_E \cdot \mathcal{M}_D$ using the first 2 blocks

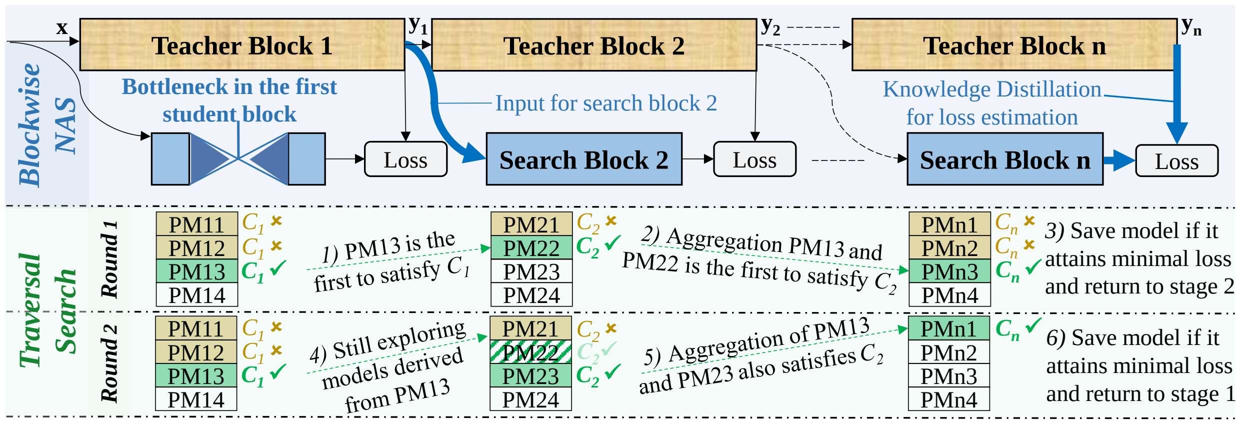
Thus, unaltered DNN components can retain their weight values with only the parameters of the new structure being trained to produce the same output values as the originals.

5.3.2 NAS for Distributed Multi-Branch Model Architecture

As illustrated in Figure 5.3, we propose to adopt a blockwise neural architecture search (NAS) approach for designing multi-branch neural network for edge computing. Briefly, blockwise NAS relies on a *teacher* model pretrained on the target task and constituting a sequence of computing blocks (e.g., the four blocks of a ResNet architecture), and a *student* model comprising the same number of search blocks, each reflecting a subspace of the overall NAS search space. As such, the NAS problem is reduced to identifying the optimal combination of student blocks where each can be searched and trained separately.

The advantages of this approach are threefold: (*i*) A modular approach aids in identifying which blocks are the most sensitive to alterations, allowing optimizations to be targeted towards the less-sensitive blocks, (*ii*) Customization of search blocks is supported, enabling the inclusion of a bottleneck design space in one of the earlier search blocks (see Figure 5.3), (*iii*) The modularity enables rendering multiple computing paths of varying degrees of complexity for diverse runtime scenarios.

Blockwise NAS using Knowledge Distillation. To manage the colossal search over-



Blockwise NAS
Teacher Block 1
Teacher Block 2
Teacher Block n
Bottleneck in the first student block
Input for search block 2
Knowledge Distillation for loss estimation
Search Block 2
Search Block n
Loss
Traversal Search
Round 1
Round 2
1) PM13 is the first to satisfy C1
2) Aggregation PM13 and PM22 is the first to satisfy C2
3) Save model if it attains minimal loss and return to stage 2
4) Still exploring models derived from PM13
5) Aggregation of PM13 and PM23 also satisfies C2
6) Save model if it attains minimal loss and return to stage 1

Figure 5.3: Blockwise NAS for edge computing (*top*) and a walk-through example for the traversal search (*bottom*). PM is for partial model and its indices are for the stage and the PM's ranking based on the loss defined in (5.5).

heads of NAS, the approach in [105] proposed to divide the NAS search space $\mathcal{A}$ into smaller successive independent supernets $\mathcal{A}_i$ with each block i possessing its shared weights W_i, leading to an exponential reduction in the search space size and the overall design turnaround time. Thus, given inputs X and ground truth values Y, an optimized subnet architecture, α^*, is formed by aggregating N subnets from the search blocks which satisfy:

$$\alpha^* = \arg\min_{\alpha \in \mathcal{A}} \sum_{i=1}^{N} \mathcal{L}_{val}(W_i^*(\alpha_i), \alpha_i; y_{i-1}, y) \tag{5.2}$$

$$s.t.\ W_i^* = \min_{W_i} \ \mathcal{L}_{train}(W_i, \mathcal{A}_i; y_{i-1}, y_i) \tag{5.3}$$

where y_{i-1} and y_i represent the inputs and ground truth labels for search block i, respectively. Practically, pre-trained DNN models on the same task can be leveraged as *teachers* to obtain y_i and y_{i-1} from their intermediate data representations at different stages, which allows guiding the search process for each search block i. In words, the main building blocks constituting a DNN architecture, such as the 4 primary blocks of stacked layers in a ResNet architecture [72], are designated as separate teacher blocks, each with its input and output representations utilized as guides for the corresponding search block. Using knowledge distillation (KD), the training and validation loss estimates, $\mathcal{L}_{train}$ and $\mathcal{L}_{val}$, between block predictions $\hat{y}_i(\cdot)$ and the teacher ground truth values can be given by:

$$\mathcal{L}_{train}(W_i, \mathcal{A}_i; y_{i-1}, y_i) = \frac{1}{K}||y_i - \hat{y}_i(y_{i-1})||^j \tag{5.4}$$

$$\mathcal{L}_{val}(W_i, \mathcal{A}_i; y_{i-1}, y_i) = \frac{1}{K \cdot \sigma^j(y_i)}||y_i - \hat{y}_i(y_{i-1})||^j \tag{5.5}$$

in which K is the number of output neurons, $\sigma(y_i)$ is the standard deviation of y_i, and j is for the function degree. The loss estimate in $\mathcal{L}_{val}$ is normalized relative to the corresponding $\sigma^j(y_i)$ to ensure fairness since feature map sizes can differ from one candidate partial model to the other within a search block. We set respective j values to two and one in our experiments.

Model Aggregation under Constraint: After the initial search process has concluded, partial model rankings are rendered for each search block according to $\mathcal{L}_{val}$. If there are no target performance constraints, then the top-ranking partial models from each block can be concatenated to construct the complete DNN model A target performance constraint (e.g., latency) denoted by C_{target} can be specified to guide the aggregation. To avoid the prohibitive act of evaluating each possible combination of partial models, we construct a lookup table for the performance costs of each candidate operation within a search block (obtained through hardware measurements). Then, we can estimate the maximum allowable cumulative performance cost for each block C_i as:

$$C_i = \sum_{n=1}^{i} cost_n = C_{target} - \sum_{n=i+1}^{N} min\_cost_n \tag{5.6}$$

where min_cost_n is the minimum cost for a partial model at block n estimated from the pre-calculated lookup table. Once each block's maximum cost C_i has been estimated using (5.6), a traversal search can be performed starting from the first search block, and *recursively* going through the partial models of the subsequent blocks as long as the corresponding C_i constraints are satisfied. In other words, the testing of subsequent blocks is skipped if the current partially constructed model at block i has a cumulative performance cost that

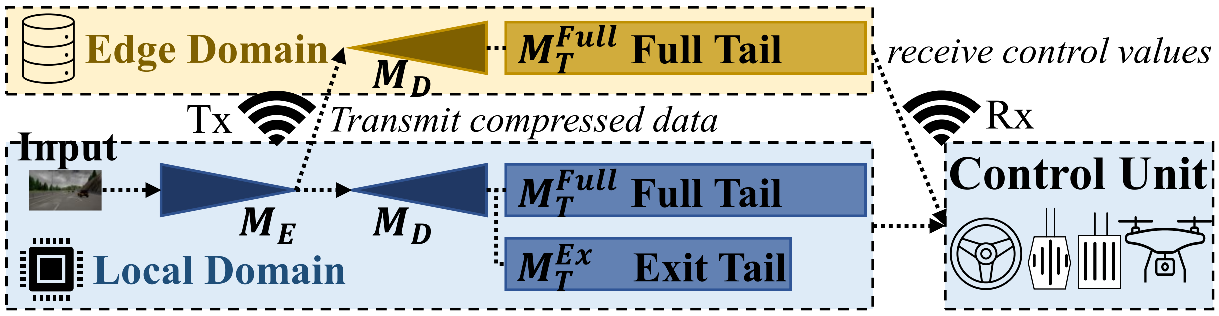


Figure 5.4: An example end-to-end distributed multi-branch DNN architecture for reliable and efficient edge computing in autonomous systems.

exceeds C_i. Furthermore, once a model satisfying the constraint has been identified, the search returns to the previous block to avoid testing inferior models [105]. A walk-through example for this traversal search is provided in Figure 5.3 (bottom).

Multi-branch Neural Network Deployment. Figure 5.4 illustrates an example on how to deploy the multi-branch DNN distributed across the execution domains of the local device and edge server. As shown, $\mathcal{M}_E$ with the optimal offloading point from the first student block is placed locally to be shared by all possible execution paths. Conversely, $\mathcal{M}_D$ is replicated across both execution domains. The remainder 'Tail' part of the model can be constructed using modules from the teacher, student, or a hybrid of them. In the Figure, we show subsequent computing blocks computed from the original *teacher* model concatenated at the end of $\mathcal{M}_D$ forming the *full* tail model, $\mathcal{M}_T^{Full}$. Furthermore, the local domain can posses an extra local early exit tail $\mathcal{M}_T^{Ex}$ following $\mathcal{M}_D$ constructed from the remaining *student* blocks. All execution paths converge on the local device to to supply downstream predictions for the control unit.

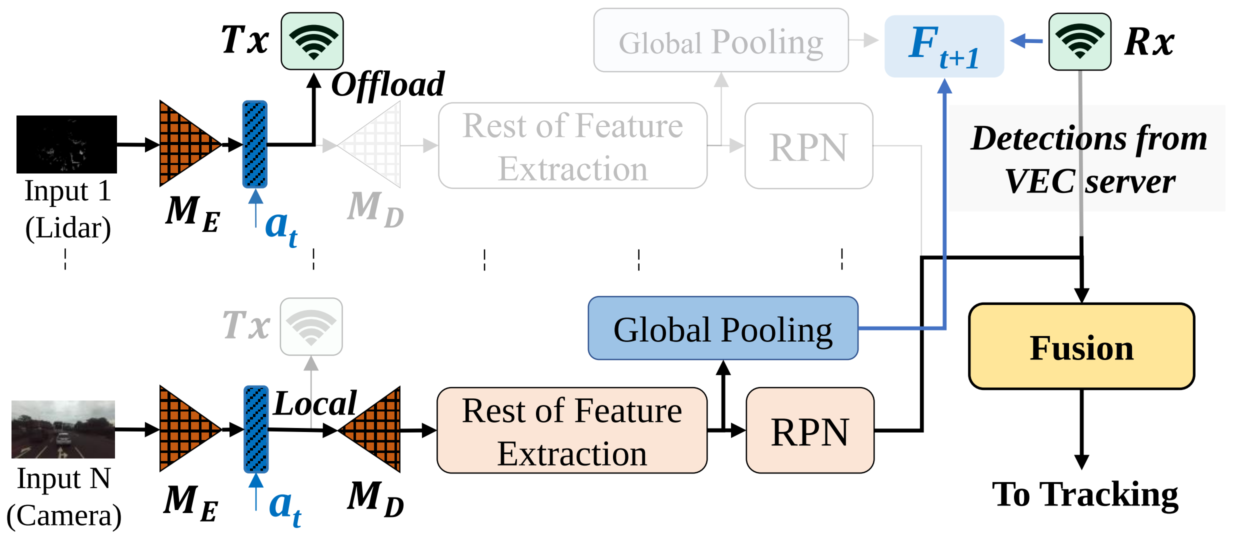


Figure 5.5: Illustration of multi-sensor object detection with late fusion and offloading support. Blue blocks/variables are passed to/from the DRL. Transparent blocks are inactive.

5.4 Context Aware Runtime Controller Design

In order to adjust the edge computing mode at runtime, we propose a deep reinforcement learning (DRL) solution that extracts contextual knowledge and discerns temporal patterns within collected data streams (e.g., mounted camera feed) as well as the wireless network conditions. The novelty of our DRL implementation is rooted in the following principles: (i) AS applications process data samples collected at high sampling frequencies, exhibiting strong spatio-temporal correlations between *successive* samples. (ii) Using low-dimensional abstract representations for DRL processing so as to realize minimal processing overheads. Figure 5.5 provides an illustrative concept of how abstract data can be pooled from concurrent object detectors in a multi-sensor platform before being used as inputs to the DRL.

5.4.1 Agent Design

As shown in Figure 5.6, our DRL solution constitutes a hierarchical agent. Given N processing pipelines from a multi-sensor platform, we can specify the DRL components as follows.

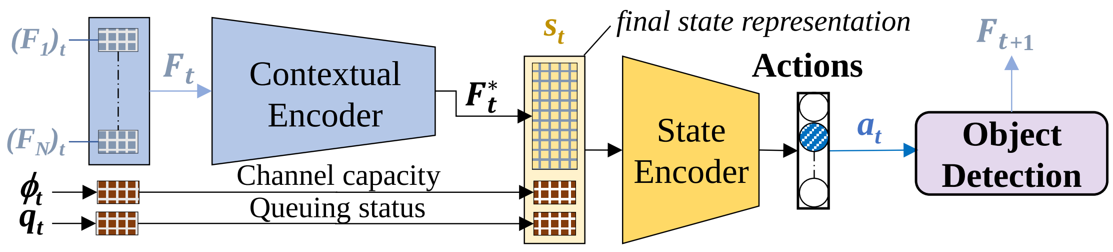


Figure 5.6: Our hierarchical agent for runtime mode selection

Contextual Encoder. In order to estimate the complexity of the corresponding scene, we leverage the computed feature set, $\mathcal{F}_t$, at time window t from the main sensor processing pipelines to guide the decision for the following window $t+1$, given as $\mathcal{F}_t = \{(\mathcal{F}_1)_t, (\mathcal{F}_2)_t, .., (\mathcal{F}_N)_t\}$. The rationale behind using the feature set of the preceding time window is twofold: (*i*) features do not need to be computed from scratch as they have already been generated within the primary processing pipelines (see the global pooling blocks in Figure 5.5), and, (*ii*) the small window size for autonomous driving (≤ 100 ms) means that successive frames share contextual information with high spatio-temporal correlations.

Still, the influence of $\mathcal{F}_t$ can outweigh other DRL inputs due to its relatively larger size. Thus, we propose to encode $\mathcal{F}_t$ into a lower-dimensional representation, where $\mathcal{F}_t$ is processed through a *contextual encoder* comprising a sequence of fully-connected layers to obtain the final abstraction $\mathcal{F}_t^*$. In our design, $\mathcal{F}_t^*$ was of $256\times$ smaller in size than $\mathcal{F}_t$.

State Encoder. The next component is the state encoder whose input is the final state representation $s_t = \{\mathcal{F}_t^*, \phi_t, q_t\}$ formed from aggregating the contextual encoder outputs, $\mathcal{F}_t^*$, the channel capacity ϕ_t, and server queuing delays q_t. In practice, the latter two metrics can be estimated by probing the edge server and/or through a function of the Received Signal Strength Index (RSSI) at the edge device. However, we defined them as such to enable their probabilistic modeling for training the DRL agent.

Action Space. Represented by the final fully-connected layer in the *state encoder*, the action space covers the set of all possible modes of operation that can be selected by the DRL at runtime. We define it as $A = \{offload_0, offload_1, offload_2, ...offload_{N-1}\}$, where an action $offload_i$ is for choosing the offloading option for i sensory pipelines, with $offload_0$ being pure local execution. In the case that the same DNN structure is shared across all pipelines, only the number of offloading pipelines matter. We do not consider $offload_N$ as a viable option so that the ADS is always guaranteed a new output every time window by having at least one pipeline always processed locally. This way, even under a worst-case scenario when tasks from $N-1$ pipelines are offloaded and results are not received within the time limit, the vehicle can still operate in a safe manner. In practice, we merely need a subset of actions $A^* \subseteq A$, with $\{offload_0, offload_{N-1}\} \subseteq A^*$, where A^* can contain the actions that exhibit notable variability in performance. At runtime, action vector, a_t, is mapped onto the control of each processing pipeline.

5.4.2 DRL Environment

To train the DRL agent, we emulate a training environment for learning a policy π that makes offloading decisions based on the current state.

Training and Reward. Reinforcement learning approaches rely on having a Q function to provide value estimates for each state-action pair so as to select the optimal action $\hat{a} = \arg\max_{a \in A} Q_\pi(\hat{s}, a)$ for each $\hat{s}$ under a learnt policy π. However, estimating state-action pair values in continuous state spaces is challenging, and DRL offers to approximate Q_π by a policy network trained to maximize a reward. With no loss in generality, our DRL employs a Double Deep Q-Network [188] with a compounded reward function $\mathcal{R}$ as:

$$\mathcal{R} = \begin{cases} \mathcal{A}, & \text{if mAP}(y) < \text{mAP}_{th} \\ \mathcal{B}, & otherwise \end{cases} \tag{5.7}$$

which evaluates to different functions based on a measure of robustness. For example in object detection, robustness can be lossely defined as meeting a certain mean Average Precision (mAP) threshold score[1]. In brief, our goal is for the agent to realize a policy that deters from offloading actions when prediction confidence is low. Our method of achieving this is through the contextual information present in $\mathcal{F}_{t-1}$ which assess the scene's complexity for guiding offloading decisions and minimizing prediction uncertainty. Thus, if mAP_{th} is not met, $\mathcal{R}$ evaluates to $\mathcal{A}$ defined as:

$$\mathcal{A} = \begin{cases} 0, & \text{if } \hat{a} == offload_0 \\ \frac{P}{N-i}, & \text{if } \hat{a} == offload_i \end{cases} \quad s.t., i \neq 0, i < N \tag{5.8}$$

where the agent is penalized whenever an offloading action is selected and mAP_{th} is not met. The penalty value is proportional to the number of offloading pipelines, i, out of N total, with a maximum *negative* penalty of P. Recall that $offload_N \notin A$ as one pipeline always executes locally to ensure at least one output is available irrespective of the wireless network conditions. On the flip side, when mAP$_{th}$ is satisfied, $\mathcal{R}$ evaluates to $\mathcal{B}$ as follows:

$$\mathcal{B} = \begin{cases} P, & \text{if } L(X|\hat{a}) > L_{th} \\ \mathcal{C}, & otherwise \end{cases} \tag{5.9}$$

which penalizes the agent by P when its selected action $\hat{a}$ causes the overall execution latency for inputs X, $L(X|\hat{a})$, to exceed the critical execution latency constraint, L_{th}. In other words, this means that the agent is penalized when not all partial outputs are available in time for late fusion. In reality, state-of-the-art ADS platforms are designed to meet the application latency demands, and hence, we set the value of L_{th} to that of *local execution*. Contrarily,

[1]Chapter 6 focuses on how to address robustness for edge computing in a formal, provably safe manner

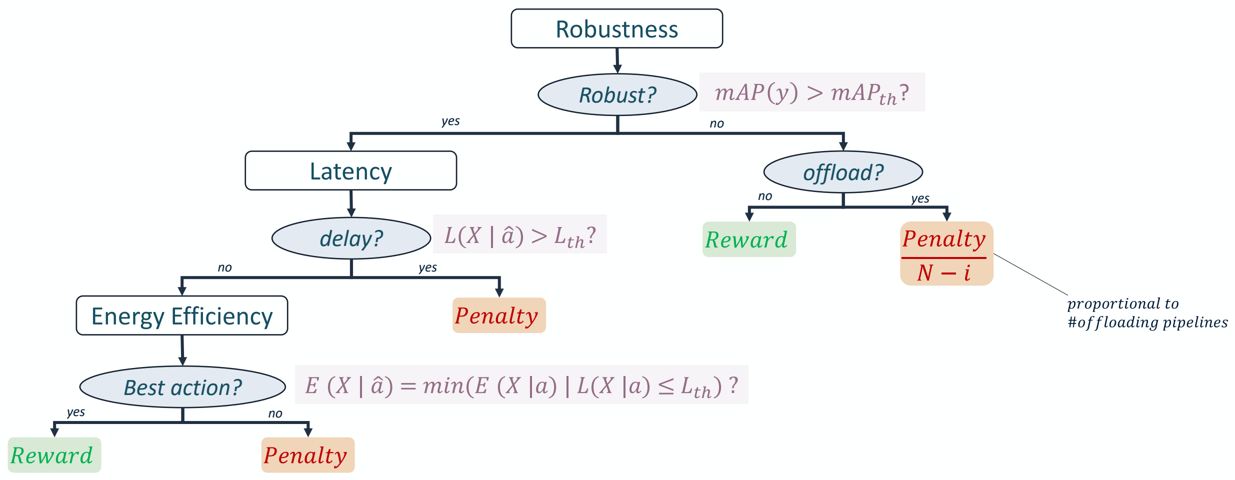


Figure 5.7: Flowchart of the adopted reward function.

when L_{th} is satisfied, $\mathcal{R}$ finally evaluates to $\mathcal{C}$ given by:

$$\mathcal{C} = \begin{cases} 0, & \text{if } E(X|\hat{a}) == min(E(X|a)|L(X|a) \le L_{th}) \\ P, & otherwise \end{cases} \qquad \forall a \in A^*, A^* \subseteq A \tag{5.10}$$

penalizing the agent by P if the energy consumption footprint $E(X|\hat{a})$ from selecting action $\hat{a}$ is not the minimal from amongst those of all viable actions $a \in A^*$ projected to meet L_{th}. A flowchart of the final adopted reward function is shown in Figure 5.7.

Latency and Energy Estimation. In order to evaluate $\mathcal{R}$ for each selected $\hat{a}$, the end-to-end estimates for energy and latency can be approximated every time window as follows:

$$L = L_{local} + L_{Tx} + L_{server} + L_{Rx} \tag{5.11}$$

$$E = E_{local} + E_{Tx} + E_{idle} + E_{Rx} \tag{5.12}$$

where the latency L can be broken down into the respective local, transmission, server, and receiving latencies. Similarly, energy consumption constitutes the same components except for incorporating idling energy as we are only concerned about the ADS energy footprint.

From here, the local components are given by:

$$L_{local} = N \times L_{\mathcal{M}_E} + (N - i) \times L_{tail} \mid \hat{a} == offload_i \tag{5.13}$$

in which $L_{\mathcal{M}_E}$ and L_{tail} are the respective latencies for executing the encoder $\mathcal{M}_E$ and the remaining tail parts of the model, respectively. When the selected action is to offload processing from i processing pipelines (i.e. $\hat{a} == offload_i$), the total local execution latency accounts for processing across the N encoders and the $N - i$ tail models. This additive form represents the most direct approach for modeling local execution. However in reality, the concurrency of pipelines can speed up local execution depending on the available hardware resources at the expense of a larger power consumption footprint, P_{local}. We approximate this trade-off through considering energy for performance evaluation, defining E_{local} as:

$$E_{local} = L_{local} \times P_{local} \tag{5.14}$$

Channel Estimation. To estimate the communication overheads, we first fit a Rayleigh distribution curve with scale σ to throughput traces Φ collected from the real-world for different wireless technologies, i.e., $\Phi \sim \text{Rayleigh}(\sigma)$. Then, we use the constructed distribution to sample independent and identically distributed (i.i.d.) random variables as the channel capacity ϕ to be used for the training and evaluation processes of the DRL agent where data transmission parameters can be evaluated as:

$$L_{Tx} = \frac{i \times b}{\phi} \mid \hat{a} == offload_i; \;\; E_{Tx} = L_{Tx} \times P_{Tx} \tag{5.15}$$

where b is the transmissible data size from one sensory pipeline while P_{Tx} is the transmission power incurred by the ADS. Similarly, the formulation for the receiving parameters, L_{Rx} and E_{Rx}, can be provided given corresponding estimates for channel capacity and data sizes in the downlink.

Server Queuing. Lastly, we model the server latency L_{server} using queuing delays:

$$q_c = \frac{(1-\rho)(\rho)^c}{1-\rho^{C+1}} \tag{5.16}$$

representing the probability that the offloaded task would encounter c other tasks before it in the server's processing queue, with ρ being the average server load, and C being the queue size. From here, we are able to generate a probability density function (pdf) for values within 0-C from which we can sample queuing positions, and consequently approximate L_{server}.

5.5 Evaluation

We provide our experimental evaluation results for the multi-sensor object detection use case for autonomous driving. The full set of results can be found in our research works [32, 138] .

5.5.1 Experimental Setup

Dataset. We use the RADIATE multimodal perception dataset [168] for its diverse driving scenarios and adverse weather conditions such as snow, fog, and rain. The dataset covers 8 object classes with annotations from a Navtech CTS350-X radar, a Velodyne HDL-32e LiDAR, and a ZED stereo camera. We implemented 4 object detection DNN pipelines: 2 stereo cameras, radar, and lidar. All inputs are mapped onto the forward-facing perspective for late fusion.

Training and Metrics The original object detectors for each sensor pipeline comprised a ResNet-18 followed by a Faster R-CNN. These models were trained using a batch size of 1, learning rate of 0.005, and the multi-task loss function in [159] which combines both classification and box regression losses. For the NMS fusion, we use a fusion IoU threshold of 0.4. We employ mAP as our evaluation metric with boxes IoU ≥ 0.5 since it is widely adopted for object detection tasks [50] where the average precision is estimated using the precision and recall values. More details about evaluating these values are in [50, 159].

Hardware and Performance Evaluation We use the industry-grade Nvidia Drive PX2 Autochauffer as our ADS hardware. The concurrent DNN models are compiled using the TensorRT library becoming inference engines. The local execution power P_{local} is estimated as the difference in the ADS power measurements when processing and idling. For the transmission power P_{Tx}, we follow [119] and evaluate it using data transfer power models [77].

Encoder-Decoder Structure. The input frame's resolution for each of the sensory pipelines is 672×376 (≈ 740.25 kB). The encoder, $\mathcal{M}_E$, comprises 3 layers (2 convolutional and 1 pooling), each with a stride of 2 with only 3 channels at the output. Therefore, when the outputs from $\mathcal{M}_E$ are quantized to 8 bits for offloading [119], the transmissible data size b in equation 5.15 becomes ≈ 11.57 kB ($64\times$ less than the input's). The decoder $\mathcal{M}_D$ mimics the structure presented in [123] to have its output of the same dimensions as that from the original second ResNet-18 block.

DRL Settings. For safety, we always execute the radar pipeline locally [113] and define $A^* = \{offload_0, offload_2, offload_3\}$. We set $P = -2$, $C = 4000$, $\rho = 0.9$, and $\text{mAP}_{th} = 0.68$ unless otherwise stated. We set $L_{th} = 68.12$ based on pure local execution latency.

5.5.2 Bottleneck Evaluation in Multi-sensor ADS

We first assess how the inclusion of $\mathcal{M}_{\mathcal{E}}$ and $\mathcal{M}_{\mathcal{D}}$ impacts the loss and prediction accuracy of object detection. Table 5.1 shows the changes in these metrics across different late fusion combinations on the Radiate evaluation dataset. As seen, full sensor fusion has the best performance in mAP, asserting how prediction robustness relates to the number of fused

Table 5.1: Loss and mAP (%) before (orig) and after (dist) integrating $\mathcal{M}_E \cdot \mathcal{M}_D$ across various late fusion combinations.

Sensor	Loss (orig)	Loss (dist)	mAP (orig)	mAP (dist)
2 Cameras	0.15	0.17	67.14	67.14
Radar+Lidar	0.10	0.11	67.14	67.14
Full Fusion	0.13	0.15	71.24	70.38

Table 5.2: Hardware Measurements on the Nvidia Drive PX2

DNN	L_{local} (ms)	E_{local} (J)	Memory (MB)
Encoder	3.78	0.03	0.025
1 pipeline	17.03	0.12	80.3
4 pipelines	68.12	0.48	321.2
DRL Agent	0.66	0.005	5.4

outputs. It is also observed that the new DNN structures maintain the same level of performance as their original counterparts, with the highest degradation in mAP from 71.24% to 70.38% experienced by the full fusion case, but still offering a better score than that of the simpler sensor combinations.

Table 5.2 displays the processing overheads for different DNN components deployed on the PX2 hardware. The encoder $\mathcal{M}_\mathcal{E}$ and DRL agent take 3.78 and 0.66 ms, respectively, emphasizing how the decision a_t is obtained before the generation of any transmissible outputs. Moreover, the execution latency for 4 pipelines on the PX2 can add up to 68.12 ms given the same power P_{local}.

5.5.3 DRL Efficiency and Robustness Evaluation

Channel Capacity and Queuing Analysis. In this experiment, we analyze the influence of the experienced channel capacity, ϕ, and queuing delay, q_t, on the optimal action choice when optimizing for energy consumption under the latency constraint L_{th}. To elaborate, we illustrate in Figure 5.8 parametric sweeps with respect to ϕ given $q_t = 15$ ms. As shown, offloading options are consistently more energy efficient than the pure local option (*offload*$_0$),

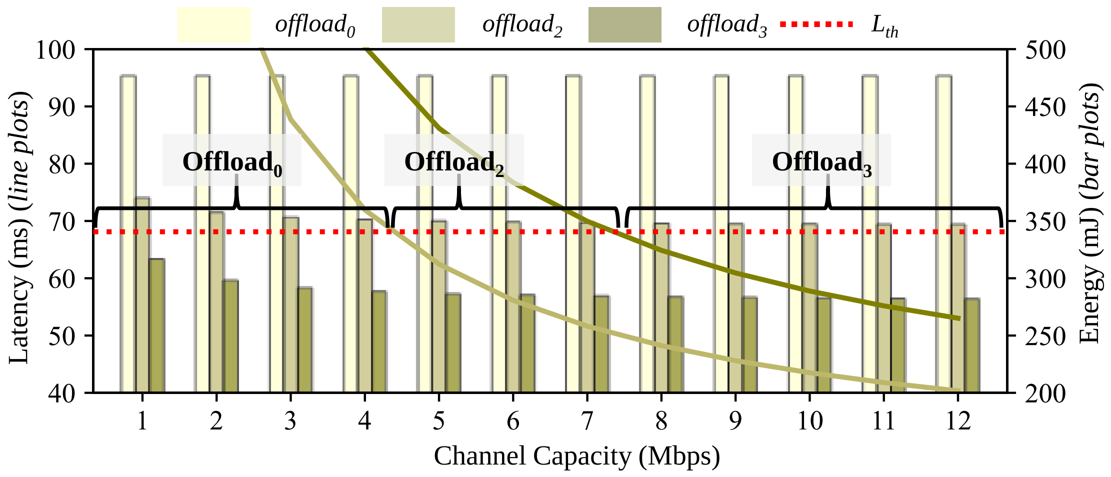


Figure 5.8: Variation of Latency and Energy Analysis w.r.t. Channel Capacity. Energy as bar charts, Latency as plot lines

but the L_{th} constraint dictates which action should be chosen considering how poor values of ϕ could disqualify some offloading choices. When $\phi > 4$ Mbps, the latency overhead for *offload*$_2$ does not exceed L_{th} making it the optimal action until $\phi > 7$ Mbps, at which the most energy-efficient option, *offload*$_3$, becomes valid. Similarly, this analysis is repeated in Figure 5.9 when sweeping across q_t given $\phi = 8$ Mbps. Naturally, the latency overhead is linearly proportional to q_t under fixed network conditions, demonstrating the influence of server load over the optimal offloading decision. From here, the key takeaway is that based on the wireless infrastructure and VEC server capabilities, the maximum number of concurrent offloading pipelines that meet L_{th} can be determined, and used accordingly to construct the decision space.

Robustness Analysis. To analyze the DRLs capacity to make robust decisions, we define 2 baseline policies for comparison: (i) a robustness-agnostic or *R-agnostic* policy that is aware of ϕ and q_t to optimize for energy so long as L_{th} is satisfied, and (ii) an *Oracle* resembling an optimal strategy which in addition to the information available to the *R-agnostic* policy, also possesses the true per-frame mAP estimate apriori, and consequently, the optimal sequence of decisions satisfying the robustness constraint mAP$_{th}$. All of the mentioned strategies are more energy-efficient than pure local execution, and the mAP$_{th}$ values are set in the

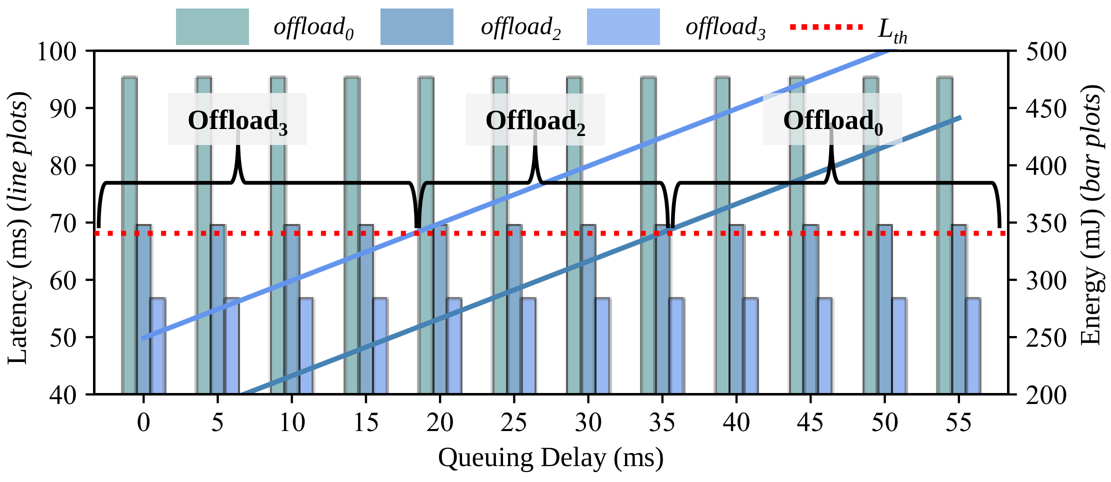


Figure 5.9: Variation of Latency and Energy Analysis w.r.t. Queuing Delay. Energy as bar charts, Latency as plot lines

Table 5.3: Robustness analysis at ρ=0.97

Policy	mAP$_{th}$	Average mAP (%)		
		$offload_0$	$offload_2$	$offload_3$
R-agnostic	N/A	64.93	64.50	64.85
DRL	0.50	60.39	67.69	68.58
	0.68	60.68	70.93	72.55
	0.98	61.82	72.49	73.14
Oracle	0.50	49.58	85.34	85.26
	0.68	50.58	93.68	93.62
	0.98	54.84	99.51	99.49

experiments to 0.5, 0.68, and 0.98, estimated based on the cumulative distribution of the evaluation dataset.

To evaluate robustness across each policy, we compute the average experienced mAP per action (AMAP) given the action selection frequencies. Mainly, a robust behavior cause frames of high uncertainty (mAP $\leq$ mAP$_{th}$) to be processed locally, implying how the AMAP experienced locally should be low compared to those from the offloading actions. We illustrate this concept in Figure 5.10 across the 3 policies for mAP$_{th}$ = 0.68 and ρ = 0.97. As seen, the *R-agnostic* policy only considers performance efficiency for its action selection, and subsequently, its AMAP across *offload*$_0$, *offload*$_2$, and *offload*$_3$, are equivalent with values of 64.39%, 65.01%, and 65.25%, respectively. Conversely, the *Oracle* resembles the ideal em-

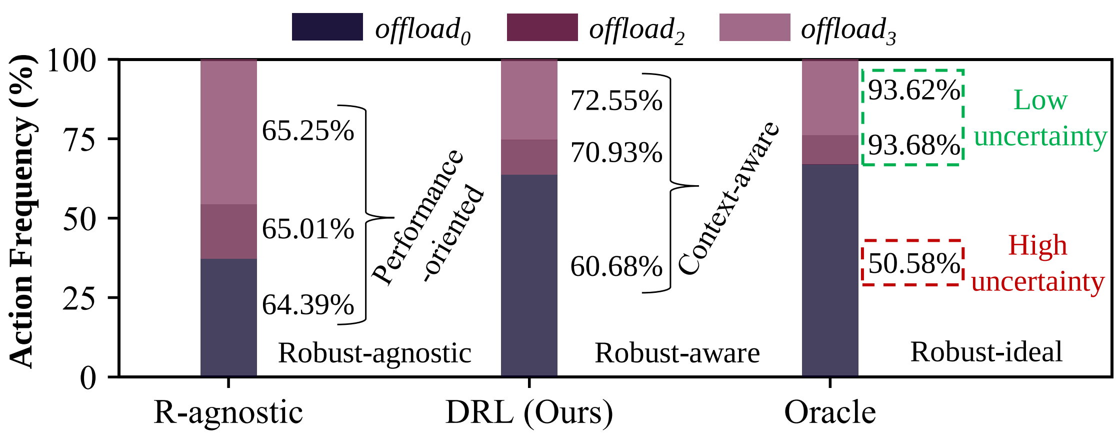


Figure 5.10: Breakdown of the action selection frequencies (%) across the 3 policies at mAP$_{th}$=0.68 and ρ=0.97. Numbers next to the bars indicate the average experienced mAP (AMAP) for the evaluation dataset inputs mapped to each action.

bodiment of robustness, assigning high uncertainty frames to *offload*$_0$, despite performance gains from offloading. In contrast to the *R-agnostic* policy, *66.94%* of the *Oracle* policy decisions are *offload*$_0$ with an AMAP of 50.58%, and an AMAP as high as *93.68%* for the remaining offloading decisions. From here, our proposed *DRL* approach strives to learn the *Oracle*'s behavior, through the observed action selection breakdown, with 63.65% of actions belonging to *offload*$_0$. Moreover, the AMAP for *offload*$_2$ and *offload*$_3$ are 70.93% and 72.55%, respectively, which despite outperforming the *R-agnostic* policy, are far from that of the *Oracle*. This is expected considering the *Oracle* policy is the unrealistic ideal behavior with apriori mAP knowledge. We extend this analysis to other thresholds values in Table 5.3, where we observe that as the robustness constraint becomes smaller, the *DRL* exhibits a behavior closer to the *R-agnostic* and farther from the *Oracle* and vice versa.

Furthermore, we vary the server load ρ in Table 5.4 and show how the action frequency varies for each policy. As ρ increases, the selection of local processing becomes more frequent across all policies, irrespective of the energy or robustness due to the L_{th} constraint. Such behavior is learned by our *DRL* solution given how the action selection frequencies closely imitate that of the *Oracle*.

Table 5.4: Action frequency analysis at $\text{mAP}_{th} = 0.68$

Policy	ρ	Action Frequency (%)		
		$offload_0$	$offload_2$	$offload_3$
R-agnostic	0.90	11.17	14.73	74.10
	0.97	37.18	17.17	45.65
	0.99	70.34	9.25	20.41
DRL	0.90	51.73	8.83	39.44
	0.97	63.65	10.95	25.39
	0.99	85.14	5.46	9.4
Oracle	0.90	53.19	8.00	38.82
	0.97	66.94	9.17	23.89
	0.99	84.37	5.04	10.60

Table 5.5: Energy analysis relative to local at $\text{mAP}_{th} = 0.98$

Metric	Local	R-agnostic	DRL	Oracle
Risky Actions (%)	0	63.37	**14.54**	0
Robust Actions (%)	100	36.63	**85.46**	100
Total Energy (kJ)	2.916	**1.729**	2.479	2.487
Total Energy Red. (%)	0	**40.7**	14.99	14.72

Energy Reduction vs Risky Actions. We also compare the energy savings relative to the pure local execution, *offload*$_0$, in addition to their risky behaviors. We first define *Risky Actions* as the fraction of offloading actions whose respective mAP scores fall below mAP_{th}, and *Robust Actions* as the fraction whose scores exceed the mAP_{th}. We compare the performance of each policy in Table 5.5, where although *R-agnostic* offers the highest energy reduction of 40.7% compared to *DRL*'s 14.99%, 63.37% of *R-agnostic*'s energy savings are *Risky Actions*, unlike *DRL* whose *Risky Actions* constitute 14.54% of the offloading decisions. Through extending this analysis further to entail multiple mAP_{th} values, i.e., a higher threshold means a stricter offloading constraint, we observe in Figure 5.11 that the robustness-aware *DRL* at higher mAP_{th} substantially reduces the percentage of risky offloads compared to the *R-agnostic* policy, with reductions reaching 77.06% at $\text{mAP}_{th} = 0.98$.

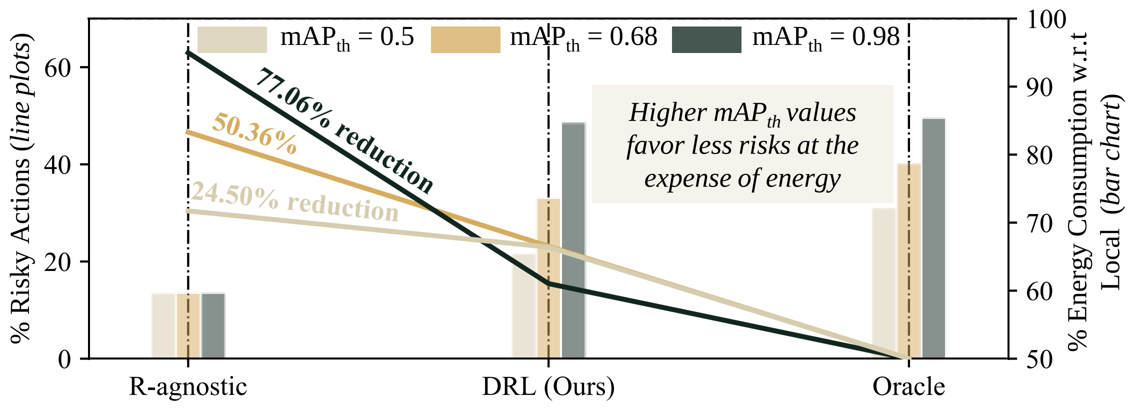


Figure 5.11: Comparing Energy and Risky Actions

5.6 Concluding Remarks and Future Directions

We studied edge (split) computing and analyzed how it can be effective in elevating the performance efficiency for autonomous systems operating through DNNs. We first demonstrated the merit of bottlenecks in multi-sensor ADS platforms. Then, we implemented a novel, hierarchical DRL solutions for runtime adaptation. Our results were promising.

Still, we highlight three areas for improvement. (i) Evaluations using other wireless technologies, C-V2X and 5G, are still needed. (ii) Real-world wireless experiments are still needed to realize more tangible outcomes considering unpredictable network conditions from unfamiliar dynamic factors (e.g., motion characteristics). (iii) The used datasets are relatively small for the DRL to learn representative policies. Exposure to larger, more diversified is needed for a large-scale adoption.

Lastly, the adoption of edge computing and data offloading methods introduce a number of concerns such as: (i) Formal Safety Guarantees on control behavior; For autonomous systems, provably-safe control behavior can be compromised under wireless channel uncertainty and potential delays. (ii) Privacy Risks; Transmitting user generated data over wireless networks to third party servers have been shown to cause serious concerns with regards to sensitive user information leakage . We study these problems in Chapter 6 and Appendix A.

Chapter 6

Provably-Safe Offloading for Edge Neural Network Controllers

Neural Network controllers do not operate in vacuum, but they are rather part of a larger system having formal properties and operational requirements. The deep-learning based autonomous driving system is part of a broader vehicular system with formal guarantees on execution latency and safety. Emerging techniques like split computing disrupt the traditional computational stack, and compromise existing formal safety guarantees due to lack of consideration for wireless uncertainty. We present through the autonomous driving use case a provably-safe, formal approach to achieve edge computing with safety guarantees.

6.1 Introduction

As was discussed in the prior Chapter, even modern wireless networks and offloading-friendly DNN architectures (e.g. encoder/decoders) cannot provide *formal guarantees* that bringing edge computing into the ADS control loop will have equivalent (or even acceptable) per-

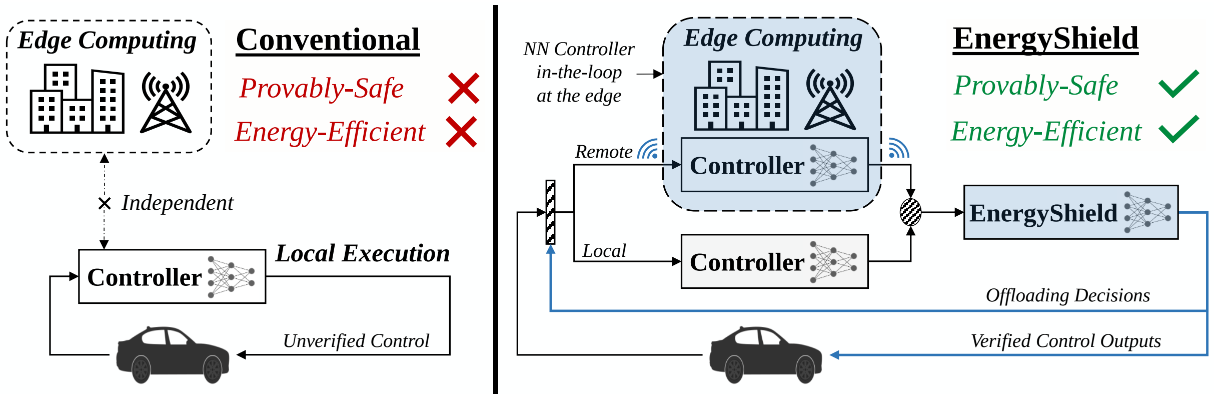


Figure 6.1: Illustration of Provably-Safe Offloading of Edge Neural Network Controllers.

formance compared to on-vehicle hardware. This is an unacceptable situation when human lives are at stake: even relatively rare and short delays in obtaining a control action or perception classification can have fatal consequences.

Hence, we propose the EnergyShield framework as a mechanism to perform DNN-to-edge offloading of ADS controllers but *in a formal, provably safe way.* Thus, EnergyShield is the first framework that enables significantly lower on-vehicle energy usage when evaluating large DNNs by intelligently offloading those calculations to edge compute in a provably safe way; see Figure 6.1. The primary idea of EnergyShield is to perform safety-aware (state-) contextual offloading of DNN calculations to the edge, under the assumption that adequate on-vehicle computation is always available as a safety fallback. This is accomplished using a controller "shield" as both a mechanism to enforce safety *and* as a novel, low-power runtime safety monitor. In particular, this shield-based safety monitor provides provably safe edge-compute response times: i.e., at each instance, EnergyShield provides a time interval within which the edge-compute must respond to an offloading request in order to *guarantee* safety of the vehicle in the interim. In the event that the edge resources don't provide a response within this time, on-board local compute proceeds to evaluate the relevant DNNs before vehicle safety is no longer assured. Further energy savings are obtained by incorporating an estimator to anticipate edge-compute load and wireless network throughput; a more intelligent offloading decision is made by comparing this estimate against the tolerable edge-compute delay provided by the runtime safety monitor, avoiding offloads likely to fail.

The main technical novelty of EnergyShield is its shield-based runtime safety monitor mentioned above. Although controller "shielding" is a well-known methodology to render generic controllers safe, the shielding aspect of EnergyShield contains two important novel contributions of its own: first, in the use of a shield not only to enforce safety but also as a runtime safety monitor to quantify the *time until the system is unsafe*; and second, in the specific design of that runtime monitor with regard to implementation complexity and energy considerations. In the first case, EnergyShield extends existing notions wherein the *current value* of a (Zeroing-)Barrier Function (ZBF) is used as a runtime monitor to quantify the safety of an agent: in particular, it is novel in EnergyShield to instead use the current value of the ZBF to derive a *sound* quantification of the *time* until the agent becomes unsafe. Moreover, EnergyShield implements this sound quantification in an extremely energy efficient way: i.e., via a small lookup table that requires only a small number of FLOPS to obtain a guaranteed time-until-unsafe. This particular aspect of the runtime safety monitor is also facilitated by using a particular, but known, ZBF and shield [55] in EnergyShield: these components are both extremely simple, and so implementable using small, energy efficient NNs [55]. Together, these design choices ensure that any energy saved by offloading is not subsequently expended in the implementation of EnergyShield itself.

We tested EnergyShield in the Carla simulation environment [44] with several DRL-trained agents. Our experiments showed that EnergyShield entirely eliminated obstacle collisions for all RL agents – i.e. made them safe – while simultaneously reducing NN energy consumption by as much as 54%. Additionally, we showed that EnergyShield has intuitive, safety-conscious offloading behavior: when the ADS is near an obstacle – and hence less safe – EnergyShield's runtime safety monitor effectively forced exclusively on-vehicle NN evaluation; when the ADS was further from an obstacle – and hence more safe – EnergyShield's runtime safety monitor allowed more offloading, and hence more energy savings.

6.1.1 Related Work

Formal Methods for Data-Trained Controllers. A number of approaches exist to assert the safety of data-trained controllers with formal guarantees; in most, ideas from control theory are used in some way to augment the trained controllers to this end. A good survey of these methods is [41]. Examples of this approach include the use of Lyapunov methods [16, 36], safe model predictive control [96], reachability analysis [9, 56, 64], barrier certificates [115, 160, 176, 185, 197, 207], and online learning of uncertainties [169].

Controller "shielding" [10] is another technique that often falls in the barrier function category [35]. Another approach tries to verify the formal safety properties of learned controllers using formal verification techniques (e.g., model checking): e.g., through the use of SMT-like solvers [46, 110, 178] or hybrid-system verification [53, 81, 205]. However, these techniques only assess the safety of a given controller rather than design or train a safe agent.

Edge Computing for Autonomous Systems. A number of different edge/cloud offloading schemes have been proposed for ADSs, however none to date has provided formal guarantees. Some have focused on scheduling techniques and network topology to achieve effective offloading [39, 54, 163, 184, 227]. Others focused on split and other NN architectures to make offloading more efficient [32, 119, 138].

6.2 Preliminaries

6.2.1 Notation

Let $\mathbb{R}$ denote the real numbers; $\mathbb{R}^+$ the non-negative real numbers; $\mathbb{N}$ the natural numbers; and $\mathbb{Z}$ the integers. For a continuous-time signal, $x(t), t \geq 0$, denote its discrete-time sampled

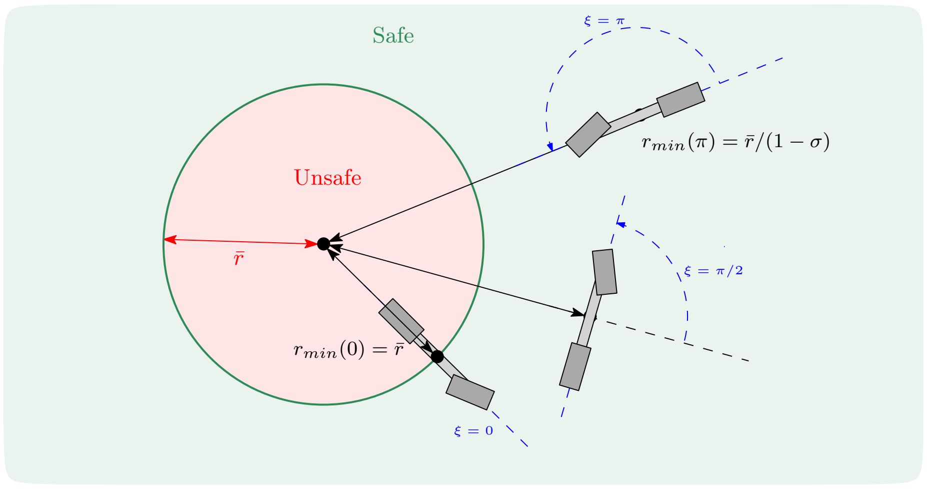


Figure 6.2: Obstacle specification and minimum barrier distance as a function of relative vehicle orientation, ξ.

version as $x[n]$ for some fixed sample period T (in seconds); i.e. let $x[n] \triangleq x(n \cdot T)$ for $n \in \mathbb{Z}$. Let $\mathbf{1}_a : \mathbb{R} \to \{a\}$ be the constant function with value a; i.e., $\mathbf{1}_a(x) = a$ for all $x \in \mathbb{R}$ (interpreted as a sequence as needed). Finally, let $\dot{x} = f(x, u)$ be a control system with $x \in \mathbb{R}^n$ and $u \in \mathbb{R}^m$, and let $\pi : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^m$ be a (possibly) time-varying controller. For this system and controller, consider a time $t_0 \geq 0$ and state x_0, and denote by $\zeta_\pi^{t_0, x_0} : \mathbb{R}^+ \to \mathbb{R}^n$ the t_0-shifted state evolution of this system controlled by π assuming $x(t_0) = x_0$. Let $\zeta_\pi^{n_0, x_0}[n]$ indicate the same, except in discrete-time with zero-order hold of π. $\|\cdot\|$ and $\|\cdot\|_2$ will denote the max and two-norms on $\mathbb{R}^n$, respectively.

6.2.2 Kinematic Bicycle Model

In this paper, we will use the kinematic bicycle model (KBM) as the formal dynamical model for our autonomous vehicle [97]. However, we consider the KBM model in terms of state variables relative to a fixed point in the plane – the obstacle to be avoided – rather than absolute Cartesian coordinates. That is, the positional states are the distance to a fixed point, $\|\vec{r}\|$, and orientation angle, ξ, of the vehicle with respect to the same. These evolve according to dynamics:

$$\begin{pmatrix} \dot{r} \\ \dot{\xi} \\ \dot{v} \end{pmatrix} = \begin{pmatrix} v\cos(\xi-\beta) \\ -\frac{1}{r}v\sin(\xi-\beta)-\frac{v}{\ell_r}\sin(\beta) \\ a \end{pmatrix}; \quad \beta \triangleq \tan^{-1}(\tfrac{\ell_r}{\ell_f+\ell_r}\tan(\delta_f)) \tag{6.1}$$

where $r(t) \triangleq \|\vec{r}\|$ and ξ are as described above; v is the vehicle's linear velocity; a is the linear acceleration input; δ_f is the front-wheel steering angle input[1]; and ℓ_f and ℓ_r are the distances of the front and rear axles, respectively from the vehicle's center of mass. Note that at $\xi = \pm\pi/2$, the vehicle is oriented tangentially to the obstacle; and at $\xi = \pi$ or 0, the vehicle is pointing directly at or away from the origin, respectively (see Figure 6.2).

We assume that the KBM has a steering constraint, i.e. $\delta_f \in [-\delta_{f\,\text{max}}, \delta_{f\,\text{max}}]$. However, we may use β directly as a control variable, since it is an invertible function of δ_f. Thus, β is also constrained as $\beta \in [-\beta_{\text{max}},\, \beta_{\text{max}}]$.

We define the state and control vectors for the KBM as: $\chi \triangleq (\xi, r, v)$ and $\omega \triangleq (a, \beta)$, with $\omega \in \Omega_{\text{admis.}} \triangleq \mathbb{R} \times [-\beta_{\text{max}}, \beta_{\text{max}}]$ the set of admissible controls. Thus, the dynamics of the KBM are given by $\dot{\chi} = f_{\text{KBM}}(\chi, \omega)$ with f_{KBM} defined by (6.1).

6.2.3 Barrier Functions and Shielding

In the sequel, we will use a controller "shield", which is a methodology for instantaneously correcting the outputs produced by a controller in closed loop; the objective is to make corrections such that the original controller, however it was designed or implemented, becomes safe – hence the "shield" moniker. Specifically, a controller shield is designed around a real-valued function over the state space of interest, called a (Zeroing-) Barrier Function (ZBF). The ZBF directly encodes a set safe states by its sign: states for which the ZBF is nonnegative are taken to be safe. The ZBF in turn indirectly specifies safe controls (as a

[1]That is, the steering angle can be set instantaneously with no steering rack dynamics.

function of state) in such a way that the sign of the ZBF is invariant along trajectories of the dynamics.

Formally, consider a control system $\dot{x} = f(x, u)$ in closed loop with a state-feedback controller $\pi : x \mapsto u$. In this scenario, a feedback controller in closed loop converts the control system into an autonomous one – the autonomous vector field $f(\cdot, \pi(\cdot))$. In this context, recall the definition of a Zeroing-Barrier Function (ZBF):

Definition 6.1 (Zeroing Barrier Function (ZBF) [211, Definition 2])**.** *Let $\dot{x} = f(x, \pi(x))$ be the aforementioned closed-loop, autonomous system with $x(t) \in \mathbb{R}^n$. Also, let $h : \mathbb{R}^n \to \mathbb{R}$, and define $\mathcal{C} \triangleq \{x \in \mathbb{R}^n : h(x) \geq 0\}$. If there exists a locally Lipschitz, extended-class-K function, α such that:*

$$\nabla_x h(x) \cdot f(x, \pi(x)) \geq -\alpha(h(x)) \text{ for all } x \in \mathcal{C} \tag{6.2}$$

then h is said to be a ***zeroing barrier function (ZBF)****.*

Moreover, the conditions for a barrier function above can be translated into a set membership problem for the outputs of such a feedback controller. This is explained in the following proposition.

Proposition 1. *Let $\dot{x} = f(x, u)$ be a control system that is Lipschitz continuous in both of its arguments on a set $\mathcal{D} \times \Omega_{admis.}$; furthermore, let $h : \mathbb{R}^n \to \mathbb{R}$ with $\mathcal{C}_h \triangleq \{x \in \mathbb{R}^n | h(x) \geq 0\} \subseteq \mathcal{D}$, and let α be a class $\mathcal{K}$ function. If the set*

$$R_{h,\alpha}(x) \triangleq \{u \in \Omega_{admis.} | \nabla_x^T h(x) \cdot f(x, u) + \alpha(h(x)) \geq 0\} \tag{6.3}$$

is non-empty for each $x \in \mathcal{D}$, and a Lipschitz continuous feedback controller $\pi : x \mapsto u$

satisfies

$$\pi(x) \in R_{h,\alpha}(x) \quad \forall x \in \mathcal{D} \tag{6.4}$$

then $\mathcal{C}_h$ is forward invariant for the closed-loop dynamics $f(\cdot, \pi(\cdot))$.

In particular, if π satisfies (6.4) and $x(t)$ is a trajectory of $\dot{x} = f(x, \pi(x))$ with $h(x(0)) \geq 0$, then $h(x(t)) \geq 0$ for all $t \geq 0$.

Proof. A direct application of ZBFs [211, Theorem 1]. □

Proposition 1 is the foundation for controller shielding: (6.3) and (6.4) establish that h (and associated α) forms a ZBF for the closed-loop, autonomous dynamics $f(\cdot, \pi(\cdot))$. Note also that there is no need to distinguish between a closed-loop feedback controller π, and a composite of π with a function that *shields* (or filters) its output based on the current state. Hence, the following definition:

Definition 6.2 (Controller Shield)**.** *Let $\dot{x} = f(x, u)$, h, $\mathfrak{C}_h$, α and $\mathcal{D} \times \Omega_{admis.}$ be as in Proposition 1. Then a **controller shield** is a Lipschitz continuous function $\mathfrak{S} : \mathcal{D} \times \Omega_{admis.} \to \Omega_{admis.}$ such that*

$$\forall (x, u) \in \mathcal{D} \times \Omega_{admis.} . \mathfrak{S}(x, u) \in R_{h,\alpha}(x). \tag{6.5}$$

6.2.4 A Controller Shield for the KBM

In this paper, we will make use of the existing ZBF and controller shield designed for the KBM in [55]. These function in concert to provide controller shielding for the safety property illustrated in Figure 6.2: i.e., to prevent the KBM from entering a disk of radius $\bar{r}$ centered at the origin. In particular, [55] proposes the following class of *candidate* ZBFs for the KBM:

$$h_{\bar{r},\sigma}(\chi) = h_{\bar{r},\sigma}(\xi, r, v) = \tfrac{\sigma\cos(\xi/2)+1-\sigma}{\bar{r}} - \tfrac{1}{r} \tag{6.6}$$

$$\alpha_{v_{\max}}(x) = K \cdot v_{\max} \cdot x \tag{6.7}$$

where $\alpha_{v_{\max}}$ is per se a class $\mathcal{K}$ function, and $\sigma \in (0,1)$ parameterizes the class. Note also that this class of ZBFs ignores the state variable, v; it is a result in [55] that this class is useful as a barrier function provided the vehicle velocity remains (is controlled) within the range $(0, v_{\max}]$. Note also that the equation has $h_{\bar{r},\sigma}(\chi) = 0$ has a convenient solution, which we denote by $r_{\min}$ for future reference:

$$r_{\min}(\xi) = \bar{r}/(\sigma\cos(\xi/2) + 1 - \sigma). \tag{6.8}$$

One main result in [55] is a mechanism for choosing the parameter σ as a function of KBM parameters (e.g. ℓ_r) and safety parameter, $\bar{r}$ so that the resulting specific function is indeed a ZBF as required.

Finally, we note that [55] also suggests an extremely lightweight implementation of the barrier based on (6.6). That is, it contains a "Shield Synthesizer" that implements a controller shield by approximating a simple single-input/single-output concave function with a ReLU NN [55, pp 6]. This construction will also prove advantageous later. We denote by $\mathfrak{S}_{\text{KBM}}$ the resulting controller shield, with associated barrier, KBM and safety parameters inferred from the context.

6.3 Framework

NOTE: In this section, we will denote by x, y and u the state, sensor and control variables of an ADS, respectively; this abstract notation will illustrate the EnergyShield framework free from specific modelling details. A formal consideration of EnergyShield is in Section 6.4.

6.3.1 EnergyShield Motivation and Context

The basic motivation for the EnergyShield framework is the following. Suppose that an ADS contains a large NN, $\mathcal{NN}_c$, that is responsible for producing a control action, u, in response to a sensor signal, y. Further assume that, by virtue of its size, computing an output of $\mathcal{NN}_c$ with on-vehicle hardware consumes significant energy. Thus, it would be advantageous, energy-wise, to *offload* evaluations of $\mathcal{NN}_c$ to edge computing infrastructure: in other words, wirelessly transmit a particular sensor measurement, y, to off-vehicle edge computers, where the output $u = \mathcal{NN}_c(y)$ is computed and returned to the requesting ADS.

The problem with this approach is largely from a safety point of view. In particular, the controller $\mathcal{NN}_c$ was designed to operate *in real time and in closed-loop*: i.e. the control action at discrete-time sample n is intended to be computed from the sensor measurement at the *same* time sample[2]. In the notation of discrete-time signals (see Section 6.2.1), this means: $u[n] = \mathcal{NN}_c(y[n])$. However, offloading a sensor measurement, $y[n]$, to the edge entails that the correct output of $\mathcal{NN}_c(y[n])$ will not be back on-vehicle before some non-zero number of samples, say Δ. Thus, $\mathcal{NN}_c(y[n])$ will not be available at time n to set $u[n] = \mathcal{NN}_c(y[n])$ as intended; rather, the soonest possible use of the output $\mathcal{NN}_c(y[n])$ will be at time $n + \Delta$, or $u[n + \Delta] = \mathcal{NN}_c(y[n])$. This delay creates obvious safety issues, since the state of the vehicle – and hence the correct control to apply – will have changed in the intervening Δ time samples. More importantly, even the "correct" control action applied at $n + \Delta$ may be

[2]In our formal consideration, we will model a one-sample computation delay.

insufficient to ensure safety: e.g., after Δ samples have elapsed, it may be too late to apply adequate evasive steering.

6.3.2 EnergyShield Structure

If we assume the ADS has enough on-vehicle compute to obtain an output $\mathcal{NN}_c(y[n])$ *in real time*[2], then the safety problem above is one of making an *offloading decision*: ideally one that saves energy without compromising safety. That is, should a particular evaluation of $\mathcal{NN}_c$ be offloaded to the edge? And how long should the ADS wait for a response so as to ensure the situation is correctable?

EnergyShield must address two intertwined issues in order to ensure safety of the ADS vehicle during offloading. On the one hand, EnergyShield must be able to *correct* the control actions provided by $\mathcal{NN}_c$ after an offload decision (see explanation above). On the other hand, EnergyShield must limit the duration it waits for each offloading request, so that actions provided by $\mathcal{NN}_c$ *can* be corrected in the first place; i.e., among all possible offloading delays, Δ, it is not immediate which may be corrected and which may not (e.g., $\Delta = \infty$ likely cannot be corrected). In this sense, knowing a particular response-delay, Δ', is correctable essentially characterizes how to take an offloading decision, since it provides an ***expiration on safety***: i.e., proceed to offload, and wait for a response until Δ' samples have elapsed – at which point resume local evaluation of $\mathcal{NN}_c$.

In particular then, EnergyShield has two main components:

- **C1: Controller Shield.** EnergyShield contains a controller shield (see Section 6.2.3), which ensures that safety is maintained irrespective of offloading-delayed controller outputs; in other words, it corrects unsafe behavior of $\mathcal{NN}_c$ that results from changes in vehicle state during offloading delays.

C2: Runtime Safety Monitor. EnergyShield contains a runtime safety monitor that provides the ADS an upper bound, Δ_{max} (in samples), on how long it should wait for a response to one of its offloading requests to maintain safety, *assuming no updates to the control action in the meantime*; i.e., provided the offload delay is $\Delta \leq \Delta_{\text{max}}$, then **C1**, the controller shield, can guarantee safe recovery after holding the last control signal update through offload delay period (**C1** may use on-vehicle computation if necessary). ***In other words,*** Δ_{max} ***specifies an expiration for the safety guarantee provided by*** **C1** ***using on-vehicle computation.***

Naturally, **C1** and **C2** need to be designed together, since their objectives are mutually informed. Indeed, in the specific design of EnergyShield, these components are designed from the same ZBF (defined in Section 6.2.3): see Section 6.4 for formal details.

Unfortunately, neither component **C1** nor **C2** can operate effectively on the same raw sensor measurements, $y[n]$, used by the controller; this is especially true given our intention to implement them via ZBFs and controller shields. In particular, both require some (limited) state information about the ADS in order to perform their tasks. Thus, EnergyShield requires a perception/estimator component to provide state information to **C1** and **C2**. Note that we deliberately exclude the design of such an estimator from the EnergyShield framework in order to provide additional flexibility: in particular, since the controller $\mathcal{NN}_c$ may effectively contain an estimator, we wish to allow for estimation to be offloaded, too – provided it is executed locally just-in-time before informing **C1** and **C2** (see Section 6.3.3). Nevertheless such an estimator is necessary for EnergyShield, so we include it as component:

C3: State Estimator. EnergyShield requires (minimal) state estimates as input to **C1** and **C2**. By convention, this estimator will be a NN denoted by $\mathcal{NN}_p : y \mapsto x$. *We assume that* $\mathcal{NN}_p$ *can be computed by on-vehicle hardware in one sample period.*

The interface of **C3** with both **C1** and **C2** makes the latter two components (state-)context aware. *That is, EnergyShield makes* ***context-aware offloading decisions*** *based on the current vehicle state.* Moreover, it is important to note that since the prior control action will be held during offload $\Delta_{\max}$, the output of **C2** is control in addition to state dependent: that is, **C2** actually produces an output $\Delta_{\max}(\hat{x}, u)$ for (arbitrary) state x and the control u applied just before offload.

EnergyShield has one further important component, but one that is motivated purely by energy savings with no effect on safety. Crucially, the known expiration of safety provided by **C2**, i.e. $\Delta_{\max}(x, u)$, affords the opportunity to use additional information in making an offload decision. In particular, an estimate of the anticipated edge response time, $\hat{\Delta}$, can be used to *forego* offloads that are unlikely to complete before the expiration of the safety deadline, $\Delta_{\max}(x, u)$. For this reason, EnergyShield contains an estimator of edge response time to preemptively skip offloads that are likely to fail:

C4: **Edge-Response Estimator.** EnergyShield specifies that an estimate of the current edge response time, $\hat{\Delta}$, is provided to inform offloading decisions.

We note that *EnergyShield doesn't specify a particular estimator to be used in this component*: any number of different estimators may be appropriate, and each estimator may lead to different energy profiles. Moreover, since $\hat{\Delta}$ is never used to override $\Delta_{\max}(x, u)$, safety is preserved irrespective of the specific estimator used.

The interconnection of the components **C1** through **C4** in EnergyShield is illustrated in Figure 6.3. Note that component **C3**, the state estimator, is connected to components **C1** and **C2**, the controller shield and safety runtime monitor, respectively. Also note that the output of **C2** provides a signal $\Delta_{\max}(x, u)$ to the offloading decision switch; also informing that decision is the estimate of immediate edge-response times provided by component **C4**.

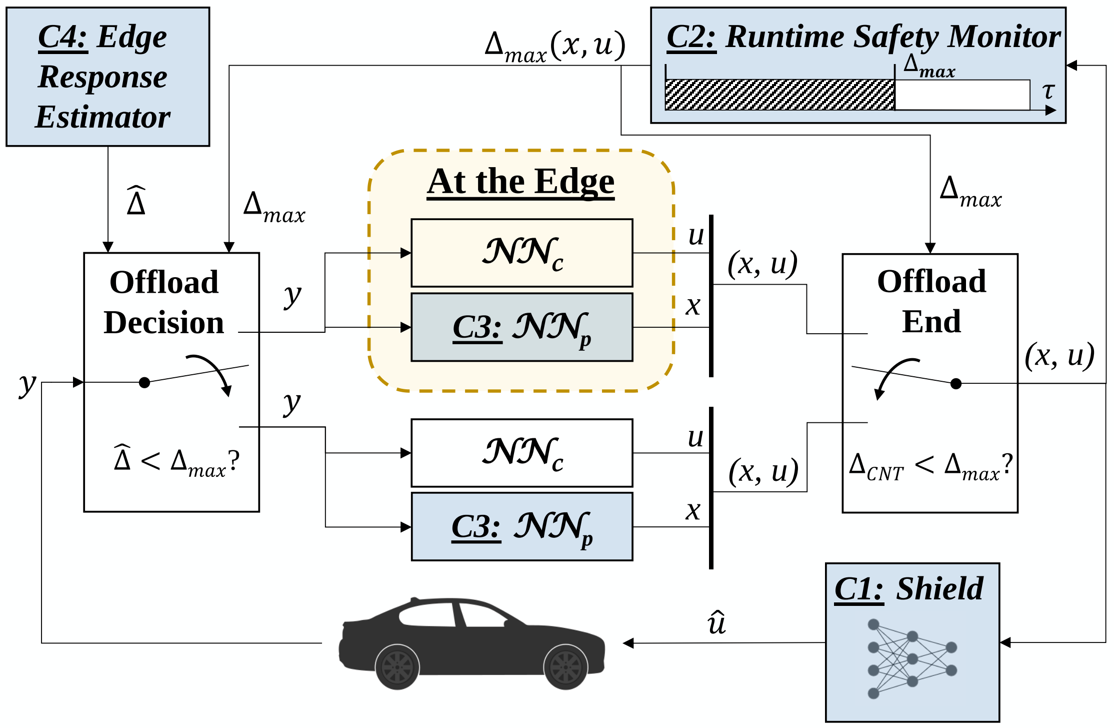
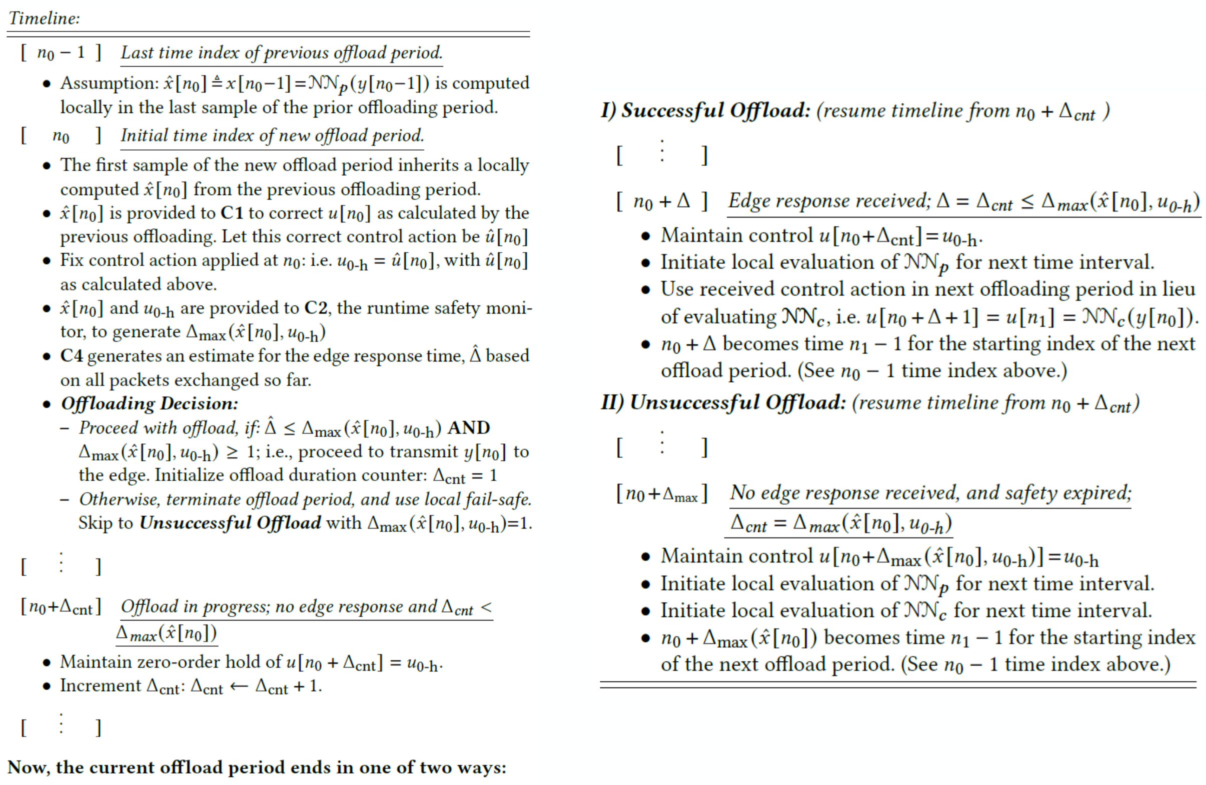


Figure 6.3: EnergyShield Framework

6.3.3 Semantics of an EnergyShield Offloading Decision

In this subsection, we consider the timeline of a single, hypothetical EnergyShield offloading decision to illustrate the details of the interplay between the components described in Section 6.3.2. In particular, suppose that an offloading interval has just been completed, and at time index n_0 a new offloading decision is to be taken.

We call the time between the initialization of an offloading decision and the time that offloading decision has been resolved an ***offloading period*** (the resolution is either by a response from the edge or a fail-over to on-vehicle compute). Figure 6.4 provides the timeline of actions during a single offloading period. In particular, note two crucial facts. First, if **C2** returns $\Delta_{\max}(\hat{x}, u) = 0$, then it effectively forces pure on-vehicle evaluation of $\mathcal{NN}_c$ and $\mathcal{NN}_p$. Second, we ensured that an up-to-date estimate of the state is always available for both **C1** and **C2** before they have to act.



Timeline:

[$n_0 - 1$] *Last time index of previous offload period.*

- Assumption: $\hat{x}[n_0] \triangleq x[n_0-1] = \mathcal{NN}_p(y[n_0-1])$ is computed locally in the last sample of the prior offloading period.

[n_0] *Initial time index of new offload period.*

- The first sample of the new offload period inherits a locally computed $\hat{x}[n_0]$ from the previous offloading period.
- $\hat{x}[n_0]$ is provided to **C1** to correct $u[n_0]$ as calculated by the previous offloading. Let this correct control action be $\hat{u}[n_0]$
- Fix control action applied at n_0: i.e. $u_{0\text{-h}} = \hat{u}[n_0]$, with $\hat{u}[n_0]$ as calculated above.
- $\hat{x}[n_0]$ and $u_{0\text{-h}}$ are provided to **C2**, the runtime safety monitor, to generate $\Delta_{\max}(\hat{x}[n_0], u_{0\text{-h}})$
- **C4** generates an estimate for the edge response time, $\hat{\Delta}$ based on all packets exchanged so far.
- ***Offloading Decision:***
  - *Proceed with offload, if:* $\hat{\Delta} \le \Delta_{\max}(\hat{x}[n_0], u_{0\text{-h}})$ **AND** $\Delta_{\max}(\hat{x}[n_0], u_{0\text{-h}}) \ge 1$; i.e., proceed to transmit $y[n_0]$ to the edge. Initialize offload duration counter: $\Delta_{\text{cnt}} = 1$
  - *Otherwise, terminate offload period, and use local fail-safe.* Skip to ***Unsuccessful Offload*** with $\Delta_{\max}(\hat{x}[n_0], u_{0\text{-h}})$=1.

[$\vdots$]

[n_0+Δ_{cnt}] *Offload in progress; no edge response and $\Delta_{cnt} < \Delta_{max}(\hat{x}[n_0])$*

- Maintain zero-order hold of $u[n_0 + \Delta_{\text{cnt}}] = u_{0\text{-h}}$.
- Increment Δ_{cnt}: $\Delta_{\text{cnt}} \leftarrow \Delta_{\text{cnt}} + 1$.

[$\vdots$]

Now, the current offload period ends in one of two ways:

I) Successful Offload: *(resume timeline from $n_0 + \Delta_{cnt}$)*

[$\vdots$]

[$n_0 + \Delta$] *Edge response received; $\Delta = \Delta_{cnt} \le \Delta_{max}(\hat{x}[n_0], u_{0\text{-}h})$*

- Maintain control $u[n_0+\Delta_{\text{cnt}}] = u_{0\text{-h}}$.
- Initiate local evaluation of $\mathcal{NN}_p$ for next time interval.
- Use received control action in next offloading period in lieu of evaluating $\mathcal{NN}_c$, i.e. $u[n_0 + \Delta + 1] = u[n_1] = \mathcal{NN}_c(y[n_0])$.
- $n_0 + \Delta$ becomes time $n_1 - 1$ for the starting index of the next offload period. (See $n_0 - 1$ time index above.)

II) Unsuccessful Offload: *(resume timeline from $n_0 + \Delta_{cnt}$)*

[$\vdots$]

[n_0+$\Delta_{\max}$] *No edge response received, and safety expired; $\Delta_{cnt} = \Delta_{max}(\hat{x}[n_0], u_{0\text{-}h})$*

- Maintain control $u[n_0+\Delta_{\max}(\hat{x}[n_0], u_{0\text{-h}})] = u_{0\text{-h}}$
- Initiate local evaluation of $\mathcal{NN}_p$ for next time interval.
- Initiate local evaluation of $\mathcal{NN}_c$ for next time interval.
- $n_0 + \Delta_{\max}(\hat{x}[n_0])$ becomes time $n_1 - 1$ for the starting index of the next offload period. (See $n_0 - 1$ time index above.)

Figure 6.4: Timeline for an offloading period experienced through EnergyShield.

6.4 EnergyShield: Provably Safe Offloading

6.4.1 Main Formal Result

Formal Assumptions We begin this section with a list of formal assumptions about the ADS. These are largely based around the structure of EnergyShield, as detailed in Section 6.3.

Assumption 1 (ADS Safety)**.** *Consider a fixed point in the plane as a stationary obstacle to be avoided by the ADS, and a disk of radius $\bar{r}$ around the origin to be a set of unsafe states; see Figure 6.2.*

Assumption 2 (ADS Model)**.** *Let Assumption 1 hold. Thus, suppose that the ADS vehicle is modeled by the KBM dynamics in* (6.1)*. Suppose further that interactions with this model happen in discrete time with zero-order hold. Let T be the sampling period.*

Assumption 3 (ADS Sensors)**.** *Let Assumptions 1 and 2 hold. Suppose the KBM-modeled ADS has access to samples of a sensor signal, $s[n] \in \mathbb{R}^N$, at each discrete time step, and there is a perception NN, $\mathcal{NN}_p : s[n] \mapsto \chi[n]$, which maps the sensor signal at each discrete time to the (exact) KBM state at the same time instant, $\chi[n]$.*

Assumption 4 (ADS Control)**.** *Let Assumptions 1 - 3 hold. Suppose this KBM-modeled ADS vehicle has a NN controller, $\mathcal{NN}_c : s \mapsto \omega$, which at each sample has access to the sensor measurement s.*

Assumption 5 (ADS Local Computation)**.** *Let Assumptions 1 - 4 hold. Suppose that the output of $\mathcal{NN}_p$ and $\mathcal{NN}_c$ can be computed by ADS on-vehicle hardware in less than T seconds – i.e., not instantaneously. Thus, suppose that the control action is obtained with a one-sample computation delay when using on-vehicle hardware: i.e., the control action applied at sample $n + 1$ is $\omega[n + 1] = \mathcal{NN}_c(s[n])$.*

Component Design Problems. There are two central problems that need to be solved: i.e., corresponding to the design of the two main components of EnergyShield, **C1** and **C2** (see Section 6.3). The solutions to these problems are deferred to Sections 6.4.2 and Section 6.4.3, respectively. We state them here in order to facilitate the statement of our main result in the next subsection.

Problem 1 (Controller Shield Design **(C1)**)**.** *Let Assumptions 1 - 5 hold. Then the problem is to design: first, design functions h and α such that they constitute a ZBF for the KBM (see Section 6.2.3); and then using this ZBF, design a controller shield, $\mathfrak{S}$ for the KBM model. The resulting controller shield must have the following additional property for **a discrete-time version of the KBM** with zero-order-hold inputs:*

- *Let $\chi[n_0 - 1]$ and $\chi[n_0]$ be KBM states such that $h(\chi[n_0 - 1]), h(\chi[n_0]) > 0$, and let $\chi[n_0]$ result from a feasible input $\hat{\omega}[n_0 - 1]$ applied in state $\chi[n_0 - 1]$. Then the control* *action*

$$\hat{\omega}[n_0] = \mathfrak{S}(\chi[n_0 - 1], \omega[n_0]) \tag{6.9}$$

must yield a state $\chi[n_0{+}1]$ *such that* $h(\chi[n_0{+}1]) > 0$*; i.e., the controller shield preserves safety under discretization of the KBM and one-step estimation delay (associated with* $\mathcal{NN}_p$*), as in the case of no computations being offloaded.*

Problem 2 (Runtime Safety Monitor Design **(C2)**)**.** *Let Assumptions 1 - 5 hold, and assume that* h*,* α *and* $\mathfrak{S}$ *solve Problem 1. Then the problem is to design a runtime safety monitor:*

$$\Delta_{max} : \mathbb{R}^3 \times \Omega_{admis.} \to \mathbb{N} \tag{6.10}$$

with the following property:

- *Let* $\chi[n_0 - 1]$ *be such that* $h(\chi[n_0 - 1]) > 0$*. Then for constant control,* $\omega = \omega[n_0]$*, applied to the discretized KBM starting from* $\chi[n_0 - 1]$ *the following is true:*

  $$\forall n = 0, \ldots, \Delta_{max}(\chi[n_0 - 1], \omega[n_0]) \ . \ h(\chi[n_0 - 1 + n]) > 0 \tag{6.11}$$

  i.e. the constant control $\omega = \omega[n_0]$ *preserves safety for at least* $\Delta_{max}(\chi[n_0 - 1], \omega[n_0])$ *samples from state* $\chi[n_0 - 1]$*.*

(The delay in $\chi[n_0 - 1]$ *accounts for the computation time of* $\mathcal{NN}_p$*.)*

Main Result We can now state our main result.

Theorem 6.1. *Let Assumptions 1 - 5 hold, and assume a ZBF for the KBM dynamics, using which Problem 1 and Problem 2 can be solved.*

Then the offloading policy described in Section 6.3.3 preserves safety of the KBM-modeled ADS (Assumptions 1 and 2).

Proof. The proof follows largely by construction. Each offload period is limited in duration by the runtime safety monitor; thus, a safety monitor that solves Problem 2 will ensure safety under the specified constant control action during the offload period. Then by the additional property of the controller shield in Problem 1, safety can be maintained after the offloading period ends: i.e., either by performing a new offload if there remains significant safety margin, or by executing locally if there is no offload safety margin. □

Corollary 1. *Let Assumptions 1 - 5 hold, and consider the ZBF for the KBM dynamics specified in Section 6.2.4. Then the controller shield in Section 6.4.2 uses this ZBF and solves Problem 1; likewise, the runtime monitor in Section 6.4.3 uses this ZBF and solves Problem 2. Hence, our implementation of EnergyShield is safe.*

6.4.2 KBM Controller Shield

Fortunately, we have access to a preexisting ZBF and controller shield designed for the KBM: see Section 6.2.4 [55]. That is, the ZBF is available after using the design methodology in [55] to choose the parameter σ (see Section 6.2.4); for simplicity, we will omit further discussion this design process. Thus, for this section, we refer to a fully implemented controller shield as $\mathfrak{S}_{\text{KBM}}$, with the understanding that it has been designed for the relevant KBM model and safety parameter $\bar{r}$ (see Figure 6.2); viz. Assumptions 1 and 2.

Thus, $\mathfrak{S}_{\text{KBM}}$ must be altered so that it satisfies the additional property required in Problem 1, hence the following Lemma.

Lemma 1. *Let Assumptions 1 - 5 hold as usual, and let $\mathfrak{S}_{KBM}$ be a controller shield designed under these assumptions as per Section 6.2.4.*

Then there exists a $\rho > 0$ such that the following controller shield:

$$\mathfrak{S}^{\rho}_{KBM} \colon ((r, \xi, v), \omega) \mapsto \begin{cases} \mathfrak{S}_{KBM}((r-\rho, \xi, v), \omega) & r-\rho \geq r_{min}(\xi) \\ \beta_{max} & r-\rho < r_{min}(\xi) \wedge \xi \geq 0 \\ -\beta_{max} & r-\rho < r_{min}(\xi) \wedge \xi < 0 \end{cases} \tag{6.12}$$

solves Problem 1; parameters other than ρ are defined in Section 6.2.

The proof of this Lemma is found in [139].

A further remark is in order about Lemma 1. Note that the altered controller shield $\mathfrak{S}^{\rho}_{\text{KBM}}$ maintains the energy efficient implementation of the controller shield $\mathfrak{S}_{\text{KBM}}$ as designed in [55]; the modified shield in (6.12) amounts to a threshold override of the original shield, $\mathfrak{S}_{\text{KBM}}$, using ρ and the value of $r_{\text{min}}(\xi)$, which is trivial to compute.

6.4.3 KBM Runtime Safety Monitor

Recall that the runtime safety monitor of EnergyShield must provide an *expiration* on the safety of the vehicle during an offload period, throughout which only a single, fixed control input is applied. This expiration must come with a *provable guarantee* that the vehicle safety is not compromised in the interim. In the formulation of EnergyShield and Problem 2, this means only that $h_{\bar{r},\sigma}$ must remain non-negative until the expiration of the deadline provided by the runtime safety monitor: see the condition (6.11) of Problem 2.

This formulation is convenient because it means that the problem can again be analyzed in continuous time, unlike our consideration of Problem 1 above: the conversion back to discrete time involves a floor operation; and compensating for the one-sample state delay induced by computing $\mathcal{NN}_p$ involves subtracting one sample from the result. That is, to solve Problem 2 and design an EnergyShield-safety monitor, it is sufficient provide a (real) time, ν, s.t.:

$$\forall t \in [0, \nu] \; . \; h\big(\zeta_{1_{\omega[n_0]}}^{0,\chi[n_0-1]}(t)\big) > 0. \tag{6.13}$$

That is, the flow of f_{KBM} started from $\chi[n_0 - 1]$ and using constant control $\omega[n_0]$ maintains $h > 0$ for the duration $[0, \nu]$. We emphasize that such a ν can be converted into the sample units expected for $\Delta_{\max}(\chi[n_0 - 1], \omega[n_0])$ by using a floor operation and subtracting one. Thus, we have the following Lemma, which solves Problem 2.

Lemma 2. *Let Assumptions 1 - 5 hold as usual. Let*

$$\Delta_{max}(\chi[n_0 - 1, \omega[n_0]) \triangleq \max(\lfloor \nu(\chi[n_0 - 1], \omega[n_0]) \rfloor - 1, 0) \tag{6.14}$$

where $\nu = \nu(\chi[n_0 - 1], \omega[n_0])$ solves the equation:

$$\sqrt{2} \cdot L_{h_{\bar{r},\sigma}} \cdot \|f_{KBM}(\chi[n_0 - 1], \omega[n_0])\|_2 \cdot \nu \cdot e^{L_{f_{KBM}} \cdot \nu} = h(\chi[n_0 - 1]) \tag{6.15}$$

for $L_{h_{\bar{r},\sigma}}$ and $L_{f_{KBM}}$ upper bounds on the Lipschitz constants of $h_{\bar{r},\sigma}$ and f_{KBM}, respectively. Then $\Delta_{max}(\chi[n_0 - 1, \omega[n_0])$ solves Problem 2.

The proof of Lemma 2 is detailed in our full article [139].

Lemma 2 specifies a complete solution to Problem 2, as claimed. However in its immediate form, it requires numerically solving (6.15) with each evaluation of $\Delta_{\max}(\chi[n_0 - 1], \omega[n_0])$; i.e., each time a safety expiration time is requested from the runtime safety monitor (every sample in the case where the offloading period is terminated before offload). The nature of (6.15) is such that solving it numerically is not especially burdensome – especially compared to the NN evaluations it replaces; however, it is also possible to implement soundly as a LUT to achieve greater energy efficiency.

6.5 Experiments and Findings

We assess EnergyShield on: (*i*) the extent of energy savings compared to the conventional local execution methods, (*ii*) its ability to enforce the safety through obstacle collision avoidance, (*iii*) how representative the upper bounds of the edge response time (Δ_{max}) are of the inherent risks, and (*iv*) its generality across controllers with different learnt policies.

6.5.1 Experimental Setup

Operational Policies: In addition to the baseline continuous local execution, we designate two EnergyShield offloading modes:

- ***Eager:*** a new offloading period is immediately started if the edge response has been received at the ADS or Δ_{max} expired.
- ***Uniform:*** the start of a new offloading interval is always delayed until Δ_{max} expires, regardless of whether edge responses have been received or not.

We define both these modes to reflect the attainable behavioral trade-offs of EnergyShield with regards to realizing an ideal control behavior or maximizing energy efficiency. The distinction between the two modes is shown in Figure 6.5 within the first offloading interval.

Experimental Scenario: We perform our experiments using the CARLA open-source simulator for autonomous driving research [44]. We follow the setup proposed in [55], and implement a similar experimental scenario. Basically, the scenario involves a four-wheeled vehicle travelling from a starting position A to destination B along a 100m motorway track with 4 pedestrian obstacles in its path. The first obstacle spawns after 40m of the track, while the remaining spawning positions are uniformly spaced between the first obstacle's position and the final destination with possible $\pm 10m$ variation along the longitudinal axis.

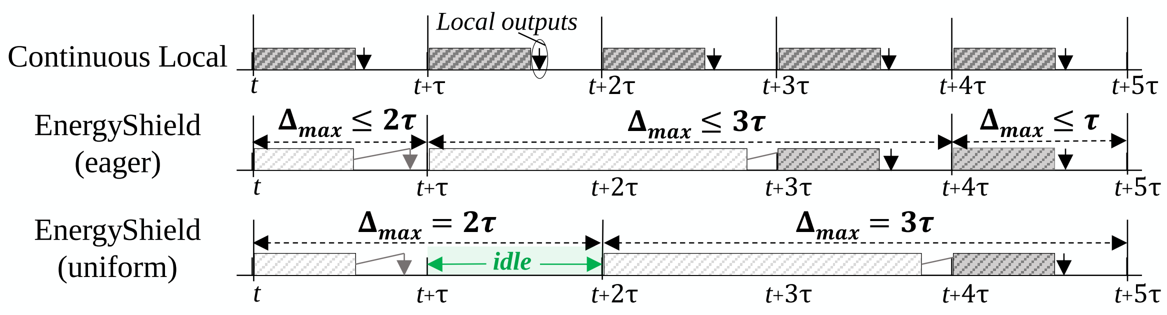


Figure 6.5: The operational policies in our experiments given base time window τ. Darker instances imply local execution.

Experimental Settings: Throughout this section, *all* of our experiments are conducted under different combinations of the following two binary configuration parameters:

- **S**: indicating whether the Controller Shield component is *active*.
- **N**: indicating whether this is a noisy version experimental test case.

In particular, the noisy version entails perturbing the obstacles' spawning positions by adding values sampled from a normal distribution $\mathcal{N}(0, 1.5m)$ along both the longitudinal and latitudinal axis.

Simulation Setup: For the controller model, its first stage entails two concurrent modules: an object detector as the large NN model of the ADS and a β Variational Autoencoder (β-VAE) providing additional latent feature representations of the driving scene. Both components operate on 160×80 RGB images from the vehicle's attached front-facing camera. In the subsequent stage, a Reinforcement Learning (RL) agent aggregates the detector's bounding box predictions, latent features, and the inertial measurements (δ_f^c, v, and a) to predict vehicle control actions (steering angle and throttle). The inertial measurements can be fetched directly from CARLA, whose positional and orientation measurements are also used directly to calculate r and ξ relative the vehicle's current nearest obstacle for obstacle state estimation. We trained the RL agents using a reward function, $\mathcal{R}$, that aims to maximize track completion rates through collision avoidance and minimize central deviance.

Performance Evaluations: We use a pretrained ResNet-152 for our object detector and benchmark its performance in terms of latency and energy consumption when deployed on the industry-grade Nvidia Drive PX2 Autochauffer ADS. We found that a single inference pass on the ResNet-152 took $\approx$ 16 ms, and accordingly, we fixed the time-step in CARLA at 20 ms. We use the Wi-Fi power model from [78] for transmission power evaluation.

Wireless Channel Model: We model the communication overheads between the ego vehicle and edge server as: $L_{comm} = L_{Tx} + L_{que}$ s.t. $L_{Tx} = \frac{data\_size}{\phi}$, where L_{que} represents potential queuing delays at the server whereas L_{Tx} is the transmission latency defined by the size of the transmission data, *data_size*, over the experienced channel throughput, ϕ.

6.5.2 EnergyShield Evaluations

The purpose of this experiment is to assess the controller's performance when supplemented with EnergyShield in terms of energy efficiency and safety. For every configuration of S and N, we run the test scenario for 35 episodes and aggregate their combined results.

Energy Efficiency: We first assess the energy efficiency gains offered by EnergyShield compared to the baseline continuous local execution. As illustrated Figure 6.6, the left barplot demonstrates that both modes of EnergyShield substantially reduce the energy consumption footprint of the NN compared to local execution across all S and N configurations. For instance, under the default configuration (S = 0, N = 0), EnergyShield energy reductions reach 20% and 40.4% for the eager and uniform modes, respectively. These numbers further improve for the subsequent configurations in which N = 1 or S = 1. Upon inspection, we find that this is the result of the ego vehicle encountering more instances in which obstacles are not in the direct line-of-sight of its heading. The reasons being that at N = 1, some ob-

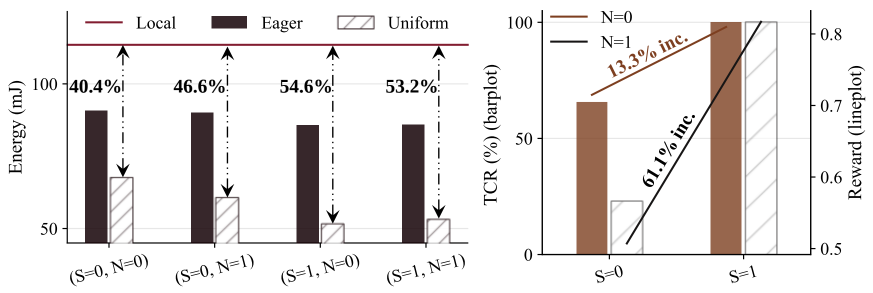


Figure 6.6: EnergyShield's energy efficiency gains with respect to continuous local execution (*left*) and safety analysis in terms of the $\mathcal{R}$ evaluation and % TCR (*right*).

stacles can be displaced out of the primary lane that the ego vehicle follows to complete the track, whereas at S = 1, such instances result from the Controller Shield applying corrective behaviors on the NN's predicted steering outputs, resulting in more tangential orientations of the vehicle with respect to the obstacles (i.e., $\xi \to \pm\pi/2$). Accordingly, large values of Δ_{max} – about 4-5 time samples (equivalent to 80-100 ms) – are increasingly sampled, and that automatically translates into more offloading decisions. For instance at (S = 1, N = 0), the energy efficiency gains reach 24.3% and 54.6% for respective eager and uniform modes.

Safety Evaluation: To assess the EnergyShield's ability to enforce safety, we designate track completion rates (TCR %) as a comparison metric to signify the proportion of times the vehicle was able to complete the track without collisions. Taking the local execution mode as the test scenario, the right barplot of Figure 6.6 shows that without an active Controller Shield (S = 0), collisions with the pedestrian obstacles cause the TCR% to be 65.7% at N = 0, and even less at 22.9% for the noisy test case (N = 1). However, when the Controller Shield is active (S = 1), collisions are completely avoided and the TCR (%) values jump to 100% for both cases. This is also visible through the respective improvements in $\mathcal{R}$ which reached 13.3% and 61.1%. To further demonstrate such occurrences, we analyze in Figure 6.7 the ego vehicle's chosen trajectories across 3 episodes of dissimilar (S, N) configurations. As shown, the (S = 0, N = 0) instance incurs a collision with the pedestrian object and does not complete the track. An active Controller Shield (S = 1), however, enforces a left or right corrective maneuvering action for obstacle avoidance and maintaining safety; see Figure 6.7

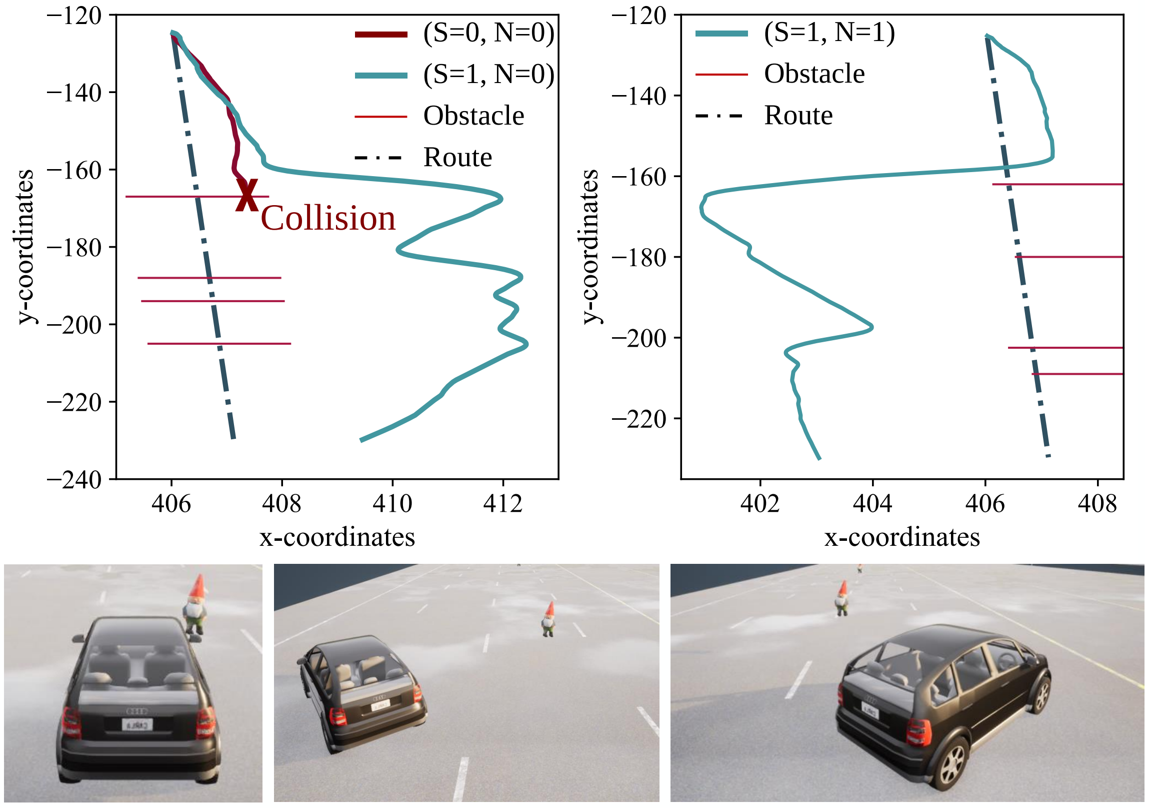


Figure 6.7: Top: Example trajectories followed by the ego vehicle with the start point at the top. Bottom: illustration of how the ego vehicle under the aforementioned operational modes behaved in reaction to the first encountered obstacle.

Energy vs. Distance: To assess how representative the Δ_{max} upper bounds provided by the Runtime Safety Monitor are of the corresponding driving scene context, we examine EnergyShield's energy consumption at different distances from the nearest obstacle (r). The hypothesis is that larger r values imply relatively "safer" driving situations, which would result in larger values of Δ_{max} to be sampled, and accordingly more offloading instances enhancing the NN's energy efficiency. As shown in Figure 6.8, we plot the average experienced normalized energy ratings of the two modes of EnergyShield with respect to local execution against r across every configuration's set of 35 episodes. Each tick on the horizontal axis accounts for an entire range of 1m distances rather than a single value – e.g., a value of 2 on the horizontal axis encompasses all distances in the range [2 - 3). At close distances (r ¡ 4m), we find that EnergyShield modes incur almost the same energy consumption overhead as that from the default local execution. This is mainly accredited to the Runtime Safety Monitor recognizing the higher risks associated with the close proximity from the objects, and accordingly outputting smaller values of Δ_{max} that can only be satisfied by local execution. As the distance from obstacles increases, so do the values of Δ_{max}, causing a gradual increase in the number of offloading instances, followed by a progressive reduction in energy consumption. For instance, the eager and uniform modes achieve 32% and 66% respective reductions in energy consumption at $r = 13$ m for the (S = 1, N = 1) configuration. Even more so, all configurations of the respective eager and uniform modes at the ($r > 20$m) bracket realize 33% and 67% respective energy gains.

6.5.3 Wireless Channel Variations

In this experiment, we assess how the performance gains of EnergyShield are affected given variations of the wireless channel conditions. Specifically, given potential changes in the channel throughput, ϕ, or the queuing delays, q, we investigate to what extent do the energy savings offered by EnergyShield vary. Additionally, we examine for every set of experimental

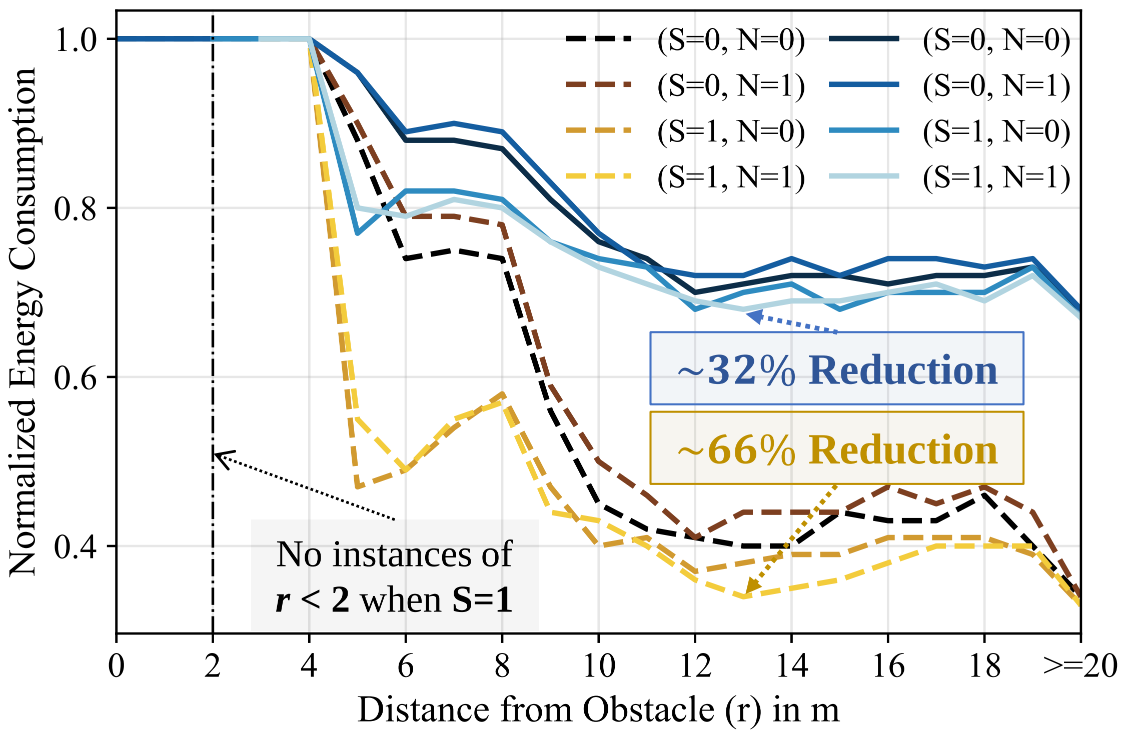


Figure 6.8: Normalized Energy Gains for the eager (*solid*) and uniform (*dashed*) EnergyShield modes with respect to the distance from obstacle (r) in m.

runs what percentage of their total elapsed time windows were extra transition windows needed to complete a single offloading instance, which we denote by the % *Extra Transit Windows* metric. From here, we first analyze such effects when varying $\sigma_\phi \in \{20, 10, 5\}$ Mbps given a fixed $q = 1$ ms, and then when varying $q \in \{10, 20, 50\}$ ms given a fixed $\sigma_\phi = 10$ Mbps. For the uniform EnergyShield, we notice in Figure 6.9 that the % *Extra Transit Windows* drops for the contrasting conditions of high throughput ($\sigma_{phi} = 20$ Mbps) and high queuing delays ($q = 50$ ms), reaching medians of 7% and 8%, respectively. This can be justified in light of how the benign channel conditions ($\sigma_\phi = 20$ Mbps) indicate that the majority of offloading instances are concluded in a single time window with no considerable need for extra transmission windows. Whereas at unfavorable wireless conditions ($q = 50$ ms), $\hat{\Delta}$ values often exceed Δ_{max}, leading EnergyShield to opt for local execution more often so as to avoid wireless uncertainty, lowering the total number of transmission windows alltogether. Such effects are also visible in the twin Figure 6.10 as EnergyShield's energy

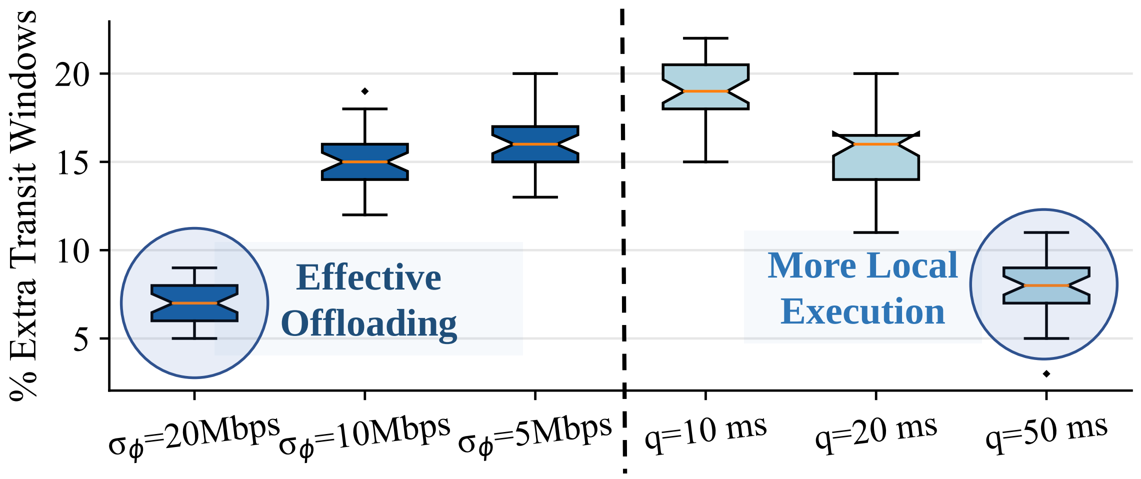


Figure 6.9: Analyzing the % extra transit windows over 35 episodes of uniform EnergyShield given various σ_ϕ and q.

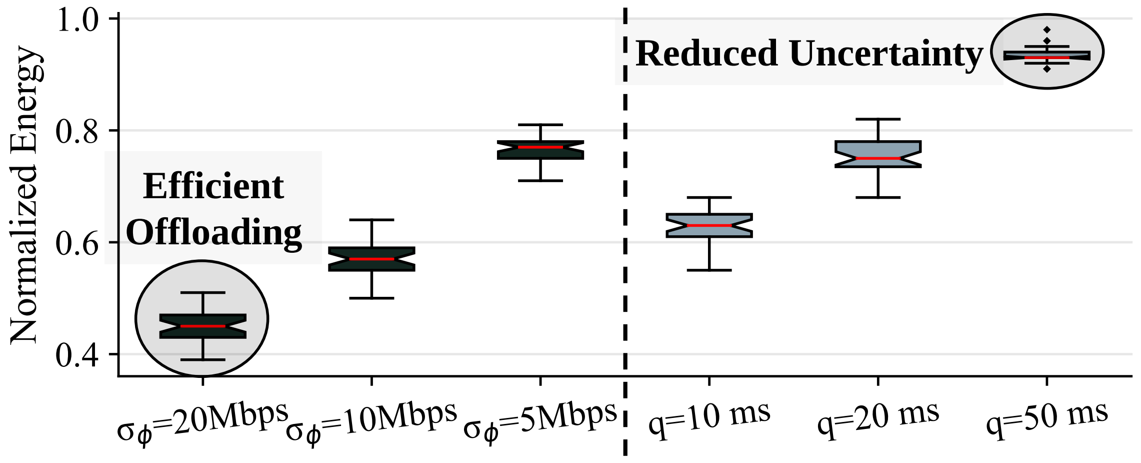


Figure 6.10: Analyzing the Normalized Energy cons. over 35 episodes of uniform EnergyShield given various σ_ϕ and q.

consumption varies across these contrasting conditions, reaching respective medians of 45% and 93% of the local execution energy at σ_ϕ=20 and q=50.

6.5.4 Generality

We train 3 extra RL controllers to evaluate how consistent EnergyShield is with regards to maintaining safety guarantees, and how the energy efficiency gains would vary across diverse driving behaviors. Our detailed results in [139] highlight EnergyShield's effectiveness.

Chapter 7

Conclusion and Future Directions

We have studied different perspectives of the HW/SW co-design space, and proposed various methodologies to enhance the performance and efficiency of AI applications and systems. Below we discuss our overall findings, explain the limitations of our study, and contemplate future research directions along this path.

7.1 Overall Findings and insights.

We go over the key findings and insights we derived from our studies.

Dynamic Neural Networks are efficient, even more so when their design is optimized for the underlying hardware. Through an awareness of the operational context and system state, dynamic neural networks have been shown to elevate efficiency for DNN models operating in the wild. However, when dynamic neural network optimization is conducted alongside the base model design for a target hardware, we find model designs that are more suited for dynamic inference operation over the original static base models as was evident through our evaluation results.

Dynamic search space characterization is promising, but still in the heuristics phase. We have characterized a diverse set of dynamic search spaces depending on the application context and accessible hardware computing units (e.g., HADAS [21] and Map-and-Conquer [23]). Despite their effectiveness, considerable room for improvement remains given the lack of a principled design methodology for dynamic neural networks.

Understanding the synergy of operator and mapping co-optimization can be a fast, cost-efficient solution to elevate efficiency. We have seen this particularly in MaGNAS [137], where through the synergistic construction of a design space entailing GNN operations, the hardware computing units, and pipelining parallelism, we were able to realize operating points with superior performance compared to existing methods. This is another effective approach to consider aside from specialized hardware design.

Speaking of SoCs, heterogeneous integration goes a long way. Beyond their original purposes (e.g., GPU for speedup and NVDLA for low-power) elevating efficiency from different angles), the composition of heterogeneous computing units can lead to superior performance for unfamiliar workloads patterns (e.g., the multi-phase sparse-dense dataflow in GNNs).

Even more so, understanding the workloads affinities is the foundation towards efficient servicing of emerging AI trends like multi-model workloads. Cases where diverse models with different computing affinities as we see in multi-tenancy and AR/VR are only bound to increase in number, diversity, and complexity. Still, the principle remains the same as efficiency can be improved through understanding the different workloads properties down to the operator level, the different forms of parallelism, as well as the heterogeneity and layout of the underlying hardware.

2.5D Multi-chip modules are here to stay and they are getting better. The demonstrated effectiveness of 2.5D multi-chip module architecture in servicing emerging trends of multi-model workloads from the edge to the cloud is a testament to their potential in the semiconductor and domain-specific industries. We can only project bigger pushes for chiplet adoption in both the semiconductor and domain-specific industries (e.g., autonomous driving) [65].

Split-Computing, Bottlenecks, and NAS enhance the effectiveness of edge computing. The combination of these techniques enable on-device machine learning while circumventing potential severe accuracy loss from using simpler or compressed model versions. The adoption of split-computing is dependent on the application context. In autonomous systems, split-computing can be seen as a dynamic, multi-branch neural network distributed across an edge device and server.

DRL Runtime Controllers are effective in capturing spatio-temporal correlation in input data streams to guide offloading decisions. Our hierarchical DRL approach which relies on measured wireless network conditions and abstract data representations has yielded effective offloading policies in multi-sensor autonomous system that outperform rule-based, conventional offloading methods, meeting runtime constraints while improving on energy efficiency.

The vulnerabilities due to split-computing are domain-specific and remain an open area of research. We have demonstrated how split-computing can compromise the existing guarantees on vehicular control safety, and proposed a novel, provably-safe offloading framework that is capable of realizing the benefits of split-computing while upholding formal guarantees on safety. Derivation of such formal guarantees under offloading actions depends on the dynamical model of the autonomous control system, which can differ from one application to another. Another vulnerability is privacy from exposing the end user data to a cloud provider. We have showcased a potential effective approach through inference differential privacy which can aid in preserving user sensitive information.

In closing, we can make the following generic conclusions. (i) Bringing runtime characterization into the design stage can lead to better design points with regards to different metrics. (ii) Incorporating specialized aspects of the application domain context into the design process can yield models more suited for the target application. (iii) HW/SW co-design can be an overloaded term. The idea is to extract the computing layers of the stack which are of interest, before analyzing their impact on the overall system.

7.2 Study Limitations and Future Directions

We discuss the limitations of our studies and future research directions along these topics.

Extension to training scenarios. Our HW/SW co-design methodologies mainly targeted inference workloads. Studying how to apply similar methods for training workloads - be it in a distributed setting, multi-chip module, or heterogeneous SoC - is still an important research direction to be explored.

Multi-application co-location on heterogeneous SoCs. Scenarios where multiple applications are running on the same heterogeneous SoC computing units could aggravate memory traffic and cause additional overhead on the computing application. Implementing a partitioned DNN solutions under process memory contention schemes remains to be investigated.

The design space of 2.5D multi-chip modules. A broader design space exploration process for the multi-chip module acceleration schemes can cover alternative topologies of NoP/NoC connectivity (e.g., Torus), additional forms of parallelism, and various memory hierarchies (e.g., 3D stacked HBMs). Furthermore, additional forms of heterogeneity can be studied, including having digital-in-memory-compute chiplets within the multi-chip module.

Dynamic scheduling of multi-model workloads. We demonstrated a novel characterization of the scheduling search space of multi-model workloads onto heterogeneous chiplet-based architectures. Our scheduling framework can be seen as a form of static scheduling which attempts to identify optimal schedules for recurring workload usage scenarios. From this characterization and the static schedule, investigating how to derive a dynamic scheduler for the multi-model workloads while meeting operational constraints remains an open discussion.

The chiplets supply chain is relatively nascent. This means that the still limited standardization between the semiconductor manufacturers and application-domain vendors leaves room for innovations across different stages of the pipeline. Where novel tools and methods can be implemented to improve the merits of existing chiplet-based design methodologies. Furthermore, domain-specific research, particularly in areas like autonomous driving, can benefit from advancements in chiplet technology.

Adopting HW/SW co-design methods for emerging model classes. We showcased the effectiveness of HW/SW co-design with regards to a representative population of model classes. Still, emerging model architectures require careful consideration and may need the development of specialized frameworks and tools. For instance, language models can use a multitude of the multi-chip modules compared to other models given their unique computational requirements (e.g., KV caching), which means that we can characterize our design space through an additional level of hierarchy that consider the interconnects between different MCMs using high-speed interconnects within and across racks. Furthermore, the application of split-computing for transformer-based architectures remains an interesting research direction.

Split-Computing optimizations in the wild. More research work still needs to be performed to assess the effectiveness of split-computing approaches on real-world autonomous systems in the wild (e.g., UAVs). That way, the impact of motion dynamics and wireless uncertainty can be evaluated in real-world settings.

Split-Computing and Formal Methods. We have derived a provably-safe offloading scheme for neural networks controllers in autonomous driving systems. Similar analysis can be conducted for other control systems at the edge such as autonomous drones which would require characterizing their own dynamical model (which can be more complex considering the additional degrees of freedom) to derive formal safety guarantees on offloading.

Privacy in Edge Computing. Research along privacy-preserving inference direction can investigate implementing novel frameworks that promote the privacy of user data during inference via a context-aware, mutual information based approach, which have been shown to achieve better privacy preservation. Further evaluation is needed for such methods given known adversarial privacy attacks (model inversion, membership inference).

Bibliography

[1] NVIDIA Jetson AGX Xavier Delivers 32 TeraOps for New Era of AI in Robotics. `https://developer.nvidia.com/blog/nvidia-jetson-agx-xavier-32-teraops-ai-robotics/`.

[2] pytorch-cifar. `https://github.com/kuangliu/pytorch-cifar`.

[3] Deaths and mortality - cdc heartdisease facts, 2020.

[4] Efm32 leopard gecko—silicon labs, 2020.

[5] Tesla Autopilot, 2021.

[6] M. Abadi et al. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 308–318, 2016.

[7] U. R. Acharya et al. Automated detection and localization of myocardial infarction using electrocardiogram: a comparative study of different leads. *Knowledge-Based Systems*, 99:146 – 156, 2016.

[8] U. R. Acharya et al. Application of deep convolutional neural network for automated detection of myocardial infarction using ECG signals. *Inf. Sci.*, 415, 2017.

[9] A. K. Akametalu, J. F. Fisac, J. H. Gillula, S. Kaynama, M. N. Zeilinger, and C. J. Tomlin. Reachability-based safe learning with gaussian processes. In *53rd IEEE Conference on Decision and Control*, pages 1424–1431. IEEE, 2014.

[10] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu. Safe reinforcement learning via shielding. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, AAAI'18. AAAI Press, 2018.

[11] P. C. M. Arachchige et al. Local differential privacy for deep learning. *IEEE Internet of Things Journal*, 7(7):5827–5842, 2019.

[12] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans. Mcm-gpu: Multi-chip-module gpus for continued performance scalability. *ACM SIGARCH Computer Architecture News*, 45(2):320–332, 2017.

[13] A. Auten, M. Tomei, and R. Kumar. Hardware acceleration of graph neural networks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[14] S. Baidya, Y.-J. Ku, H. Zhao, J. Zhao, and S. Dey. Vehicular and edge computing for emerging connected and autonomous vehicle applications. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[15] N. Beck, S. White, M. Paraschou, and S. Naffziger. 'zeppelin': An soc for multichip architectures. In *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 40–42. IEEE, 2018.

[16] F. Berkenkamp, M. Turchetta, A. P. Schoellig, and A. Krause. Safe model-based reinforcement learning with stability guarantees. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, page 908–919, NY, USA, 2017. Curran Associates Inc.

[17] G. Bhat et al. An ultra-low energy human activity recognition accelerator for wearable health applications. *ACM Trans. Embed. Comput. Syst.*, 2019.

[18] G. Bhat, N. Tran, H. Shill, and U. Y. Ogras. w-har: An activity recognition dataset and framework using low-power wearable devices. *Sensors*, 2020.

[19] J. Bort-Roig, N. Gilson, A. Puig-Ribera, R. Contreras, and S. Trost. Measuring and influencing physical activity with smartphone technology: A systematic review. *Sports Medicine*, 2014.

[20] R. Bousseljot, D. Kreiseler, and A. Schnabel. Use of the ptb's ecg signal database cardiodat via the internet. 40(s1):317–318, 1995.

[21] H. Bouzidi et al. Hadas: Hardware-aware dynamic neural architecture search for edge performance scaling. *arXiv preprint arXiv:2212.03354*, 2022.

[22] H. Bouzidi et al. HADAS: Hardware-Aware Dynamic Neural Architecture Search for Edge Performance Scaling. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023.

[23] H. Bouzidi, M. Odema, H. Ouarnoughi, S. Niar, and M. A. A. Faruque. Map-and-Conquer: Energy-Efficient Mapping of Dynamic Neural Nets onto Heterogeneous MP-SoCs. In *Proceedings of the 60th ACM/IEEE Design Automation Conference (DAC)*, 2023.

[24] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[25] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han. Once-for-all: Train one network and specialize it for efficient deployment. *arXiv preprint arXiv:1908.09791*, 2019.

[26] H. Cai, L. Zhu, and S. Han. Proxylessnas: Direct neural architecture search on target task and hardware, 2019.

[27] J. Cai, Y. Wei, Z. Wu, S. Peng, and K. Ma. Inter-layer scheduling space definition and exploration for tiled accelerators. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–17, 2023.

[28] J. Cai, Z. Wu, S. Peng, Y. Wei, Z. Tan, G. Shi, M. Gao, and K. Ma. Gemini: Mapping and architecture co-exploration for large-scale dnn chiplet accelerators. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 156–171. IEEE, 2024.

[29] P. Chatarasi, H. Kwon, A. Parashar, M. Pellauer, T. Krishna, and V. Sarkar. Marvel: a data-centric approach for mapping deep learning operators on spatial accelerators. *ACM Transactions on Architecture and Code Optimization (TACO)*, 19(1):1–26, 2021.

[30] C. Chen, K. Li, X. Zou, and Y. Li. Dygnn: Algorithm and architecture support of dynamic pruning for graph neural networks. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1201–1206. IEEE, 2021.

[31] C. Chen and ohers. A survey on graph neural networks and graph transformers in computer vision: A task-oriented perspective. *arXiv preprint arXiv:2209.13232*, 2022.

[32] L. Chen, M. Odema, and M. A. A. Faruque. Romanus: Robust task offloading in modular multi-sensor autonomous driving systems. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pages 1–8, 2022.

[33] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. KDD '16, 2016.

[34] Z. Chen et al. Visformer: The vision-friendly transformer. In *Proc. of the IEEE/CVF international conference on computer vision*, 2021.

[35] R. Cheng, G. Orosz, R. M. Murray, and J. W. Burdick. End-to-end safe reinforcement learning through barrier functions for safety-critical continuous control tasks. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence*, AAAI'19. AAAI Press, 2019.

[36] Y. Chow, O. Nachum, E. Duenez-Guzman, and M. Ghavamzadeh. A lyapunov-based approach to safe reinforcement learning. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, page 8103–8112, NY, USA, 2018. Curran Associates Inc.

[37] M. Courbariaux and Y. Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *arXiv e-prints*, 2016.

[38] P. Cuff and L. Yu. Differential privacy as a mutual information constraint. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 43–54, New York, NY, USA, 2016. Association for Computing Machinery.

[39] M. Cui, S. Zhong, B. Li, X. Chen, and K. Huang. Offloading autonomous driving services via edge computing. *IEEE Internet of Things Journal*, 7(10):10535–10547, 2020.

[40] I. Dagli et al. AxoNN: energy-aware execution of neural network inference on multi-accelerator heterogeneous SoCs. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*, 2022.

[41] C. Dawson, S. Gao, and C. Fan. Safe control with learned certificates: A survey of neural lyapunov, barrier, and contraction methods for robotics and control. *IEEE Transactions on Robotics*, pages 1–19, 2023.

[42] B. U. Demirel, I. A. Bayoumy, and M. A. A. Faruque. Energy-efficient real-time heart monitoring on edge–fog–cloud internet of medical things. *IEEE Internet of Things Journal*, 9(14):12472–12481, 2022.

[43] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[44] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun. Carla: An open urban driving simulator. In *Conference on robot learning*, pages 1–16. PMLR, 2017.

[45] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. Shidiannao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 92–104, 2015.

[46] S. Dutta, S. Jha, S. Sankaranarayanan, and A. Tiwari. Output range analysis for deep feedforward neural networks. In A. Dutle, C. Muñoz, and A. Narkawicz, editors, *NASA Formal Methods*, pages 121–138, Cham, 2018. Springer.

[47] C. Dwork. A firm foundation for private data analysis. *Communications of the ACM*, 54(1):86–95, 2011.

[48] C. Dwork et al. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3-4):211–407, 2014.

[49] A. E. Eshratifar, A. Esmaili, and M. Pedram. BottleNet: A Deep Learning Architecture for Intelligent Mobile Cloud Computing Services. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6, 2019.

[50] M. Everingham et al. The pascal visual object classes (VOC) challenge. *International journal of computer vision*, 88(2):303–338, 2010.

[51] FacebookResearch. Hrvit-b1. `https://github.com/facebookresearch/HRViT/blob/main/models/hrvit.py#L1125-L1155`, 2022.

[52] M. Farhadi, M. Ghasemi, and Y. Yang. A novel design of adaptive and hierarchical convolutional neural networks using partial reconfiguration on fpga. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2019.

[53] M. Fazlyab, A. Robey, H. Hassani, M. Morari, and G. J. Pappas. *Efficient and Accurate Estimation of Lipschitz Constants for Deep Neural Networks*. Curran Associates Inc., NY, USA, 2019.

[54] J. Feng, Z. Liu, C. Wu, and Y. Ji. Mobile edge computing for the internet of vehicles: Offloading framework and job scheduling. *IEEE vehicular technology magazine*, 14(1):28–36, 2018.

[55] J. Ferlez, M. Elnaggar, Y. Shoukry, and C. Fleming. ShieldNN: A provably safe NN filter for unsafe NN controllers. *arXiv preprint arXiv:2006.09564*, 2020.

[56] J. F. Fisac, A. K. Akametalu, M. N. Zeilinger, S. Kaynama, J. Gillula, and C. J. Tomlin. A general safety framework for learning-based control in uncertain robotic systems. *IEEE Transactions on Automatic Control*, 64(7):2737–2752, 2019.

[57] M. Fredrikson, S. Jha, and T. Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1322–1333, New York, NY, USA, 2015. Association for Computing Machinery.

[58] M. Gao, X. Yang, J. Pu, M. Horowitz, and C. Kozyrakis. Tangram: Optimized coarse-grained dataflow for scalable nn accelerators. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 807–820, 2019.

[59] Y. Gao et al. Graph neural architecture search. In *IJCAI*, volume 20, pages 1403–1409, 2020.

[60] Y. Gao, H. Yang, P. Zhang, C. Zhou, and Y. Hu. Graph neural architecture search. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 1403–1409, 2020.

[61] R. Garg et al. Understanding the design-space of sparse/dense multiphase gnn dataflows on spatial accelerators. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 571–582. IEEE, 2022.

[62] L. Ge, Z. Ren, Y. Li, Z. Xue, Y. Wang, J. Cai, and J. Yuan. 3d hand shape and pose estimation from a single rgb image. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10833–10842, 2019.

[63] S. Ghodrati, B. H. Ahn, J. Kyung Kim, S. Kinzer, B. R. Yatham, N. Alla, H. Sharma, M. Alian, E. Ebrahimi, N. S. Kim, C. Young, and H. Esmaeilzadeh. 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.

[64] V. Govindarajan, K. Driggs-Campbell, and R. Bajcsy. Data-driven reachability analysis for human-in-the-loop systems. In *2017 IEEE Conference on Decision and Control (CDC)*, pages 2617–2622, 2017.

[65] B. C. Group. The future of automotive compute. `https://media-publications.bcg.com/The-Future-of-Automotive-Compute.pdf`, 2023.

[66] R. Hadidi et al. Toward collaborative inferencing of deep neural networks on internet-of-things devices. *IEEE Internet of Things Journal*, 7(6):4950–4960, 2020.

[67] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[68] K. Han, Y. Wang, J. Guo, Y. Tang, and E. Wu. Vision GNN: An image is worth graph of nodes. In *Advances in Neural Information Processing Systems*, 2022.

[69] Y. Han, G. Huang, S. Song, L. Yang, H. Wang, and Y. Wang. Dynamic neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[70] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, et al. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629. IEEE, 2018.

[71] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition, 2015.

[72] 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 (CVPR)*, pages 770–778, 2016.

[73] Y. He, X. Zhang, and J. Sun. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[74] K. Hegde, P.-A. Tsai, S. Huang, V. Chandra, A. Parashar, and C. W. Fletcher. Mind mappings: enabling efficient algorithm-accelerator mapping space search. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 943–958, 2021.

[75] C. Hong, Q. Huang, G. Dinh, M. Subedar, and Y. S. Shao. Dosa: Differentiable model-based one-loop search for dnn accelerators. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 209–224, 2023.

[76] X. Hou et al. Distredge: Speeding up convolutional neural network inference on distributed edge devices. In *IPDPS*. IEEE, 2022.

[77] J. Huang et al. A close examination of performance and power characteristics of 4G LTE networks. In *Proc. of Intl. Conf. on Mobile Systems, Applications, and Services*, MobiSys '12, page 225–238, 2012.

[78] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 225–238, 2012.

[79] Q. Huang, M. Kang, G. Dinh, T. Norell, A. Kalaiah, J. Demmel, J. Wawrzynek, and Y. S. Shao. Cosa: Scheduling by constrained optimization for spatial accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 554–566. IEEE, 2021.

[80] R. Hwang, T. Kim, Y. Kwon, and M. Rhu. Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 968–981. IEEE, 2020.

[81] R. Ivanov, J. Weimer, R. Alur, G. J. Pappas, and I. Lee. Verisig: Verifying safety properties of hybrid systems with neural network controllers. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '19, page 169–178, New York, NY, USA, 2019. Association for Computing Machinery.

[82] E. Jeong et al. Tensorrt-based framework and optimization methodology for deep learning inference on jetson boards. *ACM Transactions on Embedded Computing Systems (TECS)*, 2022.

[83] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[84] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1651–1669. USENIX Association, 2018.

[85] P. Kairouz, S. Oh, and P. Viswanath. The composition theorem for differential privacy. In *International conference on machine learning*, pages 1376–1385. PMLR, 2015.

[86] K. Kandasamy et al. Tuning hyperparameters without grad students: Scalable and robust bayesian optimisation with dragonfly. *Journal of Machine Learning Research*, 21(81):1–27, 2020.

[87] D. Kang et al. Scheduling of deep learning applications onto heterogeneous processors in an embedded device. *IEEE Access*, 8, 2020.

[88] Y. Kang et al. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. Association for Computing Machinery, 2017.

[89] A. Kannan, N. E. Jerger, and G. H. Loh. Enabling interposer-based disintegration of multi-core processors. In *Proceedings of the 48th international symposium on Microarchitecture*, pages 546–558, 2015.

[90] S.-C. Kao et al. Gamma: Automating the hw mapping of dnn models on accelerators via genetic algorithm. In *ICCAD*. IEEE, 2020.

[91] S.-C. Kao, G. Jeong, and T. Krishna. Confuciux: Autonomous hardware resource assignment for dnn accelerators using reinforcement learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 622–636. IEEE, 2020.

[92] S.-C. Kao and T. Krishna. Magma: An optimization framework for mapping multiple dnns on multiple accelerator cores. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 814–830. IEEE, 2022.

[93] J. Kim and S. Ha. Energy-aware scenario-based mapping of deep learning applications onto heterogeneous processors under real-time constraints. *IEEE Transactions on Computers*, 2022.

[94] S. Kim, H. Genc, V. V. Nikiforov, K. Asanović, B. Nikolić, and Y. S. Shao. Moca: Memory-centric, adaptive execution for multi-tenant deep neural networks. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 828–841. IEEE, 2023.

[95] K. Kiningham et al. Grip: A graph neural network accelerator architecture. *IEEE Transactions on Computers*, 2022.

[96] T. Koller, F. Berkenkamp, M. Turchetta, and A. Krause. Learning-based model predictive control for safe exploration. In *2018 IEEE Conference on Decision and Control (CDC)*, pages 6059–6066, 2018.

[97] J. Kong, M. Pfeiffer, G. Schildbach, and F. Borrelli. Kinematic and dynamic vehicle models for autonomous driving control design. In *2015 IEEE Intelligent Vehicles Symposium (IV)*, pages 1094–1099, 2015.

[98] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna. Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 754–768, 2019.

[99] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A. Parashar. Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings. *IEEE micro*, 40(3):20–29, 2020.

[100] H. Kwon, L. Lai, M. Pellauer, T. Krishna, Y.-H. Chen, and V. Chandra. Heterogeneous dataflow accelerators for multi-dnn workloads. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 71–83. IEEE, 2021.

[101] H. Kwon, K. Nair, J. Seo, J. Yik, D. Mohapatra, D. Zhan, J. Song, P. Capak, P. Zhang, P. Vajda, et al. Xrbench: An extended reality (xr) machine learning benchmark suite for the metaverse. *Proceedings of Machine Learning and Systems*, 5, 2023.

[102] L. Landrieu and M. Simonovsky. Large-scale point cloud semantic segmentation with superpoint graphs. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4558–4567, 2018.

[103] S. Laskaridis, S. I. Venieris, H. Kim, and N. D. Lane. Hapi: Hardware-aware progressive inference. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2020.

[104] B. Li, T. Patel, S. Samsi, V. Gadepally, and D. Tiwari. Miso: exploiting multi-instance gpu capability on multi-tenant gpu clusters. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 173–189, 2022.

[105] C. Li, J. Peng, L. Yuan, G. Wang, X. Liang, L. Lin, and X. Chang. Block-wisely supervised neural architecture search with knowledge distillation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1989–1998, 2020.

[106] G. Li, M. Muller, A. Thabet, and B. Ghanem. Deepgcns: Can gcns go as deep as cnns? In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 9267–9276, 2019.

[107] M. Li et al. Privynet: A flexible framework for privacy-preserving deep neural network training. *arXiv preprint arXiv:1709.06161*, 2017.

[108] X. Li, C. Lou, Z. Zhu, Y. Chen, Y. Shen, Y. Ma, and A. Zou. Predictive exit: Prediction of fine-grained early exits for computation-and energy-efficient inference. *arXiv preprint arXiv:2206.04685*, 2022.

[109] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars. The architectural implications of autonomous driving: Constraints and acceleration. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 751–766, 2018.

[110] C. Liu, T. Arnon, C. Lazarus, C. Barrett, and M. J. Kochenderfer. Algorithms for verifying deep neural networks. *arXiv preprint arXiv:1903.06758*, 2019.

[111] C. Liu, K. Kim, J. Gu, Y. Furukawa, and J. Kautz. Planercnn: 3d plane detection and reconstruction from a single image. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4450–4459, 2019.

[112] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi. Edge computing for autonomous driving: Opportunities and challenges. *Proceedings of the IEEE*, 107(8):1697–1716, 2019.

[113] S. Liu, J. Tang, Z. Zhang, and J.-L. Gaudiot. Computer architectures for autonomous driving. *Computer*, 50(8):18–25, 2017.

[114] Z. Liu, J. Leng, Z. Zhang, Q. Chen, C. Li, and M. Guo. Veltair: towards high-performance multi-tenant deep learning services via adaptive compilation and scheduling. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 388–401, 2022.

[115] C. Llanes, M. Abate, and S. Coogan. Safety from fast, in-the-loop reachability with application to uavs. In *2022 ACM/IEEE 13th International Conference on Cyber-Physical Systems (ICCPS)*, pages 127–136, 2022.

[116] L. Lu, N. Guan, Y. Wang, L. Jia, Z. Luo, J. Yin, J. Cong, and Y. Liang. Tenet: A framework for modeling tensor dataflow based on relation-centric notation. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 720–733. IEEE, 2021.

[117] F. Ma and S. Karaman. Sparse-to-dense: Depth prediction from sparse depth samples and a single image. In *2018 IEEE international conference on robotics and automation (ICRA)*, pages 4796–4803. IEEE, 2018.

[118] X. Ma, C. Si, Y. Wang, C. Liu, and L. Zhang. Nasa: accelerating neural network design with a nas processor. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 790–803. IEEE, 2021.

[119] A. Malawade, M. Odema, S. Lajeunesse-DeGroot, and M. A. Al Faruque. Sage: A split-architecture methodology for efficient end-to-end autonomous vehicle control. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5s):1–22, 2021.

[120] A. V. Malawade, T. Mortlock, and M. A. Al Faruque. Hydrafusion: Context-aware selective sensor fusion for robust and efficient autonomous vehicle perception. In *2022 ACM/IEEE 13th International Conference on Cyber-Physical Systems (ICCPS)*, pages 68–79. IEEE, 2022.

[121] A. V. Malawade, S.-Y. Yu, B. Hsu, D. Muthirayan, P. P. Khargonekar, and M. A. Al Faruque. Spatiotemporal scene-graph embedding for autonomous vehicle collision prediction. *IEEE Internet of Things Journal*, 9(12):9379–9388, 2022.

[122] Y. Matsubara, S. Baidya, D. Callegaro, M. Levorato, and S. Singh. Distilled split deep neural networks for edge-assisted real-time systems. In *Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges*, pages 21–26, 2019.

[123] Y. Matsubara, D. Callegaro, S. Baidya, M. Levorato, and S. Singh. Head network distillation: Splitting distilled deep neural networks for resource-constrained edge computing systems. *IEEE Access*, 8:212177–212193, 2020.

[124] Meta. D2go. `https://github.com/facebookresearch/d2go`, 2022.

[125] Microsoft. Announcing microsoft copilot, your everyday ai companion. `https://blogs.microsoft.com/blog/2023/09/21/announcing-microsoft-copilot-your-everyday-ai-companion/`, 2023.

[126] Microsoft. Azure openai service. `https://azure.microsoft.com/en-us/products/ai-services/openai-service`, 2023.

[127] F. Mireshghallah et al. Shredder: Learning noise distributions to protect inference privacy. In *Proceedings of the 25th Intl. Conf. on Arch. Support for Programming Languages and Operating Systems (ASPLOS'20)*, 2020.

[128] F. Mireshghallah et al. Not all features are equal: Discovering essential features for preserving prediction privacy. In *Proceedings of the Web Conference 2021*, WWW '21, page 669–680, 2021.

[129] MLCommons. Mlperf inference. `https://mlcommons.org/benchmarks/inference-datacenter/`, 2023.

[130] A. Mohan, S. Sripad, P. Vaishnav, and V. Viswanathan. Trade-offs between automation and light vehicle electrification. *Nature Energy*, 5(7):543–549, 2020.

[131] P. Molchanov et al. Importance estimation for neural network pruning. In *CVPR*, 2019.

[132] M. A. H. Monil et al. Mephesto: Modeling energy-performance in heterogeneous socs and their trade-offs. In *PACT*, pages 413–425, 2020.

[133] S. Naffziger, N. Beck, T. Burd, K. Lepak, G. H. Loh, M. Subramony, and S. White. Pioneering chiplet technology and design for the amd epyc™ and ryzen™ processor families: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 57–70. IEEE, 2021.

[134] A. Newcomb. Facebook data harvesting scandal widens to 87 million people, 2018.

[135] NVIDIA. Nvdla deep learning accelerator. `http://nvdla.org,2017.`, 2023.

[136] M. Odema and M. A. Al Faruque. Privynas: Privacy-aware neural architecture search for split computing in edge-cloud systems. *IEEE Internet of Things Journal*, 2023.

[137] M. Odema, H. Bouzidi, H. Ouarnoughi, S. Niar, and M. A. Al Faruque. Magnas: A mapping-aware graph neural architecture search framework for heterogeneous mpsoc deployment. *ACM Transactions on Embedded Computing Systems*, 22(5s):1–26, 2023.

[138] M. Odema, L. Chen, M. Levorato, and M. A. A. Faruque. Testudo: Collaborative intelligence for latency-critical autonomous systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2022.

[139] M. Odema, J. Ferlez, G. Vaisi, Y. Shoukry, and M. A. Al Faruque. Energyshield: Provably-safe offloading of neural network controllers for energy efficiency. In *Proceedings of the ACM/IEEE 14th International Conference on Cyber-Physical Systems (with CPS-IoT Week 2023)*, pages 187–198, 2023.

[140] M. Odema, N. Rashid, and M. A. Al Faruque. Eexnas: Early-exit neural architecture search solutions for low-power wearable devices. In *2021 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6. IEEE, 2021.

[141] M. Odema, N. Rashid, and M. A. Al Faruque. Energy-aware design methodology for myocardial infarction detection on low-power wearable devices. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, pages 621–626, 2021.

[142] M. Odema, N. Rashid, B. U. Demirel, and M. A. Al Faruque. LENS: Layer Distribution Enabled Neural Architecture Search in Edge-Cloud Hierarchies. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 403–408. IEEE, 2021.

[143] M. Orenes-Vera, E. Tureci, D. Wentzlaf, and M. Martonosi. Massive data-centric parallelism in the chiplet era. *arXiv preprint arXiv:2304.09389*, 2023.

[144] P. Panda, A. Sengupta, and K. Roy. Conditional deep learning for energy-efficient and enhanced pattern recognition. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 475–480. IEEE, 2016.

[145] P. Panda, A. Sengupta, and K. Roy. Conditional deep learning for energy-efficient and enhanced pattern recognition. In *DATE*, 2016.

[146] N. Papernot, M. Abadi, Úlfar Erlingsson, I. Goodfellow, and K. Talwar. Semi-supervised knowledge transfer for deep learning from private training data, 2017.

[147] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 304–315. IEEE, 2019.

[148] J. Park, M. Naumov, P. Basu, S. Deng, A. Kalaiah, D. Khudia, J. Law, P. Malani, A. Malevich, S. Nadathur, et al. Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv preprint arXiv:1811.09886*, 2018.

[149] D. Paul, J. Singh, and J. Mathew. Hardware-software co-design approach for deep learning inference. In *2019 7th International Conference on Smart Computing & Communications (ICSCC)*, pages 1–5. IEEE, 2019.

[150] M. Phuong and C. H. Lampert. Distillation-based training for multi-exit architectures. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[151] R. Pujol, H. Tabani, L. Kosmidis, E. Mezzetti, J. Abella Ferrer, and F. J. Cazorla. Generating and exploiting deep learning variants to increase heterogeneous resource utilization in the nvidia xavier. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 23, 2019.

[152] Qualcomm. Quacomm hexagon 680. `https://www.hotchips.org/wp-content/uploads/hc_archives/hc27/HC27.24-Monday-Epub/HC27.24.20-Multimedia-Epub/HC27.24.211-Hexagon680-Codrescu-Qualcomm.pdf`, 2015.

[153] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[154] R. Ranftl, K. Lasinger, D. Hafner, K. Schindler, and V. Koltun. Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer. *IEEE transactions on pattern analysis and machine intelligence*, 44(3):1623–1637, 2020.

[155] Y. Rao et al. Dynamicvit: Efficient vision transformers with dynamic token sparsification. *NeurIPS*, 34, 2021.

[156] N. Rashid and M. A. Al Faruque. Energy-efficient real-time myocardial infarction detection on wearable devices. In *EMBC*, 2020.

[157] N. Rashid, M. Dautta, P. Tseng, and M. A. A. Faruque. Hear: Fog-enabled energy aware online human eating activity recognition. *IEEE Internet of Things Journal, doi: 10.1109/JIOT.2020.3008842.*, 2020.

[158] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, et al. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459. IEEE, 2020.

[159] S. Ren, K. He, R. Girshick, and J. Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28, 2015.

[160] A. Robey, H. Hu, L. Lindemann, H. Zhang, D. V. Dimarogonas, S. Tu, and N. Matni. Learning control barrier functions from expert demonstrations. In *2020 59th IEEE Conference on Decision and Control (CDC)*, pages 3717–3724, 2020.

[161] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015. Springer International Publishing.

[162] M. Sandler et al. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

[163] K. Sasaki, N. Suzuki, S. Makido, and A. Nakao. Vehicle control system coordinated between cloud and mobile edge computing. In *2016 55th Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE)*, pages 1122–1127. IEEE, 2016.

[164] B. Shahian Jahromi, T. Tulabandhula, and S. Cetin. Real-time hybrid multi-sensor fusion framework for perception in autonomous vehicles. *Sensors*, 19(20):4357, 2019.

[165] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

[166] K. Shang, H. Ishibuchi, L. He, and L. M. Pang. A survey on the hypervolume indicator in evolutionary multiobjective optimization. *IEEE Transactions on Evolutionary Computation*, 25(1):1–20, 2020.

[167] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, et al. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 14–27, 2019.

[168] M. Sheeny et al. Radiate: A radar dataset for automotive perception in bad weather. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–7. IEEE, 2021.

[169] G. Shi, X. Shi, M. O'Connell, R. Yu, K. Azizzadenesheli, A. Anandkumar, Y. Yue, and S.-J. Chung. Neural lander: Stable drone landing control using learned dynamics. In *2019 International Conference on Robotics and Automation (ICRA)*, page 9784–9790. IEEE Press, 2019.

[170] M. Shi, Y. Tang, X. Zhu, Y. Huang, D. Wilson, Y. Zhuang, and J. Liu. Genetic-gnn: evolutionary architecture search for graph neural networks. *Knowledge-Based Systems*, 247:108752, 2022.

[171] Y. Shi, Y. Wang, C. Wu, C.-F. Yeh, J. Chan, F. Zhang, D. Le, and M. Seltzer. Emformer: Efficient memory transformer based acoustic model for low latency streaming speech recognition. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6783–6787. IEEE, 2021.

[172] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In Y. Bengio et al., editors, *ICLR*, 2015.

[173] Y. Song et al. Sara: Self-aware resource allocation for heterogeneous mpsocs. In *DAC*, 2018.

[174] D. Sopic, A. Aminifar, A. Aminifar, and D. Atienza. Real-time classification technique for early detection and prevention of myocardial infarction on wearable devices. In *BioCAS*, 2017.

[175] D. Sopic, A. Aminifar, A. Aminifar, and D. Atienza. Real-time event-driven classification technique for early detection and prevention of myocardial infarction on wearable systems. *IEEE Trans. Biomed. Circuits and Systems*, 12(5), 2018.

[176] M. Srinivasan, A. Dabholkar, S. Coogan, and P. A. Vela. Synthesis of control barrier functions using a supervised machine learning approach. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7139–7145, 2020.

[177] J. R. Stevens et al. Gnnerator: A hardware/software framework for accelerating graph neural networks. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 955–960. IEEE, 2021.

[178] X. Sun, H. Khedr, and Y. Shoukry. Formal verification of neural network controlled autonomous systems. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '19, page 147–156, New York, NY, USA, 2019. Association for Computing Machinery.

[179] 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.

[180] E. Talpes, D. D. Sarma, G. Venkataramanan, P. Bannon, B. McGee, B. Floering, A. Jalote, C. Hsiong, S. Arora, A. Gorti, et al. Compute solution for tesla's full self-driving computer. *IEEE Micro*, 40(2):25–35, 2020.

[181] E. Talpes, D. D. Sarma, D. Williams, S. Arora, T. Kunjan, B. Floering, A. Jalote, C. Hsiong, C. Poorna, V. Samant, et al. The microarchitecture of dojo, tesla's exascale computer. *IEEE Micro*, 2023.

[182] T. Tambe, C. Hooper, L. Pentecost, T. Jia, E.-Y. Yang, M. Donato, V. Sanh, P. Whatmough, A. M. Rush, D. Brooks, et al. Edgebert: Sentence-level energy optimizations for latency-aware multi-task nlp inference. In *Micro-54*, pages 830–844, 2021.

[183] Z. Tan, H. Cai, R. Dong, and K. Ma. Nn-baton: Dnn workload orchestration and chiplet granularity exploration for multichip accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1013–1026. IEEE, 2021.

[184] S. Tang, B. Chen, H. Iwen, J. Hirsch, S. Fu, Q. Yang, P. Palacharla, N. Wang, X. Wang, and W. Shi. Vecframe: A vehicular edge computing framework for connected autonomous vehicles. In *2021 IEEE International Conference on Edge Computing (EDGE)*, pages 68–77, 2021.

[185] A. J. Taylor, A. Singletary, Y. Yue, and A. D. Ames. A control barrier perspective on episodic learning via projection-to-state safety. *IEEE Control Systems Letters*, 5(3):1019–1024, 2021.

[186] S. Teerapittayanon, B. McDanel, and H.-T. Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *23rd International Conference on Pattern Recognition (ICPR)*, pages 2464–2469. IEEE, 2016.

[187] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[188] 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.

[189] K. Vatanparvar and M. A. Al Faruque. Battery lifetime-aware automotive climate control for electric vehicles. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6, 2015.

[190] K. Vatanparvar and M. A. Al Faruque. Energy management as a service over fog computing platform. In *Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems*, ICCPS '15, page 248–249, New York, NY, USA, 2015. Association for Computing Machinery.

[191] K. Vatanparvar and M. A. Al Faruque. Eco-friendly automotive climate control and navigation system for electric vehicles. In *2016 ACM/IEEE 7th International Conference on Cyber-Physical Systems (ICCPS)*, pages 1–10. IEEE, 2016.

[192] K. Vatanparvar, J. Wan, and M. A. Al Faruque. Battery-aware energy-optimal electric vehicle driving management. In *2015 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 353–358. IEEE, 2015.

[193] P. Vivet, E. Guthmuller, Y. Thonnart, G. Pillonnet, G. Moritz, I. Miro-Panadès, C. Fuguet, J. Durupt, C. Bernard, D. Varreau, J. Pontes, S. Thuries, D. Coriat, M. Harrand, D. Dutoit, D. Lattard, L. Arnaud, J. Charbonnier, P. Coudrain, A. Garnier, F. Berger, A. Gueugnot, A. Greiner, Q. Meunier, A. Farcy, A. Arriordaz, S. Cheramy, and F. Clermidy. 2.3 a 220gops 96-core processor with 6 chiplets 3d-stacked on an active interposer offering 0.6 ns/mm latency, 3tb/s/mm 2 inter-chiplet interconnects and 156mw/mm 2@ 82%-peak-efficiency dc-dc converters. In *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 46–48. IEEE, 2020.

[194] D. Wang, M. Li, C. Gong, and V. Chandra. Attentivenas: Improving neural architecture search via attentive sampling. *arXiv preprint arXiv:2011.09011*, 2020.

[195] H. Wang, B. Kim, J. Xie, and Z. Han. E-auto: A communication scheme for connected vehicles with edge-assisted autonomous driving. In *2019 IEEE International Conference on Communications (ICC)*. IEEE, 2019.

[196] J. Wang et al. Not just privacy: Improving performance of private deep learning in mobile cloud. In *Proceedings of the 24th ACM SIGKDD Intl. Conf. on Knowledge Discovery & Data Mining*, pages 2407–2416, 2018.

[197] L. Wang, E. A. Theodorou, and M. Egerstedt. Safe learning of quadrotor dynamics using barrier certificates. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2460–2465, 2018.

[198] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon. Dynamic graph cnn for learning on point clouds. *Acm Transactions On Graphics (tog)*, 38(5):1–12, 2019.

[199] Z. Wang, G. R. Nair, G. Krishnan, S. K. Mandal, N. Cherian, J.-S. Seo, C. Chakrabarti, U. Y. Ogras, and Y. Cao. Ai computing in light of 2.5 d interconnect roadmap: Big-little chiplets for in-memory acceleration. In *2022 International Electron Devices Meeting (IEDM)*, pages 23–6. IEEE, 2022.

[200] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

[201] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE international symposium on high performance computer architecture (HPCA)*, pages 331–344. IEEE, 2019.

[202] Q. Wu, H. Liu, R. Wang, P. Fan, Q. Fan, and Z. Li. Delay-sensitive task offloading in the 802.11 p-based vehicular fog computing systems. *IEEE Internet of Things Journal*, 7(1):773–785, 2019.

[203] Y. N. Wu, P.-A. Tsai, A. Parashar, V. Sze, and J. S. Emer. Sparseloop: An analytical approach to sparse tensor accelerator modeling. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1377–1395. IEEE, 2022.

[204] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.

[205] W. Xiang, D. M. Lopez, P. Musau, and T. T. Johnson. *Reachable Set Estimation and Verification for Neural Network Models of Nonlinear Dynamic Systems*, pages 123–144. Springer, Cham, 2019.

[206] Q. Xiao, S. Zheng, B. Wu, P. Xu, X. Qian, and Y. Liang. Hasco: Towards agile hardware and software co-design for tensor computation. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1055–1068. IEEE, 2021.

[207] W. Xiao, N. Mehdipour, A. Collin, A. Y. Bin-Nun, E. Frazzoli, R. D. Tebbens, and C. Belta. Rule-based optimal control for autonomous driving. ICCPS '21, page 143–154, New York, NY, USA, 2021. Association for Computing Machinery.

[208] D. Xu, D. Anguelov, and A. Jain. Pointfusion: Deep sensor fusion for 3D bounding box estimation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 244–253, 2018.

[209] D. Xu, Y. Zhu, C. B. Choy, and L. Fei-Fei. Scene graph generation by iterative message passing. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5410–5419, 2017.

[210] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019.

[211] X. Xu, P. Tabuada, J. W. Grizzle, and A. D. Ames. Robustness of control barrier functions for safety critical control. *IFAC-PapersOnLine*, 48(27):54–61, 2015. Analysis and Design of Hybrid Systems ADHS.

[212] Y. Xu et al. Pccs: Processor-centric contention-aware slowdown model for heterogeneous system-on-chips. In *MICRO*, 2021.

[213] L. Xun et al. Optimising resource management for embedded machine learning. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1556–1561. IEEE, 2020.

[214] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie. Hygcn: A gcn accelerator with hybrid architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–29. IEEE, 2020.

[215] S. Yan, Y. Xiong, and D. Lin. Spatial temporal graph convolutional networks for skeleton-based action recognition. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[216] J. Yang, J. Lu, S. Lee, D. Batra, and D. Parikh. Graph r-cnn for scene graph generation. In *Proceedings of the European conference on computer vision (ECCV)*, pages 670–685, 2018.

[217] J. H. Yoo et al. 3d-cvf: Generating joint camera and lidar features using cross-view spatial feature fusion for 3d object detection. In *Computer Vision–ECCV 2020: Proceedings, Part XXVII 16*, 2020.

[218] H. You et al. Gcod: Graph convolutional network acceleration via dedicated algorithm and accelerator co-design. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 460–474. IEEE, 2022.

[219] H. You, C. Wan, Y. Zhao, Z. Yu, Y. Fu, J. Yuan, S. Wu, S. Zhang, Y. Zhang, C. Li, et al. Eyecod: eye tracking system acceleration via flatcam-based algorithm & accelerator co-design. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 610–622, 2022.

[220] J. You, Z. Ying, and J. Leskovec. Design space for graph neural networks. *Advances in Neural Information Processing Systems*, 33:17009–17021, 2020.

[221] J. Yu et al. Slimmable neural networks. In *International Conference on Learning Representations*, 2019.

[222] J. Yu et al. Bignas: Scaling up neural architecture search with big single-stage models. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part VII 16*. Springer, 2020.

[223] S.-Y. Yu, A. V. Malawade, D. Muthirayan, P. P. Khargonekar, and M. A. Al Faruque. Scene-graph augmented data-driven risk assessment of autonomous vehicle decisions. *IEEE Transactions on Intelligent Transportation Systems*, 23(7):7941–7951, 2021.

[224] Z. Yu et al. Mia-former: Efficient and robust vision transformers via multi-grained input-adaptation. In *AAAI*, volume 36, 2022.

[225] Z. Yuan, B. Wu, G. Sun, Z. Liang, S. Zhao, and W. Bi. S2dnas: Transforming static cnn model for dynamic inference via neural architecture search. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part II 16*, pages 175–192. Springer, 2020.

[226] K. Zhang, Y. Mao, S. Leng, Q. Zhao, L. Li, X. Peng, L. Pan, S. Maharjan, and Y. Zhang. Energy-efficient offloading for mobile edge computing in 5g heterogeneous networks. *IEEE access*, 4:5896–5907, 2016.

[227] X. Zhang, A. Zhang, J. Sun, X. Zhu, Y. E. Guo, F. Qian, and Z. M. Mao. Emp: edge-assisted multi-vehicle perception. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 545–558, 2021.

[228] Y. Zhang et al. G-cos: Gnn-accelerator co-search towards both better accuracy and efficiency. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2021.

[229] S. Zheng, X. Zhang, L. Liu, S. Wei, and S. Yin. Atomic dataflow based graph-level workload orchestration for scalable dnn accelerators. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 475–489. IEEE, 2022.

[230] A. Zhou et al. Hardware-aware graph neural network automated design for edge computing platforms. In *Proceedings of the 60th ACM/IEEE Design Automation Conference (DAC)*, 2023.

[231] K. Zhou et al. Auto-gnn: Neural architecture search of graph neural networks. *Frontiers in big Data*, 2022.

[232] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

Appendix A

Privacy-aware Neural Architecture Search for Split Computing

A.1 Overview

This appendix discusses the inference privacy concerns from split-computing as data is exposed to cloud servers that can cause sensitive user information leakage, and how to incorporate privacy awareness in DNN model design. The full study details are listed in [136].

A.2 Introduction

the transmission of user's data to the cloud poses privacy concerns as users have no control over how the data is used once it has been made available to the provider [190]. Previous incidents have already seen providers sharing users' personal information with other third parties (e.g., Facebook incident in 2018 [134]). Another study has demonstrated that aggregate user data at the cloud which are reused for model training are liable to model inversion attacks, mainly due to the model weights leaking users' sensitive attributes (as was demonstrated in a model inversion attack recovering images from a facial recognition system [57]).

Addressing this, researchers have proposed numerous techniques to provide privacy guarantees under such remote inference model [84, 127, 128, 196]. Despite their effectiveness, these methods were applied to models originally targeted for *single* platform deployment, that is, models whose design was not optimized for edge-cloud system operation. As works in [6, 107] have observed that varying a subset of architectural parameters can impact privacy guarantees, the inherent privacy-preserving capabilities of a model can be affected by the choice of architectural parameters. Hence, an argument can be made that architectural parameter choices can be optimized to enhance a DNN model's privacy preserving capabilities, giving rise to the following questions from a DNN model designer's perspective:

- How to assess candidate model architectural designs with regards to upholding inference privacy guarantees given a remote inference operational scheme?
- How to model the relationship between architectural design choices and the inherent privacy-preserving capabilities given a split computing model of computation?
- How to implement a design framework for non-monolithic DNN models balancing the underlying accuracy-performance-privacy trade-offs?

A.2.1 Novel Contributions

In this work, we study the value of incorporating privacy as a design metric given a remote inference deployment scheme. To achieve this, a formal metric needs to be utilized for to quantify privacy as a design objective such as the rigorous Differential Privacy (DP) standard [47, 48] with its quantifiable privacy loss budget ϵ. In DP, calculated noise is added to user's data from Gaussian/Laplacian distributions for obfuscation and minimizing its likelihood of revealing sensitive information before transmission to the cloud. From here, we summarize our contributions in this study as follows.

- We propose a methodology to design DNNs for remote inference applications in a privacy-aware fashion using the Differential Privacy (DP) standard.
- We conduct extensive empirical evaluations on the well-know VGG [172] and MobileNetv2 [162] DNN models to analyze how inference privacy budget ϵ can vary according to the choices of DNN architectural parameters.
- We develop a customized privacy-aware NAS framework for remote inference to search for optimal architectural designs with respect to accuracy, performance, and privacy. Experiments demonstrate promising accuracy-performance-privacy trade-offs compared to conventional approaches.

A.3 Preliminaries

A.3.1 Differential Privacy (DP)

Differential Privacy [47, 48] has been established as a rigorous standard for providing quantifiable privacy guarantees on users' sensitive data. A formal definition for ϵ-DP is given:
Definition 1: *A randomized mechanism $\mathcal{A}$ is ϵ-differential private, iff for any adjacent inputs d and d', and any output $\mathcal{S}$ of $\mathcal{A}$,*

$$Pr[\mathcal{A}(d) = S] \leq e^{\epsilon} Pr[\mathcal{A}(d') = S]$$

where ϵ is a measurable privacy budget, whereas d and d' represent adjacent inputs differing by a single data item. d and d' are defined according to the application, where they can range from entire datasets differing by a single entry [6], or acquired signals instances differing in content by a single item (e.g., two sentences differing by at most i number of words) [196]. Generally, smaller ϵ values indicate stronger privacy guarantees.

For a deterministic function f, obtaining its ϵ-DP compliant randomized mechanism $\mathcal{A}_f$ entails the addition of noise calibrated to the global sensitivity of f. Global sensitivity, Δf, represents the maximum absolute distance $|f(d) - f(d')|$ for any adjacent input pairs d and d', and the additive noise can be incorporated as follows:

$$\mathcal{A}_f(d) = f(d) + u \tag{A.1}$$

where u represents the noise tensor which can be sampled from a Laplacian distribution $Lap(0, \frac{\Delta f}{\epsilon})$ of mean 0 and scale $\frac{\Delta f}{\epsilon}$ to attain a privacy budget of ϵ [48].

Added to its quantifiability, DP is characterized by the two essential properties of *postprocessing immunity* and *composition* [48, 85]. The first ensures that after the data has been processed through an ϵ-DP randomized mechanism to generate a specific output, further handling or processing of this output by other algorithms will *not* degrade the original privacy guarantee, i.e., the output would still remain ϵ-DP. Whereas the latter composition property characterizes aggregation of privacy losses when similar or neighboring data are processed by two DP algorithms [11, 85].

If noise perturbation is applied at the user's device, this resembles an instance of *local differential privacy* [11]. Owing to its inherent property of *immunity to post-processing*, local DP can ensure data privacy for remote inference models, where $\mathcal{A}_f$ represents the local model deployed on the user device, and the privacy guarantee is associated with making each data sample indistinguishable at the provider's side [196].

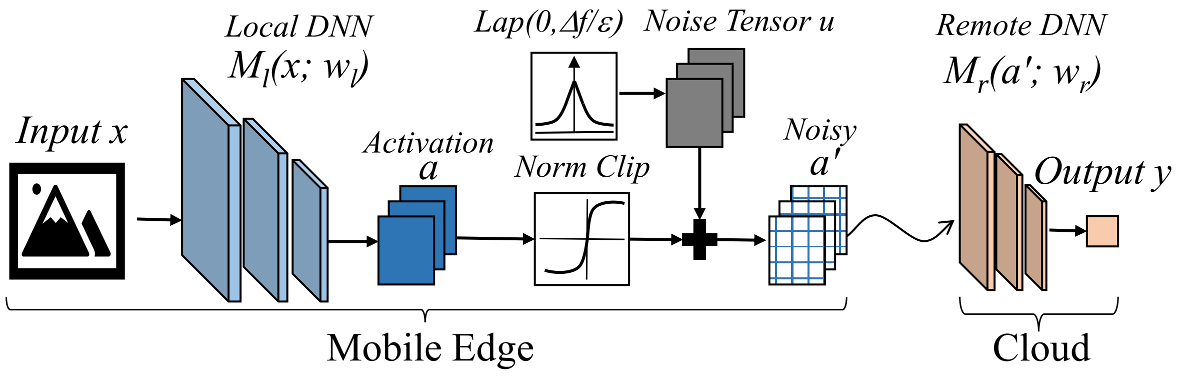


Figure A.1: Inference Privacy Model of Computation for DNNs using ϵ-DP.

A.4 ϵ-DP Remote Inference Model

Figure A.1 illustrates the basic inference privacy model of computation for deep neural network (DNNs) achieved through ϵ-DP. As shown, additional computing blocks are added on the user's mobile-edge device following the local model, $\mathcal{M}_l$, to incorporate ϵ guarantees prior to offloading. These computing blocks are described in further detail as follows:

Norm Clipping: Δf is estimated by restricting the effect of each input to an absolute maximum threshold. As the privacy guarantee ϵ is to be associated with the local model's output $\mathbf{a}$, each $\mathbf{a}$ first needs to be bounded according to the Δf estimate at the data offloading point. Thus, each output $\mathbf{a}$ is scaled down to become $\mathbf{a} \leftarrow \mathbf{a}/max(1, \frac{\|a\|_\infty}{B})$, where $\|a\|_\infty$ is the infinity norm of $\mathbf{a}$ and B is a clipping threshold leading Δf to become $2B$. The Bound B can be approximated based on the median of infinity norms belonging to output activations of public training data samples [6, 196].

Additive Noise: Next, perturbation is applied to the scaled down activation $\mathbf{a}$ proportional to the desired privacy budget ϵ (see equation A.1). In particular, a noise tensor n of the same dimensions as $\mathbf{a}$ is populated with random samples from the distribution $Lap(0, \frac{\Delta f}{\epsilon})$, and added to $\mathbf{a}$ to generate the noisy representation $\mathbf{a}'$, which can then be transmitted to the cloud. In this scheme, the cloud-side DNN can be viewed as a *post-processing* stage for the local-side DNN given how local DP is applied at the client's part of the model. Accordingly, the DP *composition* property becomes primarily associated with the local DNN side of the model, and from the cloud perspective, its outputs can typically be only traced back to the noisy intermediate representation a'.

DNN noisy retraining: As noise addition can lead the model's utility to deteriorate, retraining on publicly-available noisy data representations is beneficial. Typically, the cloud-side model $\mathcal{M}_r$ is retrained on perturbed representations of public data instances to enhance the model's resilience when dealing with noisy representations $\mathbf{a}'$. Meanwhile, local models $\mathcal{M}_l$ remain unaltered [196]. The retraining loss function for $\mathcal{M}_r$ can be as:

$$\mathcal{L}_{total}(w_r; a, a') = \lambda \mathcal{L}_{clean}(w_r; a) + (1-\lambda)\mathcal{L}_{noisy}(w_r; a') \tag{A.2}$$

where w_r represent M_r's weight parameters while λ trades off the contribution of the clean and noisy representations, taking values in the range of [0, 1].

It should be noted that there are additional techniques to further strengthen formal ϵ privacy guarantee, as the data nullification technique in [196]. However, our analysis in the following sections is conducted using the basic form of ϵ-DP remote inference model to analyze in the vanilla form the relation between a model's architectural build and privacy budget.

A.5 Analysis of the relation between DNN architectural parameters and ϵ-DP

Through intensive empirical evaluations, we examine how the ϵ-DP guarantees for remote inference can vary according to the underlying DNN structure. Our analysis is established based on the benchmark CIFAR-10 image dataset following relevant DP works [6, 107, 196].

Table A.1: The variation of ϵ with architecture and noise injection layer.

Layer		DNN Architecture			
		VGG11	VGG13	VGG16	VGG19
MP_C	Bound	0.8	0.775	0.825	0.625
	Acc.	21.0	25.7	32.7	47.4
	ϵ@b=0.5	**3.2**	**3.1**	**3.3**	**2.5**
Conv_E1	Bound	0.175	0.15	0.15	0.275
	Acc.	19.7	19.9	20.6	12
	ϵ@b=0.5	0.7	0.6	**0.6**	**1.1**

Proposition 1. ***ϵ variability per layer. Within a model**, the strength of ϵ-DP guarantee varies depending on the chosen noise injection layer. **Across model variants**, the noise injection layer capacity to influence ϵ varies based on its relative position within the computational graph.*

In this experiment, we analyze how the sensitivity bounds and the ϵ privacy guarantee vary when the position of noise injection layer changes *within* and *across* different model architectures. This first analysis is performed using pretrained models directly with no cloud-side retraining after noise injection (we leave that analysis for the immediate subsequent experiments). Briefly, the choice of offloading (noise injection) layer influences the amount of perturbation needed to achieve ϵ-DP guarantee, and in turn affects the DNN model's utility. As a motivational study, We analyze how ϵ would vary across 4 pre-trained variants of the VGG family of DNNs [172] under two potential injection layers: MP_C, which is the 3rd *Max Pooling* layer, and Conv_E1, the first *Conv* layer in the 5th block of the VGG architecture. We assume additive noise tensors are sampled from a Laplacian distribution with a scale of $b = 0.5$ (recall $b = \frac{\Delta f}{\epsilon}$). As shown in Table A.1, we observe that not only do clipping bounds B differ based on the choice of injection layer but also across the distinct variants. This implies that ϵ budget computed would be different under the same b for different layers. For instance, the privacy guarantee in VGG16 at MP_C is $\epsilon = 3.3$ opposed to $\epsilon = 0.6$ at Conv_E1. Also for the same layer across different VGG variants, a stricter

ϵ of 2.5 at MP_C can be attained for VGG19 compared to ϵ values from the other VGG variants. More interestingly, despite providing $\epsilon = 2.5$ budget at MP_C for VGG19, the model's utility does not degrade as much as that for the other variant models with looser ϵ budgets at MP_C.

Key Takeaway. *The noise injection layer position is to be optimized alongside the design process of DNN model architectures supporting ϵ-DP for inference privacy.*

Proposition 2. ***Depth.*** *For the same required ϵ budget, the likelihood of a model sustaining severe drop in accuracy decreases as the noise injection layer position tends towards the model's deeper layers.*

Depth: In this analysis, we assume a tight privacy budget requirement of $\epsilon = 2.8$ based on results from [146,196]. We use two DNNs, VGG11 and VGG16 [172], trained to $\sim 94\%$ test accuracy on CIFAR-10 using the training hyperparameters in [2]. These two architectures resemble architectural depth variation since VGG16 possess one more *Conv* layer per each block than VGG11, but they share other architectural configuration parameters. Then for each layer, we evaluate how the overall DNN utility would degrade when the layer applies noise injection. We also retrain the cloud-side DNN on noisy perturbations in each case, and re-evaluate the overall utility. The two upper bar plots in Figure A.2 demonstrate how the accuracy is impacted for every potential injection layer for both DNNs. At a first glance, we can observe that the deeper injection layers generally offer better overall accuracy under a specific ϵ. This is comprehensible given how they already deal with more abstract representations of data. We also notice that the required perturbation level b (shown in red) to achieve *2.8-DP* differs for each layer depending on the corresponding bounds, which are estimated using infinity norms' median at each respective layer. We also notice that the model's classification capability suffers significantly when the injection layer is set prior to *MP_C*. On the other hand, we also note that the accuracy does not degrade as much for the VGG16 as its counterpart does across the latter layers for the same b requirement.

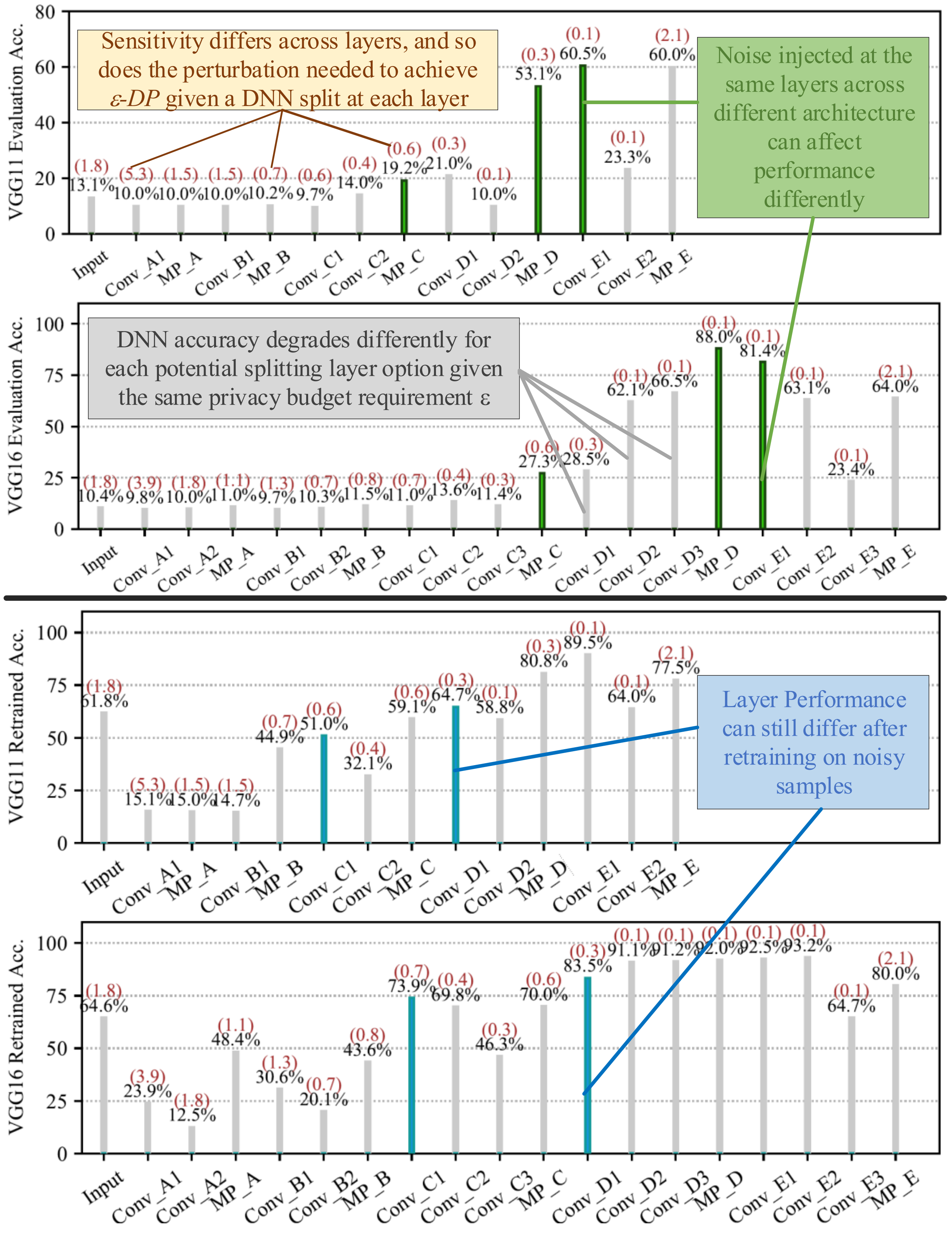


Figure A.2: The effect of varying architectural depth on utility illustrated through VGG11 and VGG16 before and after retraining for each potential injection layer (red values on top of the bars indicate b values).

The other two bar plots show the accuracy of the DNNs after the noisy retraining of the cloud-side DNN. The key insight is that some of injection layers which initially led to a complete loss of utility (e.g., *Conv_C1*) can reach acceptable levels of accuracy after retraining. We observe that the deeper VGG16 layers generally sustain better recovery of utility after retraining.

Key Takeaway. *Retraining cloud-side DNNs on noisy representations can enhance the inference privacy model's accuracy. Thus, the chosen depth of noise injection presents a trade-off between privacy, accuracy, and computational complexity.*

Proposition 3. ***Operation type.*** *The choice of kernel operations within a DNN model after ϵ-DP noise injection affects its degree of utility drop.*

Operation type: We compare the pretrained VGG16 against a variant which possess 5×5 *Conv* layers at blocks A and C (1^{st} and 3^{rd}) instead of the traditional 3×3 operations. As shown in the top two entries in Table A.2, same layers with disparate operations affect the model's utility differently. Specifically, some injection layers can degrade utility severely in one architecture but not the other. For example, when *MP_C* is set as the injection layer, performance drops significantly in VGG16 but not in its variant even after retraining. The opposite occurs when the injection layer is *Conv_D3*, showing how the operations' type influence utility under privacy constraints.

Key Takeaway. *The inference privacy-by-design approach for DNN model architectures is to co-optimize the choice of DNN operators alongside the noise injection position.*

Proposition 4. ***Width (# Channels).*** *Wider models with larger #channels can leverage computational redundancy to support inference privacy with tighter ϵ budgets.*

Width (# Channels): We provide a VGG16 variant containing half the number of output channels at its 2^{nd} and 4^{th} blocks, B and D. Despite being more concise, the variant's performance for different injection layers, shown in the last entry of Table A.2, is almost

Table A.2: Operation and Width variations effect on utility for ϵ=2.8; all architectures are pre-trained to $\sim$ 94% test accuracy.

Architecture	Layer	b	Eval. Acc.	Ret. Acc.
VGG16	MP_C	0.6	27.3%	**63.9**%
	Conv_D2	0.1	62.1%	91.1%
	Conv_D3	0.1	66.5%	**91.2**%
VGG16 5×5 ops.@ blocks A&C	MP_C	0.5	51.4%	**83.8**%
	Conv_D2	0.1	52.4%	90.2%
	Conv_D3	0.1	29.4%	**42.2**%
VGG16 0.5×#C_{in}@ blocks B&D	MP_C	0.6	26.5%	65.2%
	Conv_D2	0.2	68.6%	91.7%
	Conv_D3	0.1	27.9%	87.4%

equivalent to that of the original VGG16 after retraining. In principal, this experiment is analogous to an analysis of the impact of channel pruning [73], except that the evaluation of performance degradation is performed on a mixture of clean and noisy samples.

Key Takeaway. *Designing model architectures for ϵ-DP inference privacy is to enable compensating for the accuracy drop incurred from noise injection through including redundant computations along the width dimensions that aid in recovering utility.*

Proposition 5. ***Residual Models.*** *To uphold the ϵ-DP inference privacy guarantee for residual models, the noise injection process* ***must*** *consider the model's multi-execution paths along its branches.*

Residual Models: For residual models like MobileNetV2 [162], we perform the same per-layer empirical performance analysis at $\epsilon = 2.8$, and evaluate the model's utility before and after retraining. The main takeaway was that partitioning and noise injection options should be restricted to the final concatenation/addition layer of every residual block, i.e., on an *inter-block* basis to avoid costly noise injection across multiple paths.

Key Takeaway. *The joining nodes that follow multi-path residual blocks represent the most adequate candidates for noise injection layers for a residual DNN model.*

A.6 PrivyNAS: Privacy-Aware Neural Architecture Search Framework for Split Computing

We establish our customized PrivyNAS framework over the ProxylessNAS framework [26], which constructs a supernet out of the MobileNetv2 architecture [162], with a learnable gating mechanism to control the sampling of architectural parameters. We support ϵ-DP within the NAS frameworks as follows through the following added features.

A.6.1 Bounds estimation within NAS

In Section A.5, bounds B were determined for a pre-trained model using the infinity norms of training data at the corresponding splitting layers. In a NAS framework however, the final model architecture is not known a priori, meaning that parameter weights – and accordingly B estimates – are susceptible to changes as the search progress.

Hence, we propose a *successive refinement* approach for the B estimates to enable the application of fine updates to the Laplacian noise distributions as the search progresses. As illustrated in Figure A.3, this is motivated by the observed pattern in which infinity norm estimates for B change during the training of a MobileNetV2 architecture on CIFAR10. In particular, we notice that the norm values across the various layers progress in the exact opposite manner of training accuracy, where the initial sharp increase in accuracy is mirrored by a sharp decline in infinity norms. Additionally, once training enters the fine-tuning phase (after $\sim$ epoch 10), the rate of change of infinity norm values remarkably drops until the training concludes, as shown in the Figure through how the median and mean estimates changes. Analogously, NAS approaches like ProxylessNAS with the supernet and shared weights features can leverage this proposed successive bounds refinement technique as the supernet represents a generalized DNN model.

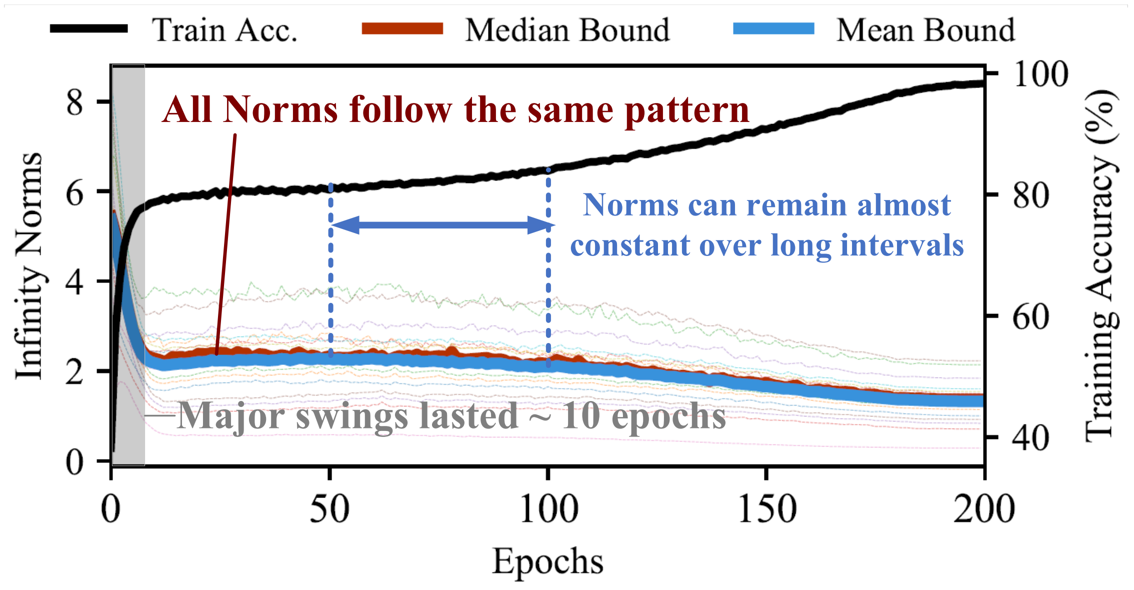


Figure A.3: The change in infinity norms (i.e., sensitivity bounds) and the training accuracy for a MobileNetV2 network on the CIFAR-10 dataset.

Successive bounds refinement is instantiated after an initial warmup training phase using clean data representations to bypass large swings. Afterwards, preliminary B estimates are computed to construct Laplacian noise distributions that meet ϵ for every candidate operation in the supernet, from which noise tensors can be sampled during the search. B estimates can then be updated using the clean data samples every number of iterations.

A.6.2 Joint Training

For inference privacy, cloud-side DNN retraining on noisy data representations is needed to maintain a model's utility (see Section A.4). In NAS, this is infeasible as neither the architecture nor the splitting layer are known beforehand. Alternatively, we propose to jointly train the supernet from the start on clean and noisy data representations where the injection layer is to be chosen dynamically as the search progresses. This is feasible in one-shot NAS techniques as all candidate architectures are trained simultaneously, and noisy representations can be provided through pre-specified ϵ budgets. Hence, the formulation of the training loss function in equation A.2 is to be modified to include the entire weights of the supernet w rather than only the cloud-side parameters. Compared to the fixed noisy layer approaches, we found that joint training across different layers could lead to severe accuracy degradation. Hence, the NAS would need to learn how to filter under-performing noise injection layers, and learn to focus the training effort on most promising layers.

A.6.3 Search Parameters Setup

Formally, the DNN model can be characterized as a DAG $\mathcal{M}(e_1, ..., e_n)$, with each edge, e_k, representing the k^{th} layer operation. In the context of NAS, the supernet represents a more generalized form of the DNN model, where instead of a singular operation, a set of all possible N operations, $\mathcal{O} = \{o_i\}$, becomes associated with each edge at layer k. Thus, e_k represents a mixed operation edge, $m_{\mathcal{O}}^k$, with N parallel paths. Hence, the overall supernet DAG can be characterized as $\mathcal{M}(m_{\mathcal{O}}^1, ..., m_{\mathcal{O}}^n)$.

To sample a candidate model design (i.e., subnet) from within the supernet, a single path is to be selected out of the $|\mathcal{O}|$ choices from $m_{\mathcal{O}}^k$ for every k^{th} layer to form a subnet. To guide the selection of paths, N learnable architectural parameters, $\{\alpha_i\}$, are specified for the N edge paths for each k^{th} layer in order to guide the selection of o_i, and ultimately provide the output features from the mixed operation $m_{\mathcal{O}}^k$. To elaborate, the application of $m_{\mathcal{O}}^k$ on an input x can be seen as $m_{\mathcal{O}}^k(x) = \sum_{i=1}^{N} g_i o_i(x)$, where $g_i \in \{0, 1\}$ represents a binary gate associated with each o_i, and only one g_i can be active at a time. In this case, α_i can be used to determine the probability of g_i being the active gate through Softmax probabilities.

For edge-cloud inference privacy, the NAS search is also to be responsible for identifying the optimal splitting/noise injection layer between the edge and cloud. Similar to candidate architecture sampling, we define additional customized K injection parameters, denoted as $\{\gamma_k\}$, that are associated with the K possible injection layer positions in the supernet. Hence,

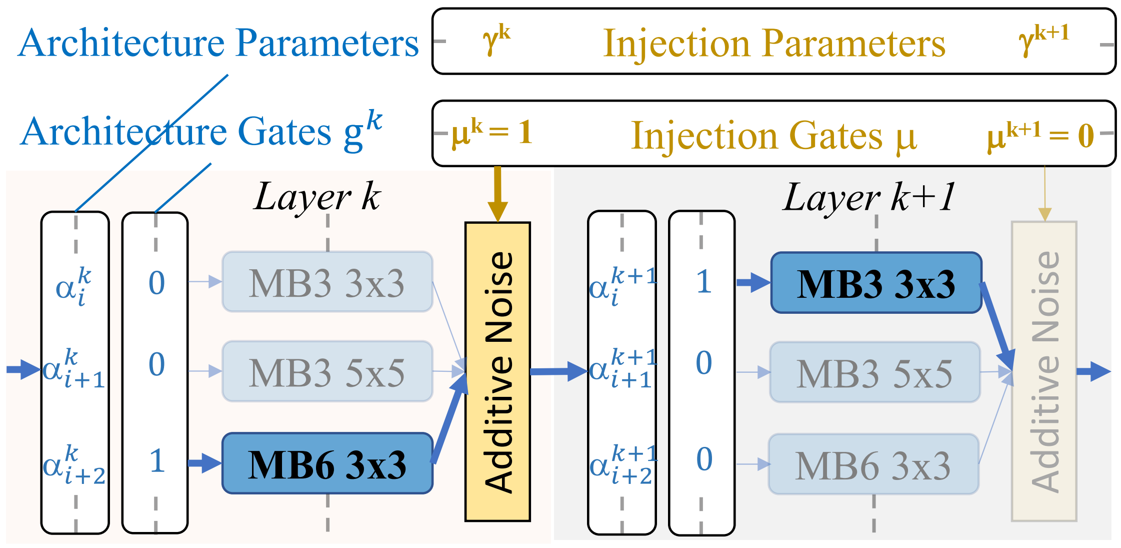


Figure A.4: Sampling of operations and the injection layer. *MBx y×y* are candidate operations from our experiments' search space inspired by [26]

the activation of an injection layer becomes also governed by a separate set of gates, $\{\mu_k\}$, where only a single μ_k can be active at any time via Softmax probability. Given the active noise injection layer at position k, the output from $m_{\mathcal{O}}^k$ is perturbed via a noise addition function, ρ_k as follows:

$$\hat{m}_{\mathcal{O}}^k = \rho_k(m_{\mathcal{O}}^k) = m_{\mathcal{O}}^k + u_k \tag{A.3}$$

where u_k represents a noise tensor to be added to the output of $m_{\mathcal{O}}^k$ at the k^{th}. An example for sampling a candidate and the noisy injection layer is shown in Figure A.4.

A.6.4 Architecture and Injection Parameters Updates

The main objective from this customized NAS implementation is to jointly optimize the supernet's architecture α, injection γ, and weight w parameters. Consequently, this bi-level

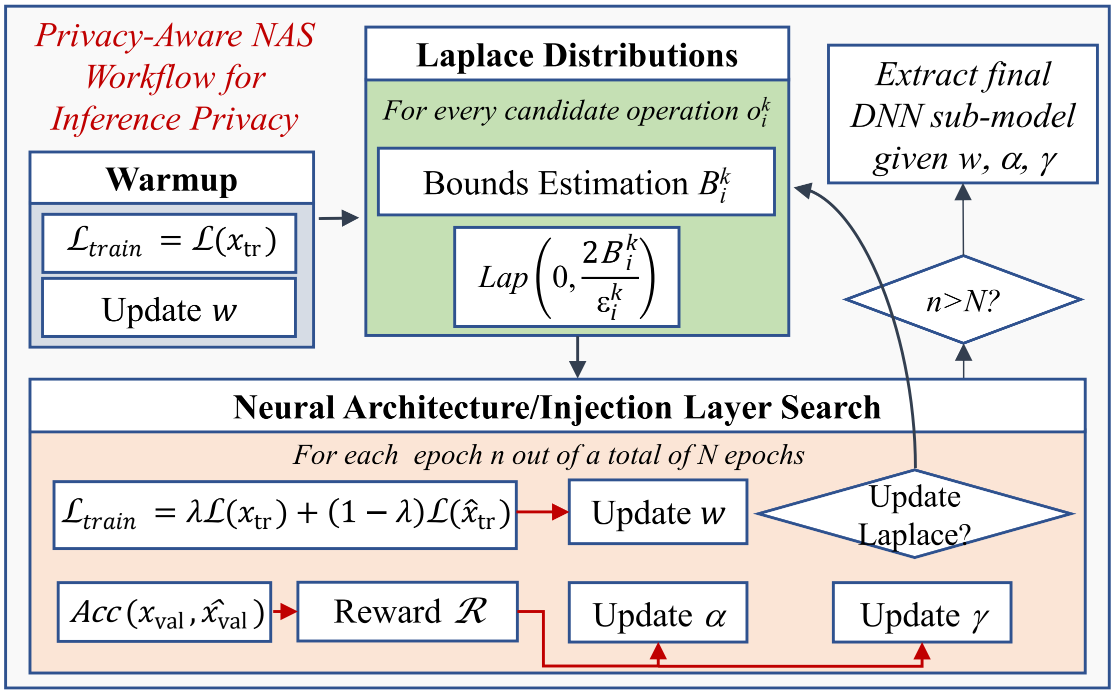

Privacy-Aware NAS Workflow for Inference Privacy
Warmup
$\mathcal{L}_{train} = \mathcal{L}(x_{tr})$
Update w
Laplace Distributions
For every candidate operation o_i^k
Bounds Estimation B_i^k
$Lap\left(0, \frac{2B_i^k}{\varepsilon_i^k}\right)$
Extract final DNN sub-model given w, α, γ
$n>N$?
Neural Architecture/Injection Layer Search
For each epoch n out of a total of N epochs
$\mathcal{L}_{train} = \lambda\mathcal{L}(x_{tr}) + (1-\lambda)\mathcal{L}(\hat{x}_{tr})$
Update w
Update Laplace?
$Acc(x_{val}, \hat{x}_{val})$
Reward $\mathcal{R}$
Update α
Update γ

Figure A.5: Basic workflow for the privacy-aware NAS for split computing.

optimization problem can be formulated as follows:

$$\min_{\alpha,\gamma} \mathcal{L}_{val}(w^*(\alpha,\gamma),\alpha,\gamma) \tag{A.4}$$

$$s.t.\ w^* = argmin_w\ \mathcal{L}_{train} \tag{A.5}$$

where the search is for the optimal parameters α^* and γ^* which minimize a validation loss $\mathcal{L}_{val}$ given optimal parameter weights w^* that minimize the training loss $\mathcal{L}_{train}$. $\mathcal{L}_{train}$ is defined based on the hybrid loss function in A.2 except that it is jointly training all the weight parameters w of the supernet. $\mathcal{L}_{valid}$ uses REINFORCE [200] to update the architectural and injection parameters based on a reward $\mathcal{R}$. The details of the update rule are in [136].

A.6.5 Privacy-Aware NAS Workflow

From here, we conceptualize a privacy-aware NAS workflow in Figure A.5 for designing DNN models suited for split-computing operation under a desired privacy budget ϵ. As shown, the warmup phase first trains the supernet on clean data representations to obtain initial estimates of B_i^k for each candidate operation o_i^k at the k^{th} layer within the supernet. Subsequently, their corresponding Laplacian distributions $Lap(0, \frac{2B_i^k}{\epsilon_i^k})$ are constructed to maintain the ϵ guarantee across all possible operations and injection layers. Next, the main search procedure can be invoked to train the shared parameter weights, w, each iteration. As stated in Section A.6.2, training is based on a mix of clean/noisy public data representations. For validation, sampled models' accuracy on a clean/noisy validation data mix are used to estimate the reward $\mathcal{R}$, and update the α and γ parameters accordingly. In the background, the successive bounds refinement is invoked periodically to update Laplacian distributions. The best performing submodel architecture and its optimal noise injection layer are returned.

A.6.6 Extension to Privacy-Accuracy co-search

We also support *privacy-accuracy* co-search within PrivyNAS to facilitate the minimization of ϵ as an optimization objective. As such, rather than defining a single Laplacian distribution per each o_i^k given a predetermined ϵ, we associate a *multitude of distributions* with each operation satisfying different ϵ guarantees. Consequently, a dictionary of Z Laplacian distributions for each operation would be continuously updated as part of the successive bounds' refinement. To avoid sampling trivial noise distributions, we define *distribution parameters* θ_ϵ^k at each k layer to be associated with each prospective ϵ value that can be sampled from Z at layer k. That way, θ_ϵ^k parameters would be able to learn which ϵ values, and in turn distributions, to select for layer k when k is sampled as the noise injection layer.

We provide an illustrative example with numbered steps in Figure A.6 on how this co-search

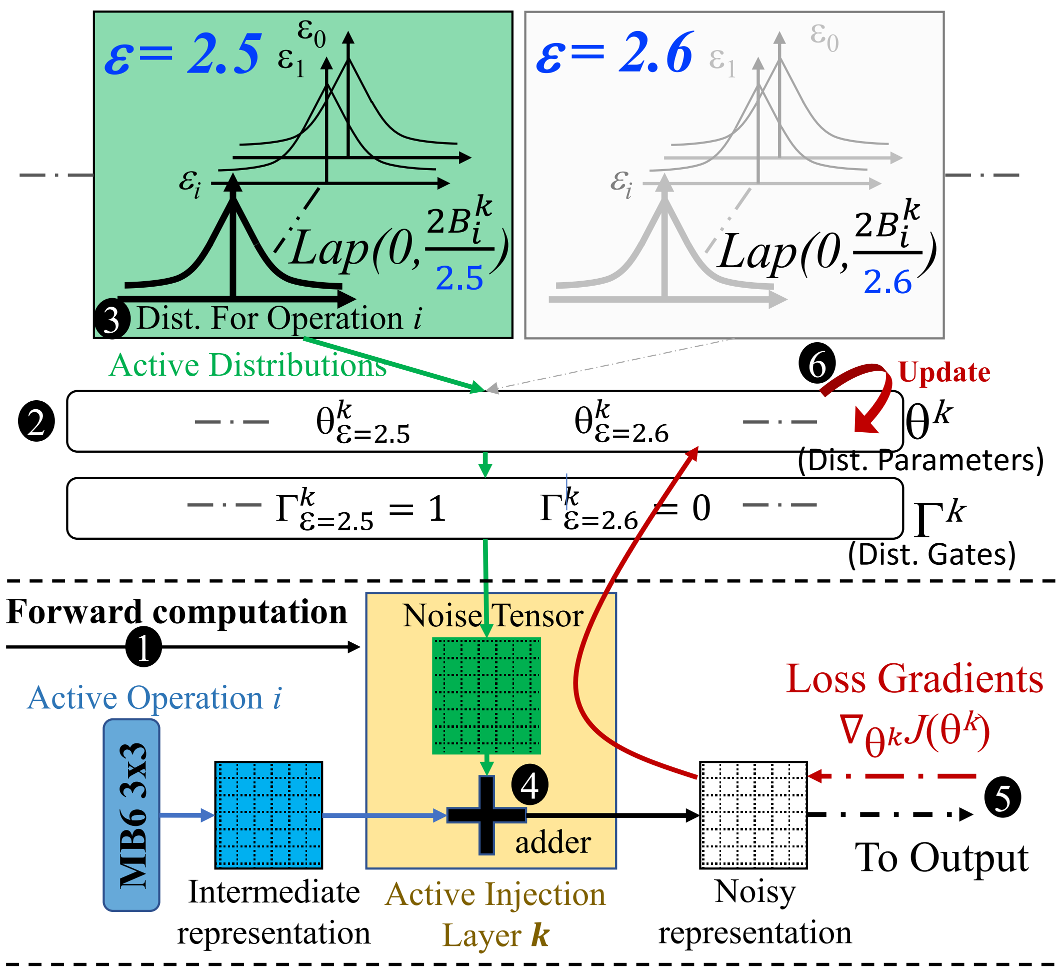


Figure A.6: Numbered step-by-step privacy-accuracy co-search (detailed in text)

progresses during a single epoch: **(1)** In the forward pass, a single operation o_i^k is activated at layer k outputting an intermediate data representation. **(2)** At active noise injection layer k, parameters θ_ϵ^k are used to select privacy budget ϵ, and its corresponding set of distributions by setting its gate Γ_ϵ^k to 1. **(3)** Based on the sampled ϵ and active operation o_i^k, the corresponding $Lap(0, \frac{2B_i^k}{\epsilon})$ is retreived. **(4)** Samples are drawn from the selected distribution to populate a noise tensor which is to be added to the intermediate representation. **(5)** The output noisy tensor is used to compute the output. **(6)** Gradients from the loss function are backpropagated to update θ^k parameters. The distribution parameters are updated in a manner similar to that for the architectural α parameters using $\mathcal{R}$, except that only the subset of θ^k parameters belonging to the current active k layer are updated at a time.

A.6.7 Reward definition and privacy-accuracy co-search support

For remote inference models, there exists an inherent trade-off between accuracy, performance, and privacy guarantees. Hence, we define the search's reward function, $\mathcal{R}$, as:

$$\mathcal{R} = acc(m) \times (\frac{L_{target}}{L(m)})^{\omega_L} \times (\frac{\epsilon_{target}}{\epsilon(m)})^{\omega_\epsilon} \tag{A.6}$$

where $acc(m)$, $\epsilon(m)$[1] , and $L(m)$ are the respective clean/noisy test accuracy, privacy budget, and latency achieved by model m. ϵ_{target} and L_{target} resemble the desired target privacy budget and latency by the designer. ω_ϵ and ω_L are configurable design trade-off parameters. $L(m)$ is associated with the edge device comprising execution and transmission overheads.

A.7 Experiments

We implement our PrivyNAS on top of ProxylessNAS [26] where we keep their default architectural search hyperparameter settings with the MobileNetV2 backbone. Our search parameters in [136] enables having 21 potential candidate positions for noise injection. We use the Nvidia Jetson TX2 (TX2) as our local edge device platform and construct a corresponding lookup table for benchmarking.

A.7.1 Privacy-aware Search Analysis

We first assess the privacy-aware NAS in comparison to a conventional privacy-agnostic NAS. Here, the conventional approach is emulated through a regular search from ProxylessNAS [26], followed by a separate independent process to integrate ϵ-DP inference privacy onto the

[1]We slightly abuse the 'ϵ' notation and reuse it for the privacy objective function in addition to the privacy budget. Purpose can be inferred from context.

Table A.3: Comparing models' accuracy (%) at ϵ=2.8. For the NAS models, we set ω_e=0, and L_{target} = 50ms when $\omega_l \neq 0$. *(c)* indicates training using only clean data while *(ret.)* stands for retrained.

Model	Total Train	Clean Test	Noisy Test	Total Test
MobileNetv2	98.9 *(c)*	92.5	27.9	60.24
MobileNetv2 *(ret.)*	85.72	88.7	65.1	76.92
ProxylessNAS ($\omega_l = 0$)	91.1 *(c)*	80.35	65.58	72.23
ProxylessNAS$_{ret}$ ($\omega_l = 0$)	85.86	80.05	**78.36**	**79.16**
Privacy-aware ($\omega_l = 0$)	**90.76**	**80.44**	**79.22**	**79.90**
Privacy-aware ($\omega_l = 0.05$)	**88.99**	**79.36**	**76.99**	**78.11**

final model. Performance evaluations for Mthe baselines are estimated based on their best accuracy scores by choosing their optimal splitting layer. A privacy budget of $\epsilon = 2.8$ is used for all implementations. As shown in Table A.3, our privacy-aware search renders a model which outperforms others in terms of average overall accuracy with 79.9%, indicating the value of the joint training during the search itself (Section A.6.2). Also, the variance between the clean and noisy test accuracies for the privacy-aware search reached 1.22%, indicating how the supernet is trained during the search to generalize its classification performance to samples perturbed in proportion to the desired ϵ budget.

A.7.2 Injection Parameters γ Analysis

We further analyze the accuracy of both our privacy-aware and the retrained ProxylessNAS models for each potential injection layer position to assess the merit of the γ parameters. In Figure A.7, the two DNNs are compared over the 21 potential injection layer positions from the search space. Note that not only does the Figure compare optimal noise injection layer choices, but all potential choices including suboptimal ones. This is to demonstrate how the γ parameters of PrivyNAS learn to sample more frequently the most promising splitting layer candidates, and subsequently focus the supernet's training around them. As such, we find that the ProxylessNAS model outperforms PrivyNAS at subopitmal noise injection layer choices as the former randomly samples candidate injection layers with the same probability. However, ProxylessNAS does not outperform PrivyNAS at the latter's best injection positions (layers $12, 16, 17, 18$, and 19) which provide the highest accuracy overall under ϵ guarantees. This is attributed to the γ parameters that learn to optimize the model architectural design around these most promising injection layer positions as a result of their higher rewards compared to sub-optimal candidates.

A.7.3 Optimizing for performance and inference privacy

Since one motivation of split computing is to reduce computational overheads on user edge devices, we conduct another experiment in which both our privacy-aware model ($\omega_l = 0.05$, $L_{target} = 50ms$ and the *ProxylessNAS* architectures from Table A.3) are first trained from scratch on clean data samples. Afterwards, their cloud-side parameters are trained on noisy samples at $\epsilon = 2.8$. Our analysis of accuracy and latency is performed at layers 8 and 12, which were the respective optimal noise injection layers for our privacy-aware model and the conventional ProxylessNAS one, respectively. As illustrated in Figure A.8, the latency-agnostic *ProxylessNAS* model achieved the highest accuracy scores at its best injection layer, 12. This is because its architecture was only optimized for accuracy without any consideration of performance overheads, and thus it incurs a high execution latency for its local DNN components reaching 101.09 ms. Whereas at injection layer 8 (our model's best), we find that our model, designed with an L_{target}=50 ms, takes 53 ms latency to execute its local DNN components ($\mathcal{M}_l$) – a 35.2% reduction from that of ProxylessNAS model. Though such performance improvement comes at the expense of 2.2% accuracy drop from the layer 12 accuracy of ProxylessNAS model, our model improves accuracy by 3.4% in the scope of layer 8 only.

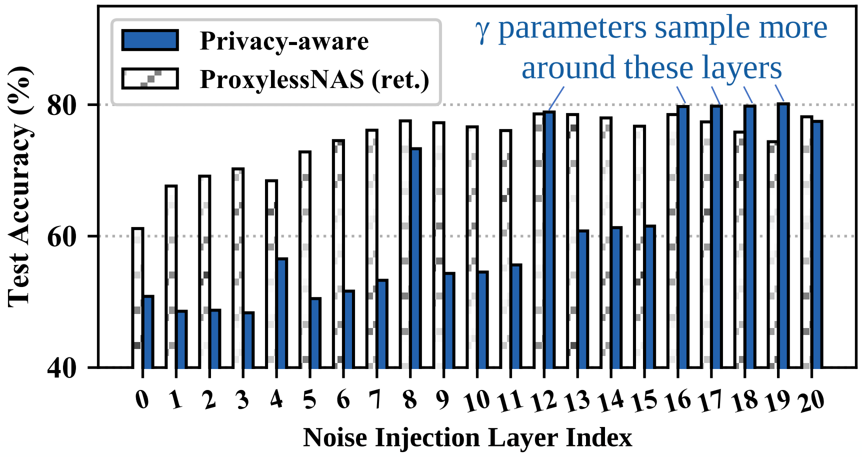


Figure A.7: Accuracy estimates of ProxylessNAS$_{ret}$ and our Privacy-aware ($\omega_l = 0$) across every potential injection layer at $\epsilon = 2.8$.

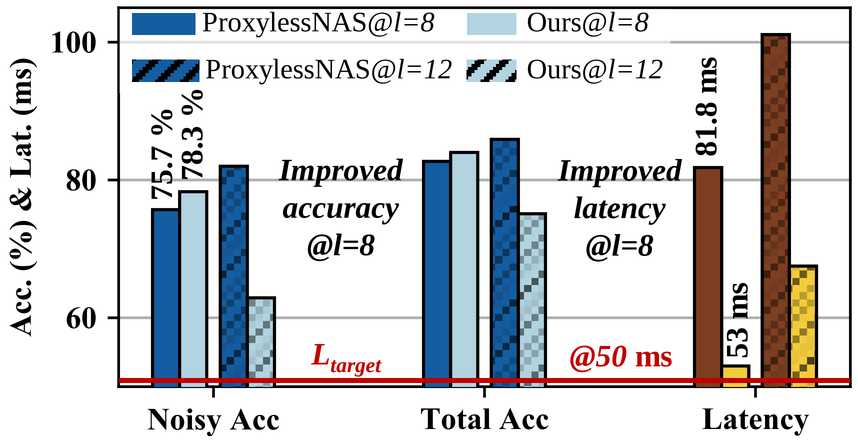


Figure A.8: Comparing noisy accuracy, total accuracy, and latency between our Privacy-aware ($\omega_l = 0.05$) and ProxylessNAS$_{ret}$ after cloud-side DNN retraining, where injection layers, 8 and 12, are the respective bests for each architecture. (*Note legend patterns hold for both accuracy and latency bars colored in blue and brown, respectively*)

A.8 Discussion and Concluding Remarks

The consideration of inference privacy during the architectural design phase of DL models can be perceived as analogous to building a model that generalizes well to inputs from a different data distribution. For ϵ-DP inference, generalization is targeted towards samples experiencing randomized perturbations according to a formal measure of noise injection [38]. Subsequently, a privacy-aware design approach exhibits in essence a behavior similar to its conventional counterpart. That is, a more complex DNN architecture would generally lead to a better model utility under privacy considerations. Still, fine tuning the architectural design well-characterization of the desired privacy budget can lead to a good balance between generalization and overfitting.

Further experimentation and analysis are needed along this direction to justify the value of accuracy-privacy co-design in practical application settings. One direction is to scale the inference privacy problem setting to practical application domains entailing the edge-cloud architecture. An example is mobile health application through which sensitive time-series data can be processed on the cloud. Another one is to study how the quality of a privacy leakage attack (e.g., membership inference or model inversion) degrades when applied to models provided by PrivyNAS compared to the baselines.

Appendix B

HW/SW Co-design for Digital Mobile Health Applications

B.1 Overview

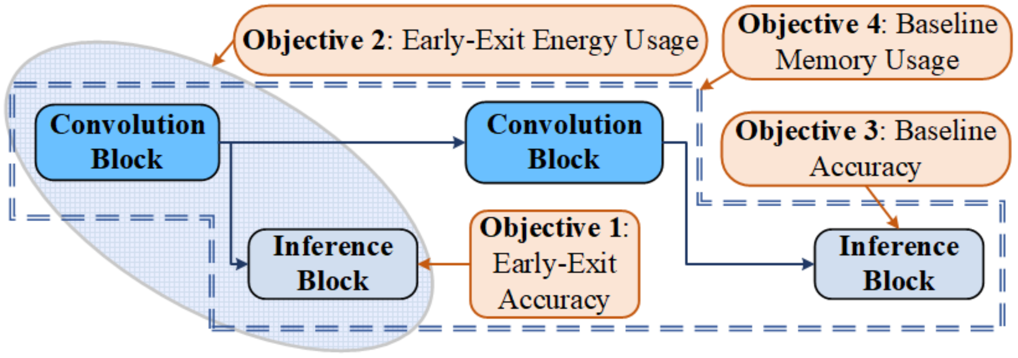
At the early stages of my PhD, I worked on a number of projects pertaining to mobile health and tinyML, where the premise was to deploy tiny machine learning models onto constrained wearables having limited compute capabilities and few kBs of memory. Our approach entailed proposing a number of neural architecture search frameworks that consider extreme model compression (account for binary neural networks) and dynamic neural networks. In the following, I showcase some of the principles we adopted for the mobile health use cases in our works [140, 141], with applications in human activity recognition and Myocardial Infarction detection.

B.2 Introduction

Adopting data-driven methodologies using Machine Learning (ML) in healthcare has become an essential practice for analyzing numerous patients' physiological signals. More so, to provide continuous real-time monitoring, the utilization of wearable devices across different applications has been growing. Traditionally, the modus operandi of these wearable devices instantiates sending the collected raw data to an intermediate device (e.g., smartphone) using a low-power wireless technology (e.g., Bluetooth). Then, the data is relayed further to a central server where the processing can take place.

However, the overheads associated with data transmission, added to the privacy considerations of sending personal physiological data over the wireless medium, represent pressing concerns for the traditional approach. As a result, the current growing trend promotes the adoption of edge computing, where the bulk of the processing is moved to the wearable device. In this setting, the wearable device can contain an ML model to perform the needed processing, and only the final result is relayed further up the hierarchy of devices. Although the traditional approach concerns are addressed, fitting adequate ML models within the resource-impoverished wearable devices is the main challenge in edge computing.

Furthermore, for mobile health applications, the patients' physiological data mostly comes in a time-series data format. Principally, time-series data can be processed through a variety of ML techniques, including Support Vector Machines (SVMs), Random Forests (RFs), 1D Convolutional Neural Networks (CNNs), and so on [8, 156, 157, 174, 175]. Although these techniques are adaptable for wearable devices' deployment, the 1D CNNs possess the merit of automatically capturing the features of time-series data through their inherent convolution operations, omitting the need for an expensive preceding feature extraction process. Furthermore, if the wearable device solution is complemented with an Early Exit (EEx) option [145], significant resource savings can be attained. This is mainly because in real-time monitoring applications, many data segments are easily identifiable like the *normal* segments, and they do not require much computational effort to be classified. Therefore, a policy can be set in motion determining for each data segment whether enough confidence exists at the EEx block to either terminate processing or continue execution.



Objective 2: Early-Exit Energy Usage
Objective 4: Baseline Memory Usage
Convolution Block
Convolution Block
Objective 3: Baseline Accuracy
Inference Block
Objective 1: Early-Exit Accuracy
Inference Block

Figure B.1: The template baseline architecture from EExNAS with potential objective functions associations.

B.2.1 Problem Statement and Research Contributions

Based on the previous arguments, the research challenges we aim to address in this work include:

- What techniques should be utilized to provide a generic design methodology for wearable applications that can render accurate and resource-efficient solutions?
- How to include potential EEx benefits at design time within the global design optimization problem?

Addressing the above-mentioned challenges, we propose a platform-aware Multi-Objective Neural Architecture Search (NAS) approach, namely EExNAS, that explores a pre-defined search space of architectural parameters to provide optimal model implementations with EEx capability. The most promising architectures are identified through their estimated evaluations over a designated set of objective functions. These objective functions can be accuracy- or performance-related (e.g., energy consumption and memory utilization). Also, they can be defined at different parts of the backbone architecture to promote the EEx capability, as shown in Figure B.1. Our research contributions can be summarized as follows:

- We propose EExNAS, a Multi-Objective NAS-based design methodology to develop resource-efficient solutions for wearable applications employing time-series data.
- We separately associate objective functions at the EEx block to optimize its implementation.
- We demonstrate the effectiveness of EExNAS across two wearable applications, Myocardial Infarction (MI) detection and Human Activity Recognition (HAR).
- On the PTB ECG dataset [20], EExNAS final solutions achieve state-of-the-art accuracy for MI detection on wearable devices, reaching 96.5% and 98.54%.
- On the w-HAR dataset [18], EExNAS final solution incurs a 0.584% accuracy drop from the state-of-the-art but is 47.076% more energy-efficient.

B.3 Related Works

B.3.1 Myocardial Infarction (MI)

MI represents one of the leading causes of death in the USA, leading to more than 600,000 deaths per year [3]. Its silent and recurrent nature necessitates continuous monitoring, where wearable devices equipped with Electrocardiogram (ECG) monitoring capability can be a viable option for real-time MI detection. In this regard, we focus on studies using only one lead ECG signal for MI detection as one lead suits the wearable devices' small form factor.

Works targeting MI detection to be compared against include k-NN [7], SVM [174], RF [175], and CNN [8, 42, 156]. All the works are suited for deployment on wearable devices except the k-NN because all the training data are required on the target device.

B.3.2 Human Activity Recognition (HAR)

HAR is used in a variety of applications like health monitoring or health tracking [19]. In this work, we showcase how the EExNAS can generalize to other wearable applications through a classification task of HAR locomotion activities using the w-HAR dataset [18]. Data segments from this dataset can belong to one of 8 existing classes, encompassing static or dynamic activities. The work in [17] proposed a hierarchical activity-aware classifier that first reckons whether the activity is static or dynamic through an SVM classifier. Then if the activity is dynamic, a decision tree classifier is invoked to determine the activity type. Their model's accuracy reached 97.34%, and our final models will be benchmarked against it.

B.4 EExNAS Design Methodology

Figure B.2 illustrates an overview of EExNAS methodology. We go through the main components of the methodology in the following subsections.

B.4.1 Neural Architecture Search

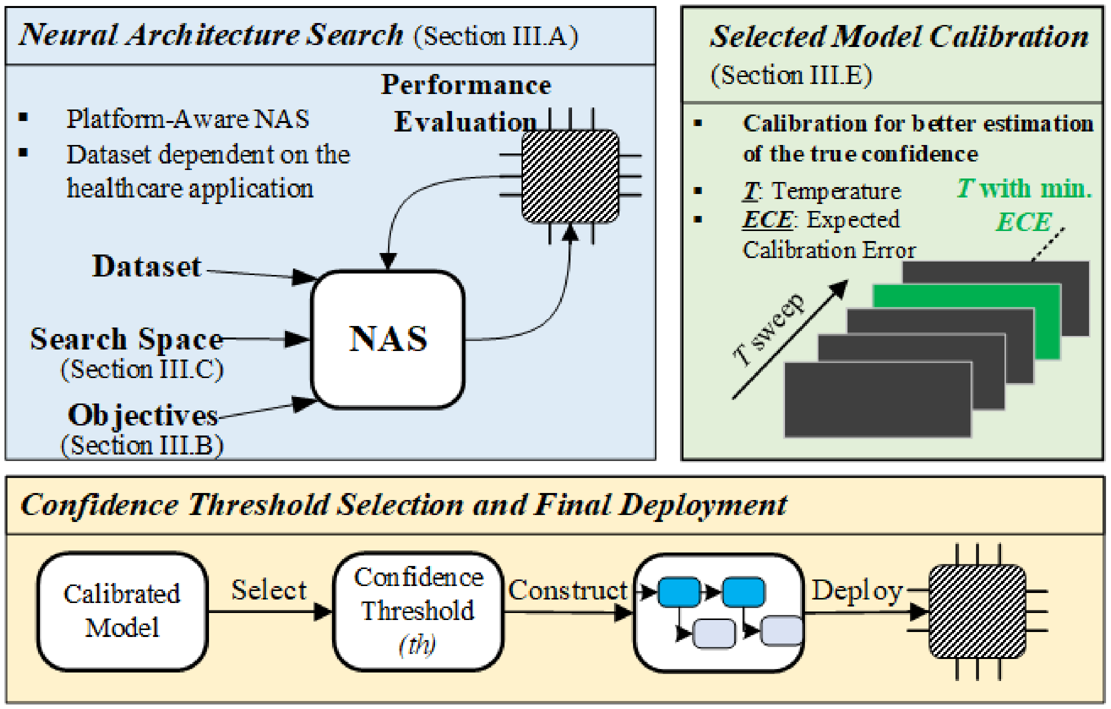
The purpose of the NAS within EExNAS is not only to identify the models with the best accuracy evaluations but also the ones that efficiently utilize the limited resources of the target wearable device. Thus, the problem becomes a multi-objective optimization one incorporating performance objectives as well. Due to the conflicting nature between the accuracy and performance objectives, the problem would not have a single solution, but a set of Pareto optimal ones instead. These Pareto-optimal solutions dominate all other explored solutions except each other, where formally in a minimization context, an architecture $x^\star$ would belong to the Pareto set if: $f_k(x^\star) \leq f_k(x) \forall k, x$ and $\exists j : f_j(x^\star) < f_j(x) \forall x \neq x^\star$. where f_k represents the k^{th} objective function.

When searching for the Pareto optimal architectures, a NAS controller each iteration needs to: sample architectural candidates from the search space, evaluate their respective objective functions, and update its search strategy based on these evaluations. In this work, the search strategy employed by the EExNAS controller is implemented using Multi-Objective Bayesian Optimization (MOBO) [165]. Other strategies like RL could've been implemented as well without any loss of generality. MOBO approximates each objective function with a surrogate Gaussian Process (GP) model. Thus, previous evaluations f_{kn} of the k^{th} objective function at iteration n are assumed to be jointly Gaussian with mean m and co-variance κ, *i.e.*, $\mathrm{f}_{kn}|x_{1:n} \sim N(m, \kappa)$, making the GP model a probabilistic distribution over possible functions of the associated objective function. From these GP models, an acquisition function is constructed and solved analytically to identify the next query point, whose true objective evaluations are determined and used to update the GP models.

B.4.2 Multi-Objective Formulation

To solve the multi-objective optimization problem, we apply linear scalarization across the multiple objective function estimates to identify the next query point through:

$$x_n = \arg\max_x \sum_i w_i . f_i \tag{B.1}$$



Neural Architecture Search (Section III.A)
Platform-Aware NAS
Dataset dependent on the healthcare application
Performance Evaluation
Dataset
Search Space (Section III.C)
Objectives (Section III.B)
NAS
Selected Model Calibration (Section III.E)
Calibration for better estimation of the true confidence
T: Temperature
ECE: Expected Calibration Error
T with min. ECE
T sweep
Confidence Threshold Selection and Final Deployment
Calibrated Model
Select
Confidence Threshold (th)
Construct
Deploy

Figure B.2: EExNAS Design Methodology Overview

where our aim each iteration n is to identify the sample x_n which maximizes a reward associated with the objective functions. Note that f_i and w_i represent the i^{th} function estimate sampled from the respective objective's GP model and its associated user-assigned weight, respectively.

To maximize the effect of the function weights, true evaluations of each objective function F_i need to be normalized with respect to their max. and min. attainable values as follows:

$$F_{i_{norm.}} = \frac{F_i - F_{i_{min}}}{F_{i_{max}} - F_{i_{min}}} \tag{B.2}$$

Hence, each weight w_i becomes the sole determiner of the extent of impact each objective function can have on the search process. In our experiments, 4 objective functions were defined as shown in Figure B.1, where the max. and min. values for the performance objectives were obtained through evaluating the largest and smallest possible architectures in the search space, respectively. Whereas for the accuracy objectives max. and min. were estimates from well- and poorly- trained models.

B.4.3 Search Space

Our search space encompasses a backbone macro-architecture of 1D convolutional blocks with an additional EEx block. Variable architectural parameters from each convolution block are used for the search space. These parameters encompass the number of output filters in addition to the kernel and strides for each of the convolution and pooling layers, respectively (Note that pooling layers are optional for each block). Therefore, each architecture in the search space can be characterized by a string x defined as follows:

$$x = (n_{F_1}, k_{c_1}, s_{c_1}, k_{p_1}, s_{p_1}, ...k_{p_N}, s_{p_N}, fc)$$

where k and s are the kernel and stride for each successive convolutional c and pooling p layer. N is the number of blocks and fc is an extra optional fully-connected layer. Although the dimensionality increases with the number of blocks in the search space, we found that utilizing 2-3 blocks is enough for multiple healthcare applications with time-series data, keeping the dimensionality relatively low. Even so, our MOBO-based solution is built on dragonfly [86], which provides techniques to handle high-dimensionality problems if needed.

B.4.4 Binary Convolutional Neural Network (BCNN)

Operators from the BCNN architecture proposed in [156] can also be used to enrich the design space. In [156], the aim was to design an efficient CNN that can fit into wearable devices with limited memory while conserving energy resources. To achieve this, the model weights are limited only to +1 or -1. Moreover, only a binary activation function is used to clamp the inputs to either +1 or -1 as introduced in the binarized neural networks [37]. This

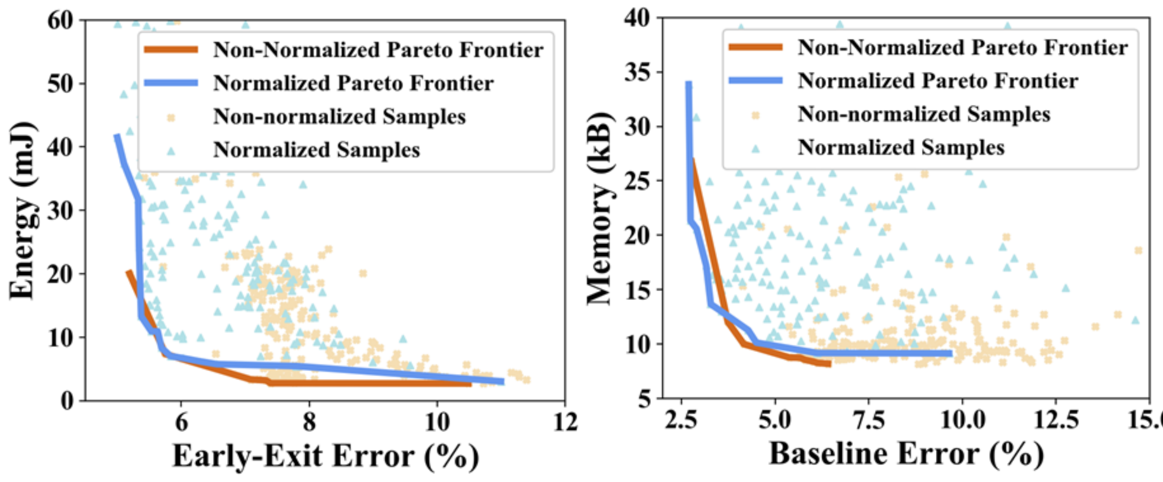


Figure B.3: Sampled architectures and Pareto frontiers in non-normalized (*brown*) and normalized (*blue*) search approaches.

binary representation of weights achieves 32× memory efficiency compared to the standard floating-point representation. Although the weights are in binary, temporaries generated between convolutional layers are still represented in floating-point. They require a lot of working memory resources which can still present an issue for wearable devices. To handle this, the computation order of inference in a binarized neural network has been modified. Unlike in the traditional order, the resulting temporaries after the convolution layer are not stored in memory. Alternatively, they are directly passed to the pooling layer followed by batch normalization and binary activation layers. This makes the models not only memory efficient but also energy efficient because of the faster and less complex binary operations.

B.5 Experimental Setup

The MOBO-based NAS runs on a desktop machine and is built on top of Dragonfly [86]. Each search run takes 200 iterations where wrapper scripts are implemented around the objective functions to automate the evaluation. Sampled architectures' from the search have their accuracy evaluations estimated after training for 30 epochs. We followed the same

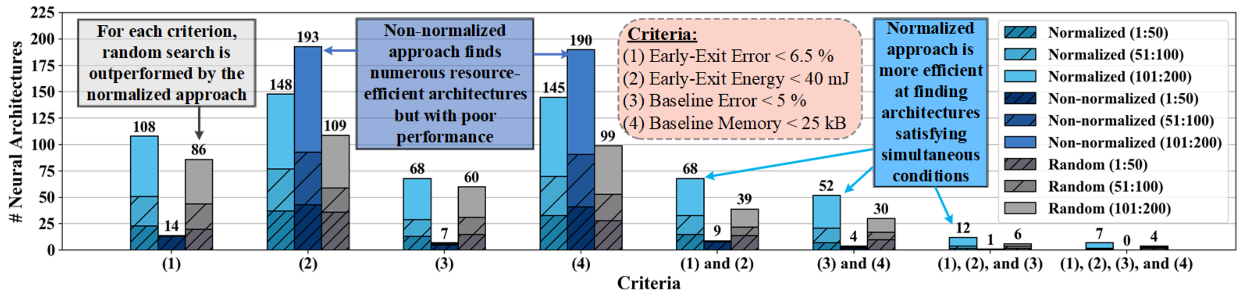


Figure B.4: Comparison between normalized, non-normalized, and random search approaches in terms of the number of architectures sampled that satisfy various criteria of the objective functions over 200 iterations of each.

Dataset preprocessing steps like filtering and segmentation in [8, 18, 156]. We also follow the training procedure in [156] for the MI ECG dataset in which the *"MI"* labeled segments are divided first into 7 groups, and then the normal segments are repeated across the 7 groups to handle the class imbalance within the dataset. The overall accuracy is then the average across the 10 fold cross-validations from all groups.

In terms of the target device, the EFM32 Giant Gecko [4] is selected as in [156, 175] for its specifications emulate those of wearable devices. Mainly, it runs on an ARM Cortex-M3 with a max operating frequency of 48 Mhz and a 128 kB RAM size. During the search process, a C code version of each sampled architecture is automatically generated and flashed onto the device to retrieve its energy and memory measurements.

B.6 Experiments and Results

The search strategy is first evaluated then the final models are bench-marked against other works as follows:

B.6.1 Search Process Assessment

First, we illustrate the effectiveness of normalizing the objective functions in Figure B.3, in which two searches, non-normalized and normalized, are run for 200 iterations each. In the non-normalized setting, it can be observed that the sample distribution is more skewed towards improving upon the performance objectives. This is due to the large variations in the energy and memory values across different models in comparison to accuracy values, making the reward estimate more reliant on these objectives. On the other hand, the normalized version remedies this through normalizing variations across all objectives between 0 and 1. In this setting, the accuracy objectives were assigned 10× more the weights of the performance objectives. Consequently, it can be seen from the samples' distribution that the search has become more biased towards minimizing the accuracy-related objectives.

Next, we compare the non-normalized, normalized, and random search approaches through the quality of their sampled architectures. The results are shown in Figure B.4 where 4 criteria are defined to reflect the models' quality with regard to the various objectives. From the figure, the first observation is that the normalized approach always outperforms the random search in identifying architectures that meet the criteria. This is attributed to the more balanced exploitation-exploration nature of the normalized search. The second observation is that although the non-normalized search finds the most resource-efficient architectures, most of these architectures are trivial and do not satisfy the accuracy-related criteria. More importantly, the non-normalized search presents the lowest number of architectures capable of satisfying simultaneous criteria, pointing up once again the necessity of normalization.

B.6.2 MI ECG Dataset Benchmarking

From the search process, two models, *a* and *b*, are chosen from the normalized Pareto frontier and retrained over 100 epochs for our final benchmarking.

We compare our models' accuracy against those from other works in Table B.1. We also provide the sensitivity and specificity evaluations as supplementary results. Although the k-NN offers the best accuracy evaluation, it is not built for wearable devices as it requires all the training data on the device. Our model *b* with EEx option at *th=0.99* offers the best accuracy results for a wearable-based solution at 98.54%. We can also observe how the calibrated conditional models can offer better accuracy than their respective baselines. Note that model *a* is the one we used for our motivational case study earlier.

In Table B.2, we compare the performance of our models against the CNN-based models [8,156]. We implemented their models in C-code and flashed them onto the EFM32 device for a fair comparison. The energy calculation for the conditional models is obtained through the summation of products of the ratios of segments classified and the total energy consumption at each exit point. Although the binarized nature of the BCNN [156] deems it the most resource-efficient solution with 13.03 mJ for each inference, the more accurate *model a (th=0.7)* is not far behind with 16.34 mJ. Note that more efficient versions of *model a* with $th < 0.7$ outperformed the BCNN with respect to both accuracy and performance. However, we focus on providing more generic model versions. Moreover, we notice *model b (th=0.99)* with the best accuracy is more efficient than its *model a (th=0.99)* counterpart. Because, unlike *model a*, the complexity of *model b*'s architecture is more evenly distributed between the two convolution blocks, making it the most suited overall candidate whenever high confidence is demanded.

B.6.3 w-HAR Benchmarking

To demonstrate how the methodology adapts to other applications, we showcase the benchmarking results on the w-HAR dataset for HAR. After NAS, *model c* is selected and retrained for 300 epochs for the final evaluation. Its optimal temperature value was found at 5.1 with

Table B.1: Performance Benchmarking on MI ECG Dataset

Work	Type	Acc. (%)	Sen. (%)	Spe. (%)
k-NN [7]	-	98.8	99.45	96.27
SVM [174]	Baseline	95	x	x
	Cond.	90	x	x
RF [175]	Baseline	83.26	87.95	78.82
	Cond.	80.32	81.02	79.63
CNN [8]	-	95.22	95.49	94.19
BCNN [156]	-	90.29	90.41	90.16
Model *a* (Ours)	Baseline	95.61	93.44	97.85
	Cond. (*th=0.7*)	**95.66**	**95.55**	**95.77**
	Cond. (*th=0.99*)	**96.5**	**95.44**	**97.59**
Model *b* (Ours)	Baseline	98.03	97.26	98.82
	Cond. (*th=0.7*)	**97.21**	**96.6**	**97.84**
	Cond. (*th=0.99*)	**98.54**	**97.66**	**99.44**

Table B.2: Measurements on the EFM32 for MI models

Work	Type	RAM Occ. (kB)	Ergy/Inf. (mJ)
CNN [8]	-	101.380	97.651
BCNN [156]	-	3.556	13.033
Model *a* (Ours)	Baseline	**15.66**	36.394
	Cond. (*th=0.7*)		**16.344**
	Cond. (*th=0.99*)		**35.978**
Model *b* (Ours)	Baseline	**15.972**	28.32
	Cond. (*th=0.7*)		**22.187**
	Cond. (*th=0.99*)		**28.189**

an ECE of 2.29%. The relatively small temperature factor means that the initial estimates were a relatively good indication of the true confidence, and the estimate values would not need to be scaled down aggressively. *Model c* is then compared against the ones in [17] in terms of both accuracy and performance. The WF1 score, obtained from the confusion matrix of each classification class, is also provided as a supplementary result. As displayed in Tables B.3 and B.4, the activity-aware implementation still achieves the best accuracy readings. However, *model c (th=0.99)* incurs a 0.584% drop in accuracy for 78.985% and 47.076% gains in memory and energy efficiency, respectively.

Table B.3: Performance Benchmarking on wHAR dataset

Work	**Type**	**Acc. (%)**	**Weigh. F1 (%)**
Baseline [17]	-	94.87	94.96
Act.-aware [17]	-	97.34	97.37
Model *c* (Ours)	Baseline	95.59	95.4
	Cond. (*th=0.9*)	**96.203**	**95.995**
	Cond. (*th=0.99*)	**96.772**	**96.627**

Table B.4: Measurements on EFM32 for HAR models

Work	**Type**	**RAM Occ. (kB)**	**Ergy/Inf. (mJ)**
[17]	Baseline	10.012	1.037
	Act.-aware		1.368
Model *c* (Ours)	Base.	2.104	0.931
	Cond. *(th=0.9)*		0.637
	Cond. *(th=0.99)*		0.724

B.7 Concluding Remarks

We have introduced a design methodology to render 1D CNN-based wearable device solutions with EEx capability. We've shown that EEx models are not only more resource-efficient but also can outperform their baselines in terms of accuracy evaluations. We demonstrated the efficiency of our methodology over MI and HAR applications,