Efficient Deployment of Deep Neural Networks on Hardware Devices for Edge AI

Halima Bouzidi

École Doctorale PHF n° 635 : Polytechnique Hauts-de-France

Thèse de Doctorat

Déposée pour obtenir le grade de docteur délivré par

l'Université Polytechnique Hauts-de-France et L'INSA Hauts-de-France

Spécialité "Informatique"

présentée et soutenue publiquement par

Halima Bouzidi

le 29 Jan 2024 à Valenciennes, France

Déploiement Efficace des Réseaux de Neurones Profonds sur les Dispositifs Matériels pour l'IA en Edge

Directeur de thèse : **Prof. Smail Niar**
Co-encadrant de thèse : **Dr. Hamza Ouarnoughi**
Co-encadrant de thèse : **Prof. El-Ghazali Talbi**

Jury

Prof. Olivier Sentieys,	Professeur, Université de Rennes, France	Rapporteur
Prof. Tulika Mitra,	Professeur, Université de Singapour, Singapour	Rapporteuse
Prof. Clarisse Dhaenens,	Professeur, Université de Lille, France	Examinatrice
Dr. Nicolas Ventroux,	Chef de laboratoire, Thales France	Examinateur
Prof. Abdessamad Ait El-Cadi,	Professeur, INSA Hauts-de-France, France	Invité

LAMIH-UPHF
Déparetement Informatique
UMR CNRS 8201, 59300 Famars, Valenciennes, France

École Doctorale PHF n° 635 : Polytechnique Hauts-de-France

Ph.D. Dissertation

Submitted to obtain a degree of Doctor in Philosophy from

l'Université Polytechnique Hauts-de-France et L'INSA Hauts-de-France

Discipline "Computer Science and Engineering"

Publicly presented and defended by

Halima Bouzidi

On the 29 Jan 2024 in Valenciennes, France

Efficient Deployment of Deep Neural Networks on Hardware Devices for Edge AI

Thesis Director : **Prof. Smail Niar**
Thesis Co-Supervisor : **Dr. Hamza Ouarnoughi**
Thesis Co-Supervisor : **Prof. El-Ghazali Talbi**

Jury

Prof. Olivier Sentieys,	Professor, University of Rennes, France	Reviewer
Prof. Tulika Mitra,	Professor, University of Singapore, Singapore	Reviewer
Prof. Clarisse Dhaenens,	Professor, University of Lille, France	Examiner
Dr. Nicolas Ventroux,	Head of Laboratory, Thales, France	Examiner
Prof. Abdessamad Ait El-Cadi,	Professor, INSA Hauts-de-France, France	Invitee

LAMIH-UPHF
Déparetement Informatique
UMR CNRS 8201, 59300 Famars, Valenciennes, France

ABSTRACT

Neural Networks (NN) have become a leading force in today's digital landscape. Inspired by the human brain, their intricate design allows them to recognize patterns, make informed decisions, and even predict forthcoming scenarios with impressive accuracy. NN are widely deployed in Internet of Things (IoT) systems, further elevating interconnected devices' capabilities by empowering them to learn and auto-adapt in real-time contexts. However, the proliferation of data produced by IoT sensors makes it difficult to send them to a centralized cloud for processing. This is where the allure of edge computing becomes captivating. Processing data closer to where it originates -at the edge- reduces latency, makes real-time decisions with less effort, and efficiently manages network congestion.

Integrating NN on edge devices for IoT enables more efficient and responsive systems, ushering in a new age of self-sustaining *Edge AI*. However, Deploying NN on resource-constrained edge devices presents a myriad of challenges : (*i*) The inherent complexity of neural network architectures, which requires significant computational and memory capabilities. (*ii*) The limited power budget of IoT devices makes the NN inference prone to rapid energy depletion, drastically reducing system utility. (*iii*) The need of ensuring harmony between NN and HW designs as they evolve at different rates. (*iv*) The lack of adaptability to the dynamic runtime environment and the intricacies of input data.

Addressing these challenges, this thesis aims to establish innovative methods that extend conventional NN design frameworks, notably Neural Architecture Search (NAS). By integrating HW and runtime contextual features, our methods aspire to enhance NN performances while abstracting the need for the *human-in-loop*. Firstly, we incorporate HW properties into the NAS by tailoring the design of NN to clock frequency variations (DVFS) to minimize energy footprint. Secondly, we leverage dynamicity within NN from a design perspective, culminating in a comprehensive Hardware-aware Dynamic NAS with DVFS features. Thirdly, we explore the potential of Graph Neural Networks (GNN) at the edge by developing a novel HW-aware NAS with distributed computing features on heterogeneous MPSoC. Fourthly, we address the SW/HW co-optimization on heterogeneous MPSoCs by proposing an innovative scheduling strategy that leverages NN adaptability and parallelism across computing units. Fifthly, we explore the prospect of *ML4ML – Machine Learning for Machine Learning* by introducing techniques to estimate NN performances on edge devices using neural architectural features and ML-based predictors. Finally, we develop an end-to-end self-adaptive evolutionary HW-aware NAS framework that progressively learns the importance of NN parameters to effectively guide the search toward Pareto optimal solutions.

Our methods can contribute to elaborating an end-to-end design framework for neural networks on edge hardware devices. They enable leveraging multiple optimization opportunities at both the software and hardware levels, thus improving the performance and efficiency of Edge AI systems.

Keywords : *Hardware-aware Neural Architecture Search*, *Dynamic Inference*, *DVFS*, *Edge AI*, *Performance Prediction*, *HW/SW Co-optimization*.

RÉSUMÉ

Les réseaux de neurones (RN) sont devenus une force majeure dans le monde de la technologie. Inspirés par le cerveau humain, leur conception complexe leur permet d'apprendre des motifs, de prendre des décisions et même de prévoir des scénarios futurs avec une précision impressionnante. Les RN sont largement déployés dans les systèmes de l'Internet des Objets (IoT pour Internet of Things), renforçant davantage les capacités des dispositifs interconnectés en leur donnant la capacité d'apprendre et de s'auto-adapter dans un contexte temps réel. Cependant, la prolifération des données produites par les capteurs IoT rend difficile leur envoi vers un centre cloud pour le traitement. Par conséquent, le traitement des données plus près de leur origine, en edge, permet de prendre des décisions en temps réel, réduisant ainsi la congestion du réseau.

L'intégration des RN à l'edge dans les systèmes IoT permet d'obtenir des solutions plus efficaces et réactives, inaugurant ainsi une nouvelle ère de edge AI. Néanmoins, le déploiement des RN sur des plateformes matérielles à ressources présente une multitude de défis. (*i*) La complexité inhérente des architectures des RN, qui nécessitent d'importantes capacités de calcul et de stockage. (*ii*) Le budget énergétique limité caractérisant les dispositifs matériels sur edge qui ne permet pas de supporter des RN complexes, réduisant drastiquement la durée de fonctionnement du système. (*iii*) Le défi d'assurer une harmonie entre la conception des RN et celle des dispositifs matériels de l'edge. (*iv*) L'absence de l'adaptabilité à l'environnement d'exécution dynamique et aux complexités des données.

Pour pallier ces problèmes, cette thèse vise à établir des méthodes innovantes qui élargissent les cadres traditionnels de conception de RN (NAS pour Neural Architecture Search) en intégrant les caractéristiques contextuelles du matériel et de l'environnement d'exécution. Tout d'abord, nous intégrons les propriétés matérielles au NAS en adaptant les RN aux variations de la fréquence d'horloge. Deuxièmement, nous exploitons l'aspect dynamique au sein des RN d'un point de vue conceptuel, en introduisant un NAS dynamique. Troisièmement, nous explorons le potentiel des RN graphiques (GNN pour Graph Neural Network) en développant un NAS avec calcul distribué sur des multiprocesseurs hétérogènes sur puce (MPSoC pour Multi-Processors Système-on-Chip). Quatrièmement, nous abordons la co-optimisation software et matérielle sur les MPSoCs hétérogènes en proposant une stratégie d'ordonnancement innovante qui exploite l'adaptabilité et le parallélisme des RN. Cinquièmement, nous explorons la perspective de *ML4ML* en introduisant des techniques d'estimation des performances des RN sur les plateformes matérielles sur edge en utilisant des méthodes basés sur ML. Enfin, nous développons un framework NAS évolutif et auto-adaptatif de bout en bout qui apprend progressivement l'importance des paramètres architecturaux du RN pour guider efficacement le processus de recherche du NAS vers l'optimalité.

Nos méthodes aident à contribuer à la réalisation d'un framework de conception de bout en bout pour les RN sur les dispositifs matériels sur edge. Elles permettent ainsi de tirer avantage de plusieurs pistes d'optimisation au niveau logiciel et matériel, améliorant les performances et l'efficacité de l'Edge AI.

Keywords : *Hardware-aware Neural Architecture Search*, *Inference Dynamique*, *DVFS*, *Edge AI*, *Prédiction de Performance*, *Co-optimization HW/SW* .

ACKNOWLEDGEMENT

Certainly, this journey has been full of ups, downs, and doubts more than anything else. With the help and support from everyone, I was able to cross the finish line. I'm so grateful for the things I've learned and the people I've met.

First and foremost, I offer my immense gratitude to my Ph.D. advisor, Prof. Smail Niar, for guiding me throughout my Ph.D. studies. His encouragement, patience, and support have been crucial in shaping both my research and my personal growth. I also thank my co-advisors, Dr. Hamza Ouarnoughi and Prof. El-Ghazali Talbi, for their invaluable guidance, support, and mentorship. I have gained numerous skills and knowledge from them that will forever shape my career.

I also offer my profound appreciation to my collaborators, Prof. Mohammad Abdullah Al-Faruque and Mohanad Odema, whose consistent support was pivotal in completing my Ph.D. journey. Their unwavering guidance has been instrumental in keeping me open to new ideas and maintaining a growth-oriented mindset.

I would like to thank my thesis committee members, Prof. Olivier Sentieys, Prof. Tulika Mitra, Prof. Clarisse Dhaenens, Dr. Nicolas Ventroux, and Prof. Abdessamad Ait El-Cadi, for their insightful comments and feedback.

I would like to express my sincere gratitude to my lab-mates, Hadjer, Imed, Farouk, Lotfi, Sofiane, Mufida, Affaf, Amine, Fabien, and Nourredine, and to my dearest friends, Yasmine, Katia, and Amal, whose support and friendship have been a source of comfort and motivation throughout my academic journey.

To my Mom, Dad, and Sisters, I've spent considerable time thinking about how to express my gratitude in a way that transcends the usual clichés, yet I find myself at a loss for words. My acknowledgment of you here isn't merely a formality, it's a heartfelt recognition. In the lottery of life, you are my winning ticket. Every good thing and every opportunity that has come my way began with you. From the depths of my heart, thank you a million times over for your endless love, support, and for bearing my absence during moments of happiness and sorrow.

Finally, in such a chaotic world, I often ask myself what can I do to be worthy of all these blessings ? I can only hope that the knowledge and skills I've gathered will empower me to contribute towards solving some of our world's issues. May we all work together towards a more fair, happy, and beautiful world.

LIST OF PUBLICATIONS

The following articles published in either international journals or conferences are directly discussed in the thesis dissertation :

1. [26] **Halima Bouzidi**, Hamza Ouarnoughi, El-Ghazali Talbi, Abdessamad Ait El Cadi, and Smail Niar. "Evolutionary-based Optimization of Hardware Configurations for DNN on Edge GPUs". The International Conference on Metaheuristics and Nature Inspired Computing, (META)" 2021.
   Location in the thesis : Chapter 2
2. [27] **Halima Bouzidi**, Hamza Ouarnoughi, El-Ghazali Talbi, Abdessamad Ait El Cadi, and Smail Niar. "Evolutionary-Based Co-optimization of DNN and Hardware Configurations on Edge GPU". The International Conference on Optimization and Learning. (OLA) 2022. Book Chapter, Communications in Computer and Information Science, Springer Nature.
   Location in the thesis : Chapter 2
3. [25] **Halima Bouzidi**, Hamza Ouarnoughi, Smail Niar, El-Ghazali Talbi, and Abdessamad Ait El Cadi. "Co-Optimization of DNN and Hardware Configurations on Edge GPUs," The Euromicro Conference on Digital System Design (DSD) 2022.
   Location in the thesis : Chapter 2
4. [21] **Halima Bouzidi**, Mohanad Odema, Hamza Ouarnoughi, Mohammad Al Faruque, Smail Niar. "HADAS : Hardware-Aware Dynamic Neural Architecture Search for Edge Performance Scaling" The Design, Automation and Test in Europe Conference and Exhibition (DATE) 2023. **Ranked among Top 30 papers and Nominated for the Best Paper Award.**
   Location in the thesis : Chapter 3
5. [165] Mohanad Odema*, **Halima Bouzidi***, Hamza Ouarnoughi, Smail Niar, Mohammad Al Faruque. "MaGNAS : A Mapping-Aware Graph Neural Architecture Search Framework for Heterogeneous MPSoC Deployment". ACM Transactions on Embedded Computing Systems 22, 5s, Article 108 (October 2023), Special Issue : CASES Conference at ESWEEK 2023.
   Location in the thesis : Chapter 4
6. [22] **Halima Bouzidi**, Mohanad Odema, Hamza Ouarnoughi, Smail Niar, Mohammad Al Faruque. 2023., "Map-and-Conquer : Energy-Efficient Mapping of Dynamic Neural Nets onto Heterogeneous MPSoCs," The ACM/IEEE Design Automation Conference (DAC), 2023. **Paper Award from HiPEAC**
   Location in the thesis : Chapter 5

7. [23] **Halima Bouzidi**, Hamza Ouarnoughi, Smail Niar, and Abdessamad Ait El Cadi. "Performance prediction for convolutional neural networks on edge GPUs". The ACM International Conference on Computing Frontiers (2021)
   Location in the thesis : Chapter 6
8. [24] **Halima Bouzidi**, Hamza Ouarnoughi, Smail Niar, and Abdessamad Ait El Cadi. "Performance Modeling of Computer Vision-based CNN on Edge GPUs". ACM Transactions on Embedded Computing Systems. 21, 5, Article 64 (September 2022)
   Location in the thesis : Chapter 6
9. [20] **Halima Bouzidi**, Hamza Ouarnoughi, Abdessamad Ait El Cadi, El-Ghazali Talbi, and Smail Niar. "Sonata : Self-adaptive Evolution for Multi-objective Hardware-aware Neural Architecture Search". Submitted to IEEE Transactions on Evolutionary Computation, 2024.
   Location in the thesis : Chapter 7

The following articles are not included or discussed in the thesis dissertation :

1. [72] Mohamed Ghebriout, **Halima Bouzidi**, Smail Niar, Hamza Ouarnoughi. "Harmonic-NAS : Hardware-Aware Multimodal Neural Architecture Search on Resource constrained Devices". The Asian Conference on Machine Learning (ACML). Proceedings of Machine Learning Research (PMLR), 2024
2. [13] Hadjer Benmeziane*, **Halima Bouzidi***, Hamza Ouarnoughi, Ozcan Ozturk, Smail Niar. "Treasure What You Have : Exploiting Similarity in Deep Neural Networks for Efficient Video Processing". Submitted to the IEEE Transactions on Computers. 2023"
3. [60] Eric Jenn, Floris Thiant, Theo Allouche, **Halima Bouzidi**, Ramon Conejo-Laguna, Omar Hlimi, Cyril Louis-Stanislas, Christophe Marabotto, Smail Niar, Serge Tembo-Mouafo and Philippe Thierion. "An Evaluation Bench for the Exploration of Machine Learning Deployment Solutions on Embedded Platforms". The European Congress on Real Time Embedded Systems (ERTS) 2024.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 The Rise of Edge AI

Edge AI refers to the fusion of Artificial Intelligence (AI) with Edge computing, heralding a new computing paradigm where AI computations can operate optimally closer to data sources – at the edge of the network system [197]. AI models, notably Neural Networks (NNs), are characterized by their deep layers and interconnected nodes that hold many neurons and parameters. Recently, NNs have demonstrated their aptitude in various domains, from image recognition and language processing to healthcare and chatbot technologies, marking their significant footprint in modern digital landscapes [185]. Given their intricate topology, NNs are computationally and memory demanding, necessitating powerful centralized Cloud servers to handle their processing on the massive amount of data generated by edge sensors. Nevertheless, due to the surge in connected users and inconsistent network bandwidth, Cloud servers struggle to guarantee prompt processing results for end-users within a tight latency delay.

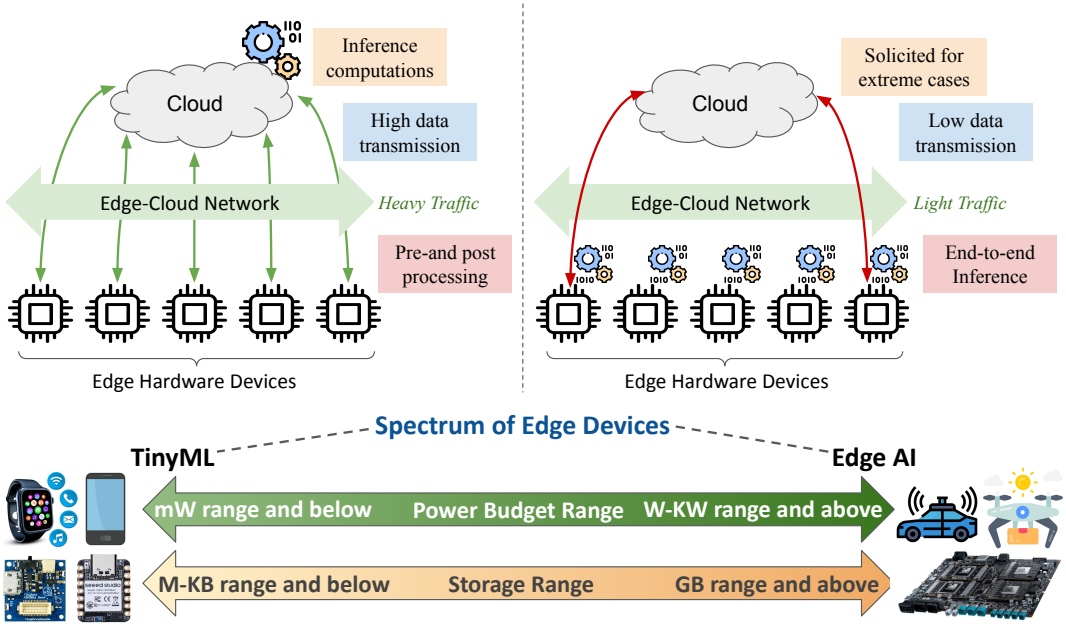


Figure 1.1 Top: Comparison between Cloud and edge computing paradigms: By delegating computations to edge devices, low data transmission and network congestion can be achieved. **Bottom**: Spectrum of edge computing devices

The advent of edge computing systems has led to a shift in the *send-to-cloud* archetype, prioritizing processing data near sensors to minimize latency and save network bandwidth. Integrating NNs with edge computing gave rise to Edge AI, in which NN models are further optimized, redesigned, or tuned to enhance their processing on less-powerful edge devices (e.g., smartphones). As shown in Figure 1.1, the benefits of Edge AI compared to Cloud computing are threefold: (*i*) reduced latency and energy consumption due to local processing. (*ii*) preserved privacy and minimal security risks by maintaining data locally. (*iii*) efficient network bandwidth management by saving it for extremely urgent cases [200].

There has been a swift and significant rise in the development of AI-specific chips, dedicated hardware accelerators, and software frameworks specifically tailored to enhance AI-on-device performances [207]. This confluence underscores the burgeoning significance of Edge AI. As we continue to witness the blossoming of applications and the introduction of advanced data processing techniques, like multimodal approaches, the melding of edge computing and neural networks is set to be crucial. Such an integration seeks to unlock the immense potential of data, paving the way for more pervasive real-time and intelligent decision-making that is more ubiquitous than ever.

1.2 The Emergence of Automated Neural Architecture Design

In recent years, the evolution of neural network design has witnessed many phases, from handcrafted to automated approaches. The newly emergent concept of *Neural Architecture Search* (NAS) has been a game changer, lessening the need for human expertise and aiming to fully automate NN's design process [270]. The paradigm of NAS lies in navigating a predefined search space encompassing the possibilities of NN architectures through a vast search process to identify the options that provide the best trade-off between prediction accuracy and computational efficiency [59]. Earlier NAS approaches [86] focused on optimizing the accuracy under efficiency proxy budgets, such as the number of Multiply-Accumulate operations (MAC) as a proxy for computational complexity or the number of learnable parameters as a proxy for model size. However, the focus has shifted with the emergence of Edge AI and the limitations of proxy metrics in capturing hardware-related aspects [23]. Instead of solely prioritizing accuracy, researchers have started pursuing a balance between accuracy and hardware efficiency. Subsequently, this gave rise to the extended concept of Hardware-aware NAS (HW-aware NAS) [33] in which neural networks are synergistically designed to optimize hardware-related metrics such as latency, energy consumption, and memory usage.

The left of Figure 1.2 illustrates neural network design techniques' evolution stages. Given the broad scope of NNs, we focus our analysis on the specific Convolutional Neural Networks (CNN) case, as they have shown impressive results in many application domains, including computer vision and natural language processing. The *First stage* marked the emergence of the earliest handcrafted CNNs (e.g., AlexNet [114], VGG [199], and GoogleNet [209]), characterized by large amounts of weights to maximize learning capacity, as accuracy was the only matter. In *Second stage*, design simplifications have been introduced to make CNN less heavy and mitigate the problem of vanishing gradient [84] due to many learnable weights. This was achieved by including skip and dense connections (e.g., ResNet [84] and Den-

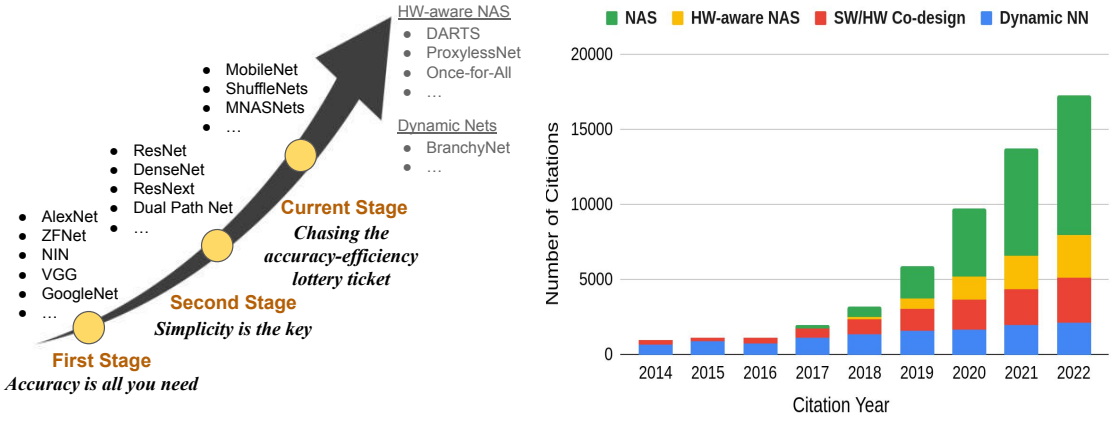


Figure 1.2 Left: The evolution stages of NN design landscape: from *static* and *handcrafted* to *dynamic* and *auto-designed*. **Right**: Statistics on the number of publications in *NAS*, *HW-aware NAS*, *SW/HW Co-design*, and *Dynamic NN* over the last years [1]

.

seNet [94]). The *Current stage* focuses on chasing the accuracy-efficiency balance that cannot be readily achieved by means of handcrafted methods. Hence, more advanced design techniques have been proposed by incorporating NAS methods that have led to the discovery of NN designs tailored to edge devices, enabling efficient inference without comprising accuracy. Notably, models such as MNASNet [213], MobileNet [90], and EfficientNet [29] are the results of advanced NAS methods. Furthermore, the pursuit of NN inference efficiency has opened the door to investigating more design prospects by (i) unleashing the potential of NN dynamicity to operate in an input-adaptive manner [82] and (ii) including HW features in a co-design manner to get the best from both worlds by scaling neural networks and hardware features [43]. The right side of Figure 1.2 reports the latest statistics on the growing popularity of each design approach to emphasize the role each plays in stretching the design landscape of NNss within the era of Edge AI.

1.3 Chasing Efficiency in the Era of Edge AI

Chasing computation and energy efficiency for Edge AI has recently been the center of attention due to the surge in demand for real-time AI applications in resource-constrained environments. Thus, many optimization approaches are being thoroughly studied and carefully adjusted to accommodate the need for new emerging applications, neural networks, and hardware paradigms. One prominent optimization strategy is CMOS [2] technology scaling to enable integrating smaller and more energy-efficient transistors onto processing units of compact chips that can serve AI accelerators [35].

As shown in Figure 1.3, the advancing speed of AI application workloads and their computing complexity, which doubles every two months, limits the potential of technology scaling, particularly regarding power consumption and thermal ma-

2. Complementary Metal-Oxide-Semiconductor is a technology for integrated circuits, known for its low power use and high noise resistance, used in most modern digital devices.

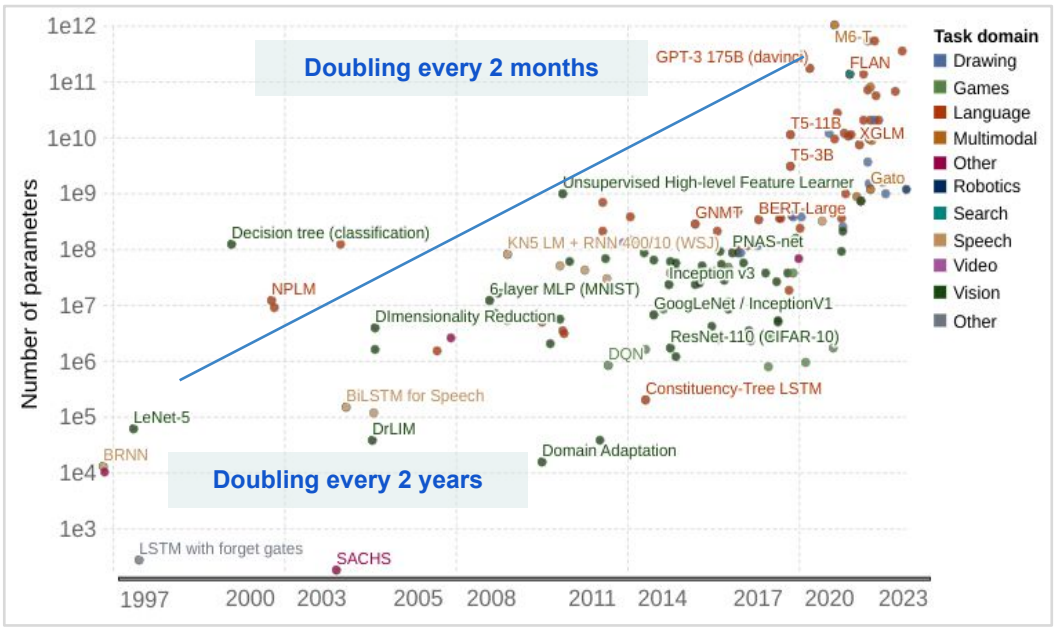


Figure 1.3 The growth rate of AI workloads in different application domains with their respective computing complexity is given in the y-axis as the number of parameters [2].

nagement. Furthermore, as transistors approach atomic limits, sustaining Moore's Law has become increasingly complex, slowing down technological advancements. As a result, optimization strategies are investigated from other perspectives – notably, from application and hardware perspectives. This thesis aims to show how the interplay between both perspectives can be leveraged to enhance efficiency in the era of Edge AI.

1.4 Edge AI Optimization Techniques

1.4.1 Software-level Optimizations

Software-level optimizations for neural networks on edge computing devices are instrumental in achieving efficient and cost-effective inference. These optimizations mainly focus on tailoring the neural network architecture and operations to minimize computational complexity while maintaining task performance. Given the strong correlation between NN model size and computational complexity, existing optimization strategies focus on providing lightweight models via *compression*. Model compression has been widely explored in literature. One widely adopted is *pruning*, which involves retrieving and removing less important weights and neurons from a NN [245, 64, 272]. Pruning is highly applied to CNN by removing specific weights (fine-grain pruning) or entire convolutional filters (coarse-grain pruning). Another technique is *quantization*, which reduces the precision of input activations and weights from 32-bit floating-point to lower-bit fixed-point representations (16-bit or 8-bit) [221, 243, 178]. *Knowledge distillation* is another compression method for NN in which a lightweight student model is trained to mimic the behavior and predictions of a large teacher model [247, 76]. Additionally,

Parameter sharing across multiple layers of NN is effective in vision-related tasks through capturing similar patterns or features across different spatial locations [181]. *Matrix factorization* offers another opportunity for model compression by decomposing a large weight matrix into smaller matrices with lower rank [231].

Compression techniques can be applied directly to a fixed design of neural networks without changing their neural architectures. Nevertheless, re-designing lightweight NN by choosing less compute-demanding operations and/or low values for neural layers depth and width can also be achieved via *Neural Architecture Search* [229, 252, 173]. NAS approaches have contributed to the realization of compact NN models such as MobileNet-v3 [89] and EfficientNet [215] by adjusting the internal neural architectural configurations to reduce model size while preserving feature representation power. Compression and NAS techniques can collaboratively contribute to facilitating the deployment of NN on edge devices by minimizing computation and memory demands that strongly impact the latency and energy consumption of real-time AI applications.

1.4.2 Hardware-level Optimizations

The hardware era has witnessed several computing devices emerging from general-purpose to specialized accelerators. AI applications are often deployed in real-time and critical environments, necessitating fast execution to return inference results within a strict delay. This has made computation acceleration a priority within the hardware community. AI has taken a large amount of the Hardware market, as shown in the left of Figure 1.4 with an estimated 227 billion dollars by 2032. The breakdown in the right of Figure 1.4 emphasizes the importance of processor scaling, depicting up to 56.29% of investment in 2022.

Edge computing devices are battery-powered and operate under a limited power budget. A reasonable use of computing power is needed to prevent dysfunctional or system failure in critical situations (e.g., autonomous cars). Thus, execution latency and power consumption are significant for Edge AI systems.

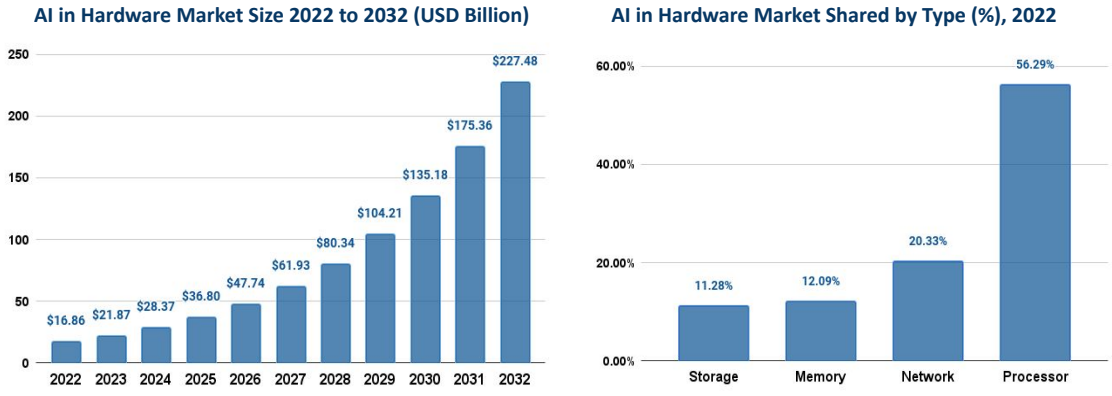


Figure 1.4 Left: The expected growth of the global AI in hardware market size from 2022-2032. **Right**: Breakdown of AI in HW market size per hardware type.

Hardware optimizations are mostly focused maximizing computing capabilities for a *fast inference* and minimizing power consumption for *energy-efficient inference*. Conventional HW accelerators play a pivotal role in achieving this in-

terplay. Graphic Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs), and Application-Specific Integrated Circuits (ASICs) have been improved through the recent decades to balance flexibility and efficiency. These accelerators leverage computing parallelism and optimized instruction sets for deep learning operations – matrix-matrix multiply - to maximize resource utilization and speed computations via specialized routines.

Specialized accelerators, such as FPGAs and ASICs, are primarily based on a systolic array, a grid of processing elements (PEs) interconnected in a regular structure for data sharing. These grids can be programmable, allowing for fine-grained computing parallelism and data reuse to optimize matrix-matrix multiplication, the core operation in neural networks.

General-purpose accelerators (GP), such as CPU and GPU, are built upon fixed hardware micro-architectures that, unlike FPGAs, don't allow for hardware reconfigurability. Instead, they encompass a grid of cores with parallel processing units that share cache memory for data communication and synchronization. Commodity edge GPUs often integrate specialized tensor cores optimized for matrix-matrix multiply. Although general-purpose accelerators are more flexible regarding programmability and can perform well for different operations, they're rigid and closed HW architecture for further design tuning (e.g., computation parallelism).

Unlike monolithic accelerators, heterogeneous MPSoCs (Multi-Processor Systems on Chips) are designed to embed different accelerators within the same die to accommodate the computing requirements of hybrid types of NN operations. Each computing unit (CU) of the MPSoC is optimized for distinct computational tasks [111]. For instance, commercial MPSoCs, such as the Nvidia Jetson series: AGX Xavier [162], TX2 [160], and Nano [163], as well as from other HW vendors such as Tesla FSD [211], and Intel Movidius Myriad 2 [161] have successfully integrated a variety of proven hardware computing units (e.g., CPU, GPU, and NPU) and industrial IPs on a single chip to achieve said purpose. This heterogeneity ensures that each phase of the inference pipeline – from data preprocessing to the final output – can be *mapped* to the most suitable CU, thereby maximizing efficiency and effectively using the MPSoC resources [22]. Furthermore, this approach accelerates inference speedup and maximizes energy efficiency by mapping computations to less energy-demanding CUs. As neural networks grow in complexity and ubiquity, heterogeneous MPSoCs play a crucial role in the speedup-energy balance.

Another way to reduce the energy consumption in edge computing systems is by leveraging Dynamic Voltage and Frequency Scaling. DVFS is a power management technique that adjusts the supply voltage and the operating clock frequency. It serves as a control knob to balance execution time and power consumption by adjusting the speed of computations and data transmission. The rationale behind DVFS for AI-based applications is the fluctuating computational demands observed in NN workloads. The NN inference pipeline encompasses hybrid operations with distinct computational and memory demands. Compute-bound operations typically require high clock frequencies on the computing unit, while memory-bound operations can benefit from optimizing the clock frequency of the DRAM. Power consumption is proportional to clock frequency and square of voltage. The quadratic correlation between power and voltage is crucial as it yields significant power gains. However, since frequency is typically proportional to voltage, dynamic voltage scaling is closely related to frequency scaling. For instance, higher frequen-

cies necessitate higher voltages and vice versa. Consequently, many accelerators offer mechanisms to employ DVFS by adjusting the clock frequencies of computing units and memory controllers. As NNs become more intricate and expansive, utilizing DVFS is essential to ensure efficient inference without compromising the device's battery life or thermal limits.

Finally, as we're approaching the end of Moore's law, physical and thermal limitations of existing CMOS technologies have pushed the hardware community to investigate unconventional computing paradigms beyond the typical Von Neumann architecture. Specifically, alternatives to conventional and monolithic accelerators are expected to be served by emerging neuromorphic, photonics, and quantum computing paradigms. Inspired by the human brain system, neuromorphic computing aims to emulate neural structures and operations, providing energy-efficient processing capabilities tailored for AI tasks like pattern recognition [110]. Quantum computing, which leverages the principles of quantum mechanics, is expected to accelerate intricate AI algorithms that are infeasible for classical computing systems [74]. Meanwhile, photonic computing employs light-based signals instead of electronic ones, offering rapid data transmission, reduced energy consumption, and eco-friendly systems – pivotal for real-time AI applications on edge systems [227]. While these innovative computing paradigms are in their early stages, they foreshadow a groundbreaking shift in Edge AI, paving the way for enhanced computational capabilities that align seamlessly with the needs of future AI applications (e.g., Multimodal Large Language Models).

1.5 Winning the 'Performance-Efficiency' Lottery Ticket

As the fields of neural networks and hardware systems progress rapidly at different rates, there's a predominant focus on optimizing each domain independently. This singular approach often ignores the critical significance of integrating, unifying, and harmonizing advancements from both fields. Existing works must bridge the gap between NN and HW innovations to unlock further potential breakthroughs from joining efforts. More precisely, the architectural design of neural networks plays a critical role in the tug-of-war between performance tasks and inference efficiency. If not carefully tuned, even the most advanced optimizations at the software or hardware levels might not yield the desired outcomes when applied to ill-designed neural networks. Regardless of the software or hardware innovations, if the foundational neural network design is flawed, these enhancements could fail to achieve their potential benefits. Conversely, while delegating more emphasis on tuning and refining the design of neural networks via handcrafted or automated approaches (e.g., NAS or HW-aware NAS) is laudable, It's neither efficient nor effective when applied solely. When Auto-design approaches are employed in isolation, without incorporating the other optimization levels mentioned previously, their results are often sub-optimal. Thus, a holistic, hierarchical, and cross-level approach that leverages various optimization strategies at different levels is expected to yield more robust and optimal results.

On the other hand, incorporating many factors into the design space exploration for NN and HW optimization often yields an exponential search space explosion. Exploring such a joint large search space employing traditional approaches of *train-and-deploy* is inefficient, time-consuming, and resource-intensive. A strategy

to accelerate the evaluation and search process to explore as many design options as possible is essential. One way to achieve this is using Machine Learning (ML) based performance estimators. By employing prediction models, we can significantly accelerate the exploration process, making it possible to quickly identify promising design candidates without exhaustive training, testing, and deploying each one. This approach falls under the umbrella of *ML for ML* or *ML4ML* to speed up the NN design, making the pursuit of the most efficient and accurate NN designs economically viable and technically feasible.

1.6 Thesis Structure and Contributions

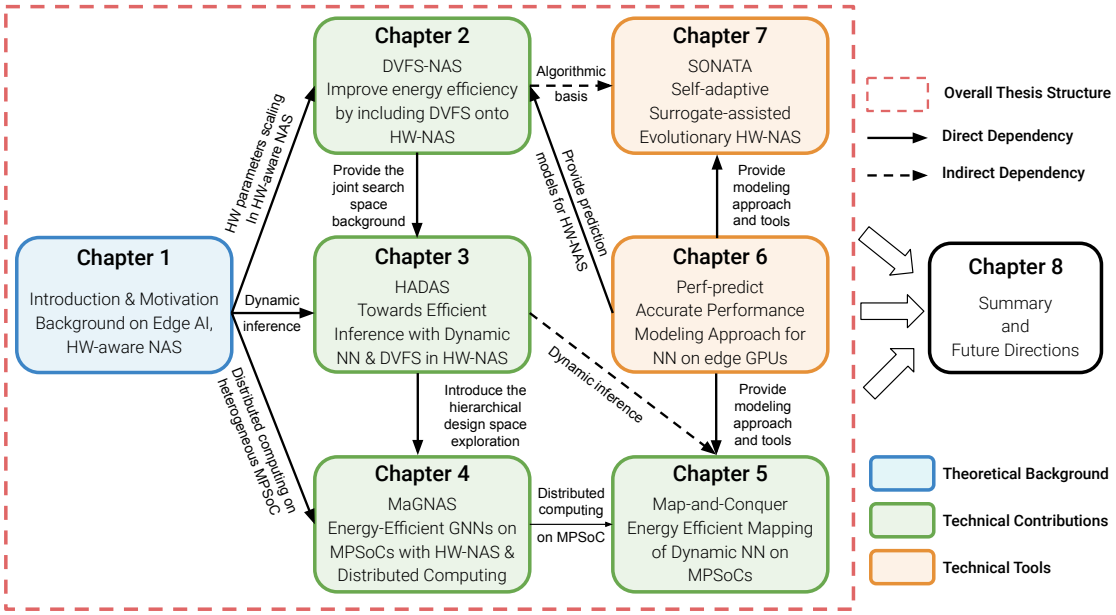


Figure 1.5 Thesis structure and dependencies between chapters

This thesis aims to mitigate the disparity observed in literature by delving into the interdisciplinary fields of multi-objective optimization, surrogate modeling, HW-aware NAS, and software and hardware optimizations. We focus on expanding the typical HW-aware NAS frameworks by leveraging various optimizations to support the ultimate objective of Edge AI – identifying the optimal Nash equilibrium between accuracy, execution speed, and energy efficiency to promote more sustainable computing. As depicted in Figure 1.5, this thesis contains the following contributions and novelties in the realm of Edge AI optimization:

— **Chapter 2: Dynamic Clock Frequency Scaling (DVFS) in HW-aware NAS** In [26, 27, 25], we extend the typical HW-aware NAS framework by including Dynamic Clock Frequency Scaling (DVFS) features as re-configurable hardware parameters on edge GPU MPSoCs. We showcase the importance of tailoring the design of NN to different DVFS policies and vice-versa. Consequently, we construct a joint search space of neural networks built upon the supernet structure of AttentiveNAS [220] and DVFS parameters by varying the clock frequencies. We propose a co-optimization framework to explore the predefined joint search space. Our proposed approach has been validated on

three datasets for Image classification, namely CIFAR-10, CIFAR-100, and ImageNet-1k, using the Jetson AGX Xavier edge GPU from NVIDIA. Compared to the default DVFS configuration (`MAXN`), evaluation results have shown up to ∼53% energy gains on the native Pytorch framework and a latency speedup of ∼81% and power saving of ∼61% with TensorRT.

— **Chapter 3**: **Hardware-aware Dynamic Neural Architecture Search**
In [21], we explore the prospect of dynamic neural networks (DyNN) through our framework, HADAS, a versatile and comprehensive HW-aware NAS framework that jointly optimizes static and dynamic components in NNs to balance the performance-efficiency tradeoff. We leverage the *early exit* dynamic inference technique to terminate computation earlier when the output can be correctly predicted. We make the design of the DyNN fully searchable by expanding the existing infrastructure of HW-aware NAS and supernets and constructing a novel joint search space for: (*i*) static backbone NN design, (*ii*) dynamic early-exit branches, and (*iii*) DVFS configurations. Then, we contribute an innovative co-optimization framework by hierarchizing the design exploration process of static and dynamic components using two dedicated optimization engines. We validate our approach using the CIFAR-100 dataset on various hardware configurations: ARM, Denver CPUs, Volta, and Pascal GPUs from the NVIDIA Jetson AGX Xavier and TX2. Evaluation results have seen the superiority of HADAS by providing up to ∼57% while sustaining an acceptable level of accuracy.

— **Chapter 4**: **Hardware-aware NAS for GNN on Heterogeneous MPSoC**
In [165], we delve into the world of Graph Neural Networks (GNN) on commodity edge heterogeneous MPSoCs. Specifically, we target the case of vision GNNs that have shown impressive results in recognizing visual patterns by transforming images into graphs of pixels and learning contextual information from neighboring graph nodes. We propose an end-to-end framework, *MaGNAS* from GNN design optimization to workloads mapping on MPSoC. We first construct and design novel supernet structures for GNNs by varying graph operations such as *aggregation* and *combination*. We provide a detailed and comprehensive system model for GNN workload characterization and mapping on heterogeneous MPSoC in a distributed, pipelined fashion. Then, we implement a hierarchical design space exploration to optimize the design of GNNs jointly with their workload mappings on the heterogeneous computing units of the MPSoC. We validate our proposed framework on various datasets for image recognition, namely CIFAR-10, CIFAR-100, Tiny-ImageNet, and Oxford-Flowers. As hardware settings, we employ (*i*) the NVIDIA Jetson AGX Xavier as a real MPSoC and (*ii*) MAESTRO, a cost model to simulate the case of an MPSoC by varying dataflows. Evaluation results have demonstrated the effectiveness of our approach by providing ∼1.57× latency speedup, ∼3.38× energy efficiency for several vision datasets while sustaining less than 0.11% accuracy reduction from SOTA models.

— **Chapter 5**: **Mapping of Dynamic Neural Nets on Heterogeneous MPSoC**
In [22], we research the potential of distributed and parallel computing for dyNN on heterogeneous MPSoCs. One way to enable the dynamic and collaborative usage of the computing units within an MPSoC is through dyNN. Specifically, by employing an early exit scheme, computing units will be acti-

vated according to the complexity of the input data. Within this context, we introduce *Map-and-Conquer*, an innovative framework designed to efficiently map the computing stages of DyNNs on heterogeneous MPSoCs in a parallel pipeline manner. Our methodology first determines the best partitioning strategy for the DyNN across its 'width' dimension, allowing for concurrent and parallel deployment of DyNN inference blocks on various computational units. We validate our approach on Transformers and CNN models on the CIFAR-100 dataset. We employ a real MPSoC from NVIDIA, Jetson AGX Xavier, equipped with a CPU, GPU, and Deep Learning Accelerator (DLA) as a hardware setting. Our evaluation results have shown that our dynamic configurations are 2.1x more energy-conserving than GPU-only setups and experience 1.7x reduced latency than DLA-only configurations.

— **Chapter 6**: **Performances Modeling of Neural Networks on Edge GPUs**
In [23, 24], we contribute to the concept of *ML4ML* by demonstrating how ML-based methods can be used to reduce the complexity of HW-aware NAS. One critical challenge of HW-aware NAS is performance evaluation. This process is time-consuming, necessitating the execution of an entire pipeline from model design to deployment and many rounds of performance measurements. To accelerate this step, performance prediction models can be used instead to provide quick estimations within microseconds. Within this scope, we propose a comprehensive performance analysis and characterization of CNN inference workloads on edge GPUs. We study the correlation between diverse CNN features and several performance metrics (i.e., latency, power consumption, and memory usage). We elaborate an end-to-end modeling methodology using ML-based methods to fit performance prediction models. We validate our approach on SOTA and synthetic CNN architectures on three edge GPUs from the NVIDIA Jetson series: AGX Xavier, TX2, and Nano. Our prediction models have shown an average error of ∼11%, ∼6%, and ∼8% for latency, power, and memory usage estimations, respectively.

— **Chapter 7**: **Surrogate-assisted Self-adaptive Evolution for HW-aware NAS**
In [**?**], we extend the concept of *ML4ML* to enhance the efficiency of evolutionary search algorithms in HW-aware NAS. Most existing HW-aware NAS methods, such as evolutionary algorithms, are built upon multi-objective optimization approaches. However, these algorithms heavily rely on randomness without established reasoning on the importance of NN design parameters on the optimization objectives. A priori knowledge of the importance of design parameters helps focus on the most rewarding ones during the search and thus wisely use the optimization budget. Furthermore, the massive amount of data generated during the search can be exploited to derive such knowledge. Given these observations, we propose `SONATA`, a self-adaptive evolution for multi-objective HW-aware NAS frameworks. We design self-adaptive evolution operators guided by the importance of NN design parameters. We leverage ML-based methods to progressively learn the importance of design parameters from the data generated during the search. An extensive evaluation of multiple NAS search spaces and edge devices has shown that our approach improves upon the baseline with an accuracy improvement up to ∼0.25%and latency/energy gains up to∼2.42x.

Chapitre 2

DVFS-NAS: Dynamic Clock Frequency Scaling for Hardware-aware Neural Architecture Search on Edge GPUs

2.1 Introduction

Deep Neural Networks (DNN) and hardware accelerators are leading forces for the recent observed progress in *Edge AI*. On the one hand, a new neural architectural paradigm is proposed each month, striving for more accuracy and efficiency. On the other hand, the hardware market has shifted towards designing devices that ensure flexibility and generality for less energy demands while satisfying the user experience with less latency. Thus, reconfigurability has been standardized across general-purpose hardware platforms such as GP-GPUs through cores and clock frequency scaling to emulate different performance and energy efficiency levels using the same device. However, recently, DNN architectures are becoming increasingly complex and hardware resource-demanding. As shown in Figure 1.3, the growth rate of DNN model complexity (i.e., "every two months") exceeds the hardware scaling capabilities of Moore's law (i.e., "every two years"). Thus, exploring new optimization dimensions in existing hardware devices is mandatory to meet the computation demands of DNN models.

Particularly, when DNN models are implemented on resource-constrained systems (e.g., edge computing), it becomes inevitable to meticulously optimize them to strike the optimal balance between accuracy and execution latency, as well as energy efficiency. In order to address this particular difficulty, researchers have introduced Hardware-aware Neural Architecture Search *(HW-aware NAS)* as a new*AutoML* paradigm, targeting edge systems [14]. HW-aware NAS incorporates hardware efficiency as an additional optimization objective during the neural architecture design space exploration.
Nevertheless, hardware efficiency is contingent upon the interplay between the architectural design of the neural network and the settings of the hardware itself [157, 244]. Most existing works on Hardware-aware NAS primarily focus on optimizing neural networks without considering the hardware accelerator's reconfigurability. As elucidated in the study conducted by [100], the aforementioned technique exhibits sub-optimality due to the limited exploration potential of the HW-aware NAS when confined to a predetermined hardware design. Therefore, including the hardware design space has made it viable to identify customized DNNs for any configuration and vice versa. We refer to this joint exploration of both search spaces as *the co-optimization*.

In this chapter, we propose to include Dynamic Clock Frequency Scaling (`DVFS`) features in the HW-aware NAS process on commodity edge GPUs. We aim to

showcase the importance of tailoring the design of NN to different DVFS policies and vice-versa. We introduce *DVFS-NAS*, an evolutionary-based co-optimization framework to explore the joint search space of DNN and DVFS to find Pareto optimal configurations that achieve the best balance between accuracy, latency, and power consumption. We validate our approach on three datasets for Image classification, namely CIFAR-10, CIFAR-100, and ImageNet-1k, using the Jetson AGX Xavier edge GPU from NVIDIA. Compared to the default DVFS configuration (MAXN), evaluation results have shown up to ∼53% energy gains on the native Pytorch framework and a latency speedup of ∼81% and power saving of ∼61% using a high-performance SDK such as TensorRT. Contributions and results of this chapter have been published in:

— [26] **Halima Bouzidi**, Hamza Ouarnoughi, El-Ghazali Talbi, Abdessamad Ait El Cadi, and Smail Niar, 2021, Evolutionary-based Optimization of Hardware Configurations for DNN on Edge GPUs. In "Proceedings of the 8th International Conference on Metaheuristics and Nature Inspired Computing, (META)" 2021.

— [27] **Halima Bouzidi**, Hamza Ouarnoughi, El-Ghazali Talbi, Abdessamad Ait El Cadi, and Smail Niar. Evolutionary-Based Co-optimization of DNN and Hardware Configurations on Edge GPU. In: Optimization and Learning. (OLA) 2022. Communications in Computer and Information Science, Springer Nature.

— [25] **Halima Bouzidi**, Hamza Ouarnoughi, Smail Niar, El-Ghazali Talbi, and Abdessamad Ait El Cadi. "Co-Optimization of DNN and Hardware Configurations on Edge GPUs," Euromicro Conference on Digital System Design (DSD) 2022.

2.2 Related Works

The typical HW-aware NAS involves exploring the neural network search space while considering hardware metrics. This way, the explored design space is limited to the variations of the *neural networks*. However, accounting for the hardware reconfigurability, another dimension can compound the design space, and the HW-aware NAS search space becomes less narrow. Considering the former case, a DNN model that fails to meet the resource limitations will be pruned by the search strategy even if it provides high accuracy. Enlarging the hardware search space allows for finding tailor-made HW designs for the pruned-accurate DNNs [100]. As a result, co-optimizing both worlds, i.e. HW and SW, can provide better trade-offs between accuracy and execution efficiency.
Along this line, recent studies [132, 128, 99, 40, 46, 129, 269, 175] have addressed the *DNN-HW co-optimization* challenge, which comprises optimizing DNN architectures in conjunction with hardware configurations. Multiple DNN-HW pairs are generated and evaluated during exploration to choose the optimal ones. Aso shown in Figure 2.1, in the literature, we distinguish two DNN-HW co-design strategies:

— ***(i)*** *Joint search* – in which DNN and HW configurations are searchable together. That is at each search iteration, a pair of DNN-HW is created and evaluated. The DNN is trained and validated on the target dataset and then deployed on the HW device to get performance measurements [141, 5, 30].

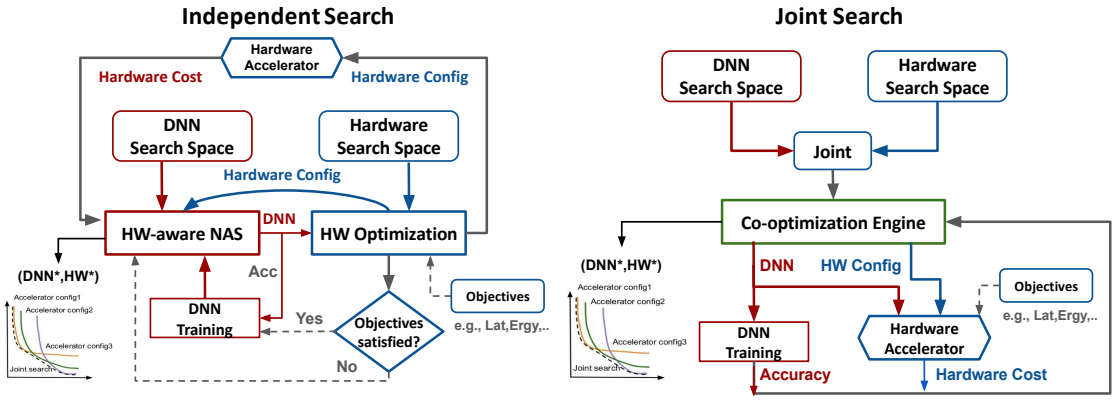


Figure 2.1 DNN-HW co-design: *Independent* Vs. *Joint* search strategies.

— **(ii)** *Independent search* – in which DNNs are first trained and validated, then the most promising configurations are elite for further HW optimization.

Table 2.1 Comparison between related works on the DNN-HW co-optimization.

DNN-HW Co-optim.	Neural Architecture Search				Hardware Deployement Optimization				Co-optim. Strategy
	Search Algorithm	Search Space	Fitness Objective	Fitness Evaluation	HW Architecture	Search Space (HW parameters)	Fitness Objective	Fitness Evaluation	
[83]	Gradient	Macro	Acc	Train-val	FPGA	Parallelism, Quant.	Lat, HRU	Analytical	Joint
[141]	RL	Macro	Acc	Train-val	FPGA	Tiling, Partitioning	Lat	Analytical	Indep.
[132]	EA	Micro	Acc	Train-val	ASIC	#PE	EDP	Simulator	Joint
[5]	RL	Micro	Acc	Train-val	FPGA	Parallelism	Lat, Area	Analytical	Indep.
[263]	Gradient	Macro	Acc	Train-val	FPGA, ASIC	#PE, Dataflow, Tiling	FPS, Lat, EDP	Simulator	Joint
[128]	Gradient	Micro	Acc	Train-val	GPU, FPGA	Parallelism, Quant.	Lat, HRU	Analytical	Joint
[99]	RL	Macro	Acc	Train-val	FPGA	#PE, Tiling, BW	Lat, LUT	Analytical	Joint
[40]	RL	Micro	Acc	ML-based	ASIC	#PE, Dataflow	Lat, Ergy	ML-based	Joint
[46]	Gradient	Micro	Acc	Train-val	ASIC	#PE, Dataflow	Lat, Ergy, Area	Simulator	Joint
[129]	Gradient	Micro	Acc	Train-val	FPGA	Parallelism, Buffering	LAT, LUT, DSP	Analytical	Joint
[133]	Gradient	Micro	Acc	Train-val	TPU, ASIC	#PE, Mapping	Acc, EDP	Simulator	Indep.
[269]	RL	Micro	Acc	ML-based	TPU	#PE, #SIMD units	Lat, Area	ML-based	Joint
[175]	EA	Macro	Acc, Size	Train-val	ASIC	Approx. multipliers	Ergy	Analytical	Joint

RL: Reinforcement learning, **EA**: Evolutionary algorithm, **EA**: Processing element. **Acc**: Accuracy, **Size**: DNN Model size, **Lat**: Latency, **Ergy**: Energy, **FPS**: Frame rate per second, **EDP**: Energy-delay product. **HRU**: Hardware resource utilization (%). **Quant**: Quantization. **Indep.**: Independent.

Although the second approach is faster, it is more prone to sub-optimality as the DNN and HW search engines act independently [189]. Thus, the two search engines must communicate and share their search results, updates, and insights to constantly adjust the overall co-optimization process.

In Table 2.1 we give a comparison between related works on DNN-HW co-optimization [1] Generally, these works can be breakdown onto two main optimization engines [100]:

1. *Neural Architecture Search (NAS)*: This engine defines how the DNN architecture design is optimized using NAS. The components of the NAS can be defined as following:
   (a) DNN search space: This comprises the possibilities of DNN architectures to be explored. Existing NAS search spaces are defined on the basis of hyperparameters granularity, for instance, on the levels Macro-architecture, Micro-architecture, Hierarchical, or Supernetworks as shown in Figure 2.2.
   (b) Fitness objectives and evaluation: This measures how well the DNN performs on the target dataset (e.g., accuracy for image classification).

1. We note that this comparison involve related works published before the year of 2022.

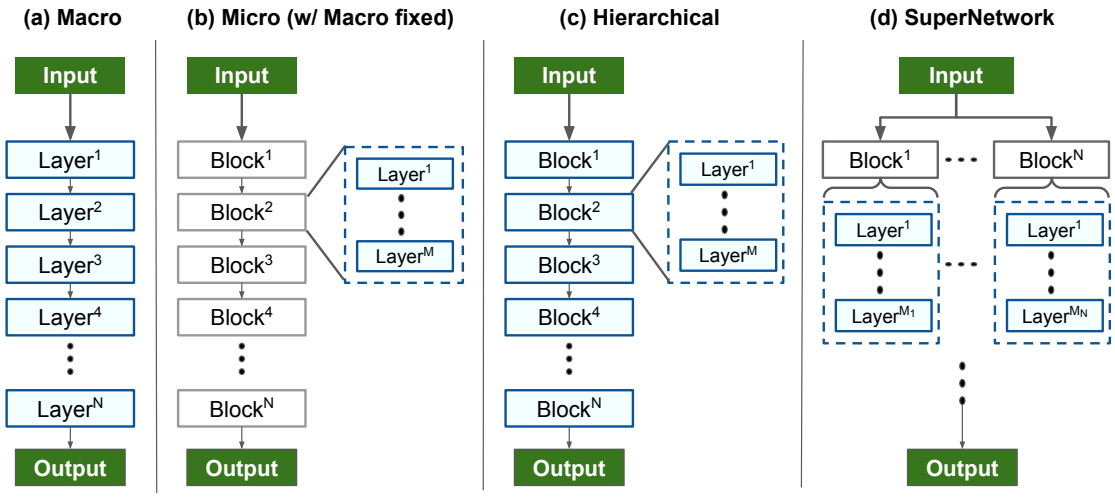


Figure 2.2 Different variants of search spaces in NAS. In blue searched blocks and in white fixed blocks. **a)** *Macro* in which the entire topology and layers of DNN are searchable. **b)** *Micro* in which the topology of the DNN is fixed to a set of blocks and the micro-architecture of these blocks is searchable. **c)** *Hierarchical* in which the macro and micro-architectures of DNN can be searched in an hierarchical way. **d)** *Supernetwork* in which a hypernetwork of DNNs is defined by varying blocks and layers. [14]

This metric is typically evaluated using training-and-validation or ML-based surrogate models.

(c) Search algorithm: Which orchestrates the exploration of the space exploration using, for example, Gradient-based optimization, reinforcement learning, or evolutionary algorithms.

2. *Hardware Deployment Optimization*: This engine defines how the Hardware accelerator is optimized and tuned to fit the requirements of the sampled DNN models by the NAS. It comprises three main components:

   (a) Hardware search space: This comprises hardware-related parameters used to design, tune, and optimize the hardware, such as computation parallelism, tiling, number of PE, dataflow, data buffers, mapping, SIMD units, approximate multipliers, and data precision.

   (b) Fitness objectives and evaluation: Depending on the application requirements, the following metrics are usually used to assess the DNN efficiency when deployed on the HW accelerator: Latency, energy, FPS as general high-level metrics, and hardware resource utilization, EDP, Area, LUT, and DSP as design-specific low-level metrics for FPGA, ASIC, and TPU devices. These objectives can be measured using analytical approaches, simulation methods, or ML-based models.

   (c) Search algorithm: This defines how the HW search space will be explored, and it is typically similar to the one used in the NAS engine.

The key concept of DNN-HW co-design works is the *co-optimization strategy* that defines how the different optimization engines communicate and update their process, i.e., in a fully joint or independent manner.

Whereas these methods target reconfigurable devices to search for appropriate HW design, we aim to explore the prospect of tuning HW-related configurations

on general-purpose devices such as edge GPUs. Dynamic clock frequency scaling is one way to tune the computing capability of edge GPUs to balance latency and energy. Finding a `DVFS` policy that achieves an optimal latency-energy balance depends highly on the DNN computation and memory requirements, which also depend on the DNN model design. Thus, we extend related works by investigating this direction and studying how the interplay between DNN architecture design and `DVFS` configurations impacts the performance-efficiency tradeoff.

2.3 Motivational Example

Figure 2.3 illustrates our motivation. This figure shows that the interplay between DNN and HW configurations yields different accuracy and hardware efficiency tradeoffs. The concept of *hardware efficiency* pertains to the balance between the latency and power consumption entailed by the DNN inference. One key parameter impacting this balance is the choice of the clock frequency configuration. Opportunely, recent devices proposed by NVIDIA allow for the adjustment of clock frequencies on the CPU, GPU, and External Memory Controller (EMC) to balance latency and power. We refer to this hardware reconfiguration by `DVFS` as a short for Dynamic Voltage and Clock Frequency Scaling.

The left of Figure 2.3 reports the results of a HW-aware NAS conducted on the NVIDIA Jetson AGX edge GPU under a fixed `DVFS` configuration that gives the highest performances. Specifically, we use `MAXN` in which clock frequencies on the CPU, GPU, and EMC are set to their maximum values and all CPU/GPU cores are active. The explored DNN models are sampled from the pretrained supernet from AttentiveNAS [220]. Each point represents an explored DNN configuration as a sampled subnet from the supernet search space. The x-axis- and y-axis represent the latency and power consumption directly measured on the edge GPU. The color of the points designates the TOP-1 accuracy.

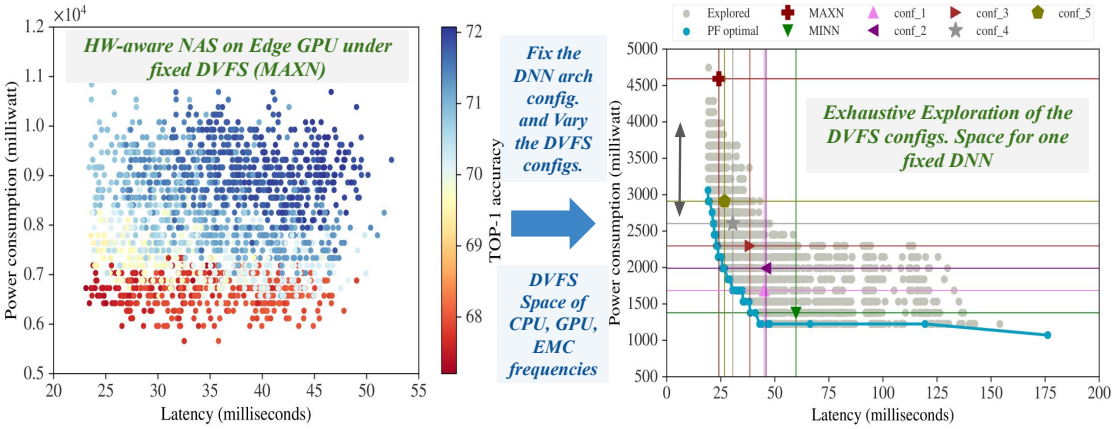


Figure 2.3 *On the left*: The results of performing an HW-aware NAS under fixed edge GPU's hardware configuration. *On the right*: The results of optimizing edge GPU's hardware configuration for a fixed DNN model.

As shown, different accuracy and hardware efficiency for each explored DNN model, which is an expected result from the HW-aware NAS.

The right of Figure 2.3 demonstrates that the HW efficiency of a single DNN configuration varies when varying the `DVFS` policy. This figure gives the results of

an exhaustive exploration of CPU, GPU, and EMC clock frequency configurations for a randomly sampled DNN model from the former HW-aware NAS. To showcase the impact of the `DVFS` configuration on the overall hardware efficiency of the DNN, we compare the obtained results with the predefined default `DVFS` configurations proposed by NVIDIA. In this figure, `MAXN` (resp. `MINN`) is the NVIDIA Jetson AGX configuration with the highest (resp. the lowest) clock frequencies. `MAXN` (resp. `MINN`), in general, maximizes (resp. minimizes) the processing speed at the cost of a high (resp. low) power consumption. The other configurations (i.e., from conf_1 to conf_5) are proposed to achieve a tradeoff between `MAXN` and `MINN` [2]. Remarkably the Pareto front, marked in blue in Figure 2.3, is obtained by the exhaustive exploration. It identifies `DVFS` configurations that completely dominate NVIDIA's predefined default configurations. We note that the Pareto front does not contain any of NVIDIA's predefined configurations (`MAXN`, `MINN`, and conf_1 to conf_5). Furthermore, the non-dominated solutions in the Pareto front improve the default configurations of NVIDIA (i.e., `MAXN` and `MINN`) by up to 57% and 40% for power consumption and latency, respectively. These results show the necessity to further explore the `DVFS` configurations to enhance the hardware efficiency of the DNN inference on edge GPUS.

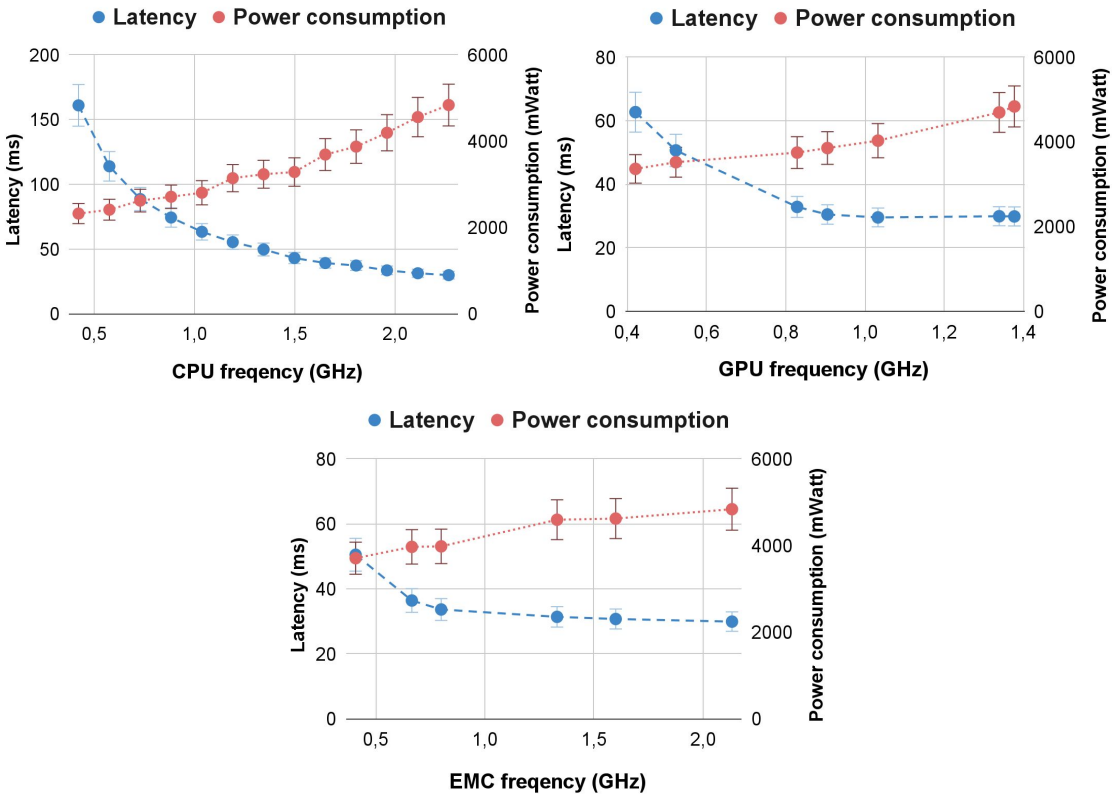


Figure 2.4 Impact of varying the CPU, GPU, and EMC frequency values on latency and power consumption of the DNN inference.

In order to get a deeper comprehension of the impact of adjusting the clock frequency, we delve into the findings shown in Figure 2.4. In this figure, we report the impact of varying the clock frequency of CPU, GPU, and EMC frequency, separately, on the inference latency and average power consumption. The experi-

2. Jetson developer kits and modules: https ://docs.nvidia.com/jetson/l4t/

mental procedure involves systematically varying one frequency at a time while keeping the remaining frequencies fixed at their highest values. For instance, the CPU frequency is varied while the GPU and EMC frequencies are held constant at their respective maximum values.
Taking as an example the GPU frequency, as shown in the left of Figure 2.4, the latency starts decreasing as the clock frequency increases from 400 MHz to 900MHz. However, after 900MHz the latency remains the same, but the power consumption keeps increasing drastically. Similar observations can be drawn from observing the CPU and EMC frequency variation. The reason behind this result is the workload requirements for the considered neural networks. For instance, if the inference is compute-bound, high values of the GPU frequency are preferable to operate at a high speed and accelerate computations. On the other hand, if the inference is memory-bound, then high CPU and EMC frequency values will help maintain the memory operations at the same pace as the GPU computation speed, which helps minimize the overall latency. Thus, a prior investigation of the inference workload requirements is crucial to determine which operating frequencies are needed to run the inference without wasting the overall power budget while sustaining an acceptable latency. For instance, from Figure 2.4, we remark that a frequency of 900 MHz gives the best trade-off between latency and power consumption. However, this value can be further adjusted by choosing adequate CPU and EMC frequencies. Furthermore, the optimal values of clock frequencies change according to the DNN workload. Thus, there is no optimal hardware configuration for all the DNNs.

In conclusion, the performances of a DNN model are determined by the interplay between the DNN architecture and the HW configuration (`DVFS`). However, understanding the impact of these factors is not straightforward, which motivates the co-optimization of DNN and operating clock frequency configurations.

2.4 Problem Statement

As discussed in the motivational example, the overall problem can be formulated as a multi-objective joint optimization problem in which we adjust both the Neural Network architecture and `DVFS` configurations for a better accuracy and hardware efficiency trade-off. As exploring the entire HW/DNN design space is time-consuming and labor-intensive regarding training, evaluation, and deployment, we need a rapid DNN/HW co-design space exploration strategy. Furthermore, while accelerating the co-design space exploration, the search framework must adequately provide a good approximation of the Pareto front (i.e., the best tradeoff between accuracy and hardware efficiency). Formally, considering both the DNN and `DVFS` configurations, the mathematical formulation of our co-optimization problem is as follows:

$$(dnn^*, \text{dvfs}^*) = \arg\min[(\mathcal{E}(dnn), \mathcal{L}(dnn, \text{dvfs}), \mathcal{P}(dnn, \text{dvfs})]^T \quad (2.1)$$

$$\text{s.t.} \quad \text{dvfs} = (clk_{cpu}, clk_{gpu}, clk_{emc}) \in (\mathcal{C}_{cpu} \times \mathcal{C}_{gpu} \times \mathcal{C}_{emc}) \ , \ dnn \in \mathcal{S}_{dnn} \quad (2.2)$$

Here dnn represents a DNN model characterized by the set of architectural parameters such as depth, number and size of kernels, and type of layers (See Section 2.5.1).

The variation of these parameters induces the realization of the DNN design search space, $\mathcal{S}_{dnn}$, that comprises variants of DNNs with different learning capacities and inference characteristics. dvfs represents a `DVFS` configuration defined by the combination of clock frequencies on the Central Processing Unit (CPU), Graphical Processing Unit (GPU), and External Memory Controller (EMC). The range values of each clock frequency are noted by $\mathcal{C}_{cpu}, \mathcal{C}_{gpu}, \mathcal{C}_{emc}$.

A DNN model *dnn* is evaluated regarding: (*i*) the inference prediction error $\mathcal{E}$ on a user-defined *dataset*. (*ii*) execution latency $\mathcal{L}$ (*iii*) power consumption $\mathcal{P}$. While the DNN learning capacity only impacts the prediction error, inference latency and power consumption are impacted by the DNN model computation complexity and the underlying `DVFS` configuration on the edge GPU. We note that the inference prediction error $\mathcal{E}$ is assessed by computing the TOP-1 error rate, which is the percentage of images from the validation set of the targeted task *dataset* for which the correct label is not the class label predicted by the DNN. Finally, given the multi-objective context of the co-optimization problem, a Pareto ranking strategy is used to rank the fitness of the explored combinations of (dnn, dvfs) using a non-dominated sorting algorithm to select the Pareto optimal combination as elite solutions. A (dnn^*, dvfs^*) is said to be Pareto-optimal if and only if for every objective $u_k \in U$:

$$u_k(dnn) \geq u(\text{dvfs}) \forall k, (dnn, \text{dvfs}) \text{ and } \exists j : u_j(dnn^*, \text{dvfs}^*) > u_j(dnn, \text{dvfs}) \quad (2.3)$$

$$\forall (dnn, \text{dvfs}) \neq (dnn^*, \text{dvfs}^*) \text{ and } U = \{\mathcal{E}, \mathcal{L}, \mathcal{P}\} \quad (2.4)$$

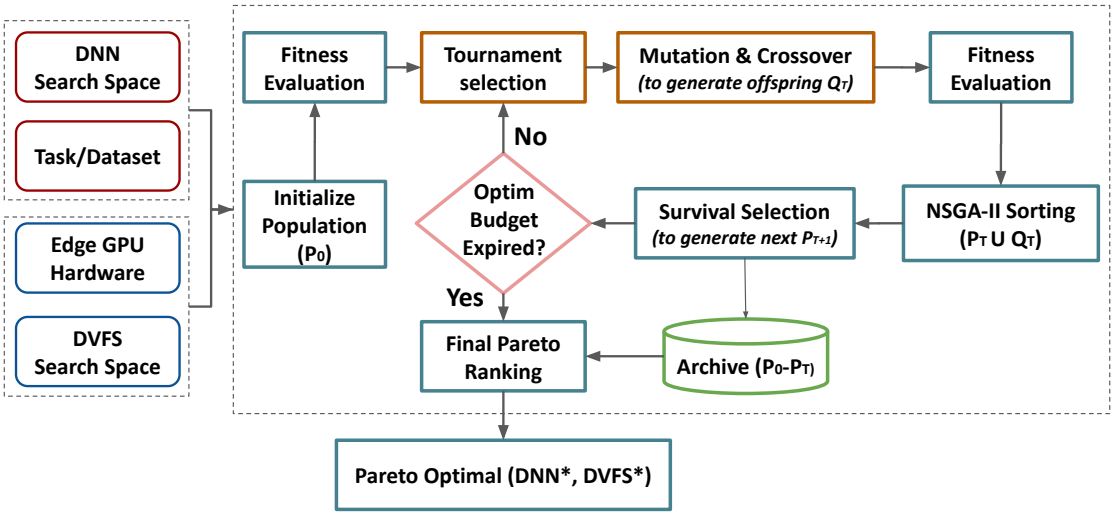


Figure 2.5 Overview on our proposed DVFS-NAS co-optimization approach.

2.5 Proposed Approach

We propose an evolutionary-based co-optimization strategy, where we search for both the optimal DNN architecture and `DVFS` configuration. We include the `DVFS` search space into the conventional DNN design space. Then, both configurations are explored using an evolutionary-based algorithm (NSGA-II). Figure 2.5 details our proposed co-optimization approach that comprises three main components:

2.5.1 Joint Search Space

By definition, the joint search space can be generalized to any DNN, task, dataset, and hardware accelerator. These four factors are considered as inputs in our co-optimization framework. In our case, we fix the task and dataset to image classification on the ImageNet-1k dataset. We design a joint search space that encompasses $\sim 3.5 \times 10^{14}$ (`DNN`$\times$`DVFS`) combination. We detail both search spaces in the following:

Table 2.2 The joint search space of DNN and hardware parameters

Decision variables	**DNN search space**					**Hardware search space**		
	Input resolution	**Width**	**Depth**	**Kernel size**	**Expand ratio**	**CPU frequency**	**GPU frequency**	**Memory frequency**
Values	[192, 288]	[16, 1984]	[1, 8]	[3, 5]	[1, 6]	[0.1, 2.3]	[0.1, 1.4]	[0.2, 2.1]
Cardinality	4	16	8	2	4	29	14	9

① **DNN search space**: The DNN search space comprises all the possible architectures. In our case, we employ the supernet approach in which a hypernetwork of networks is constructed and trained once via the weight-sharing approach [30]. In our case, we use the same supernet provided in [220] built on the FBNet [229] macro-architecture and composed of the inverted residual block *MBConv* introduced in MobileNet [186]. As shown in Figure 2.6, the micro-architecture of the supernet is searchable through varying the MBConv layers parameters such as channel width, depth, kernel size, and expansion ratio. Other meta-parameters, such as the input resolution, are also included. We detail each architectural parameter value and range in Tables 2.2 and 2.3. We note that the original supernet search space was designed to limit the complexity of the largest network to less than 2,000 MFLOPs. Overall, the search space complexity is equal to $\mathcal{O}(2.94\times10^{11})$ candidate neural architecture.

② **DVFS search space**: We extend the search space of the NAS by including the `DVFS` configurations. As explained in Section 2.3, the clock frequency of each hardware component among the CPU, GPU, and EMC plays a role in the execution efficiency, depending on the DNN workload requirements. Thus, we consider each of them an independent and searchable hardware parameter. As clock frequencies need to follow a domain range, the possible values are fixed by the hardware constructor to ensure the HW device's safe functioning. Thus, our search space is limited to the possible values of clock frequencies, which also depends on the edge GPU architecture. We provide more details on the clock frequency ranges and possible values in Table 2.2. We note that in all our experiments, we turn off the dynamic adjustment of clock frequencies and employ instead a static setting in which all frequencies are held constant to avoid the high variance in latency and power consumption measurements. Furthermore, we activated 50% of CPU cores while fully using the GPU. The reason is that we noticed that CPU cores do not have a high contribution to the DNN inference and remain in an IDLE state most of the time.

Table 2.3 Detailed overview on the DNN architecture search space. Each row denotes the search space of a fixed block of the DNN macro-architecture.

Block name	Channel width	Depth	Kernel size	Expansion ratio	SE
Conv	{16, 24}	-	3	-	-
MBConv-1	{16, 24}	{1,2}	{3, 5}	1	N
MBConv-2	{24, 32}	{3, 4, 5}	{3, 5}	{4, 5, 6}	N
MBConv-3	{32, 40}	{3, 4, 5, 6}	{3, 5}	{4, 5, 6}	Y
MBConv-4	{64, 72}	{3, 4, 5, 6}	{3, 5}	{4, 5, 6}	N
MBConv-5	{112,128}	{3, 4, 5, 6, 7, 8}	{3, 5}	{4, 5, 6}	Y
MBConv-6	{192, 200, 208, 216}	{3, 4, 5, 6, 7, 8}	{3, 5}	6	Y
MBConv-7	{216, 224}	{1, 2}	{3, 5}	6	Y
MBPool	{1792, 1984}	-	1	6	-
Input resolution	{192, 224, 256, 288}				

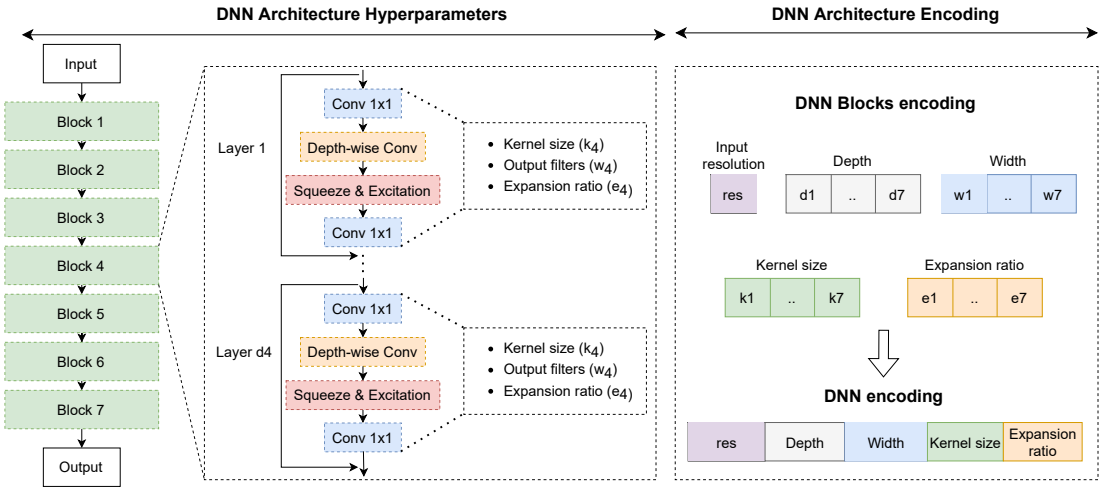


Figure 2.6 Details on the DNN search space encoding: A candidate DNN architecture is real-encoded using a single vector that comprises five sub-vectors depicting: input resolution, depth, width, kernel size, and expansion ratio of each block

2.5.2 Evolutionary Search Strategy

Given the complexity of the underlying joint search space, an effective search strategy is needed to efficiently explore the candidate configurations and retain the Pareto optimal ones. Among the existing search algorithms, we employ an evolutionary approach as it has been proven effective for the NP-hard NAS optimization problem [138]. Specifically, we use NSGA-II [51] that applies a fast, non-dominated sorting algorithm via Pareto ranking and a crowding distance as diversity measurement to select elite solutions for child population reproduction.

As shown in Figure 2.5, The algorithm started by randomly generating N (DNN, `DVFS`) combinations to construct the first population P_0. Then, a fitness evaluation is performed to obtain the values of the corresponding objectives (i.e., prediction error, latency, and power consumption). Afterward, a tournament selection [18] is employed to select a subset of promising (DNN, `DVFS`) configurations from P_T to generate the offspring population Q_T via *Mutation* and *Crossover*. Q_T then undergoes a fitness evaluation and a two-step NSGA-II sorting is applied on

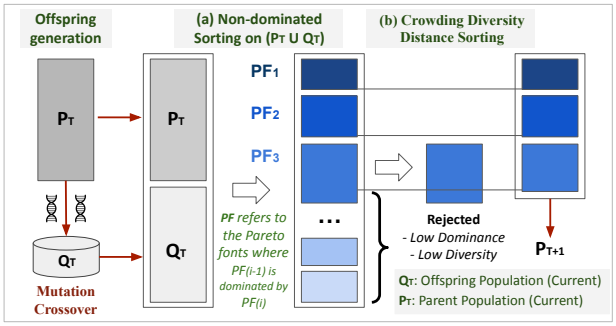
the $(P_T \cup Q_T)$ as detailed in Figure 2.7:

— (i) First, configurations in $(P_T \cup Q_T)$ are sorted according to their Pareto dominance using the non-dominated sorting algorithm that generated sets of Pareto fronts $PF_{1,\dots,m}$ in which PF_{i-1} dominates PF_i for $1 \geq i \leq m$.

— (ii) Second, configurations in $\bigcup_{i=1}^{m} PF_i$ are further sorted according to their crowding distance in which higher values depict better diversity [52].

Pareto fronts ranking and diversity scores are jointly used to retain a ratio of elite survival candidates solutions that create the next population P_{T+1}.

NSGA-II is expected to perform optimally and efficiently within a low optimization budget (e.g., number of generations). The parameters used for NSGA-II are detailed in Table 2.7. In our framework, the first population is initialized using the Latin HyperCube Sampling (LHS) method for a high variance of the search variables. We choose a high crossover probability of 80% to increase good candidate solutions' reproducibility. On the other hand, we fix a mutation probability to 30% to prevent the risk of losing traces of optimal candidate solutions. Crossover and mutation are chosen uniformly to balance both *exploration* and *exploitation*. For the optimization budget we fix the size of populations P_T to $N = 100$ and we run the search algorithm for a total of $G = 50$ generations.



NSGA-II Parameter	Value
Sampling	Random
Generations (G)	50
Population size (N)	100
Population init.	LHS
Mutation, prob.	Uniform, 30%
Crossover, prob.	Uniform, 80%
Elite Selection	Tournament, 25%

Figure 2.7 Left: NSGA-II two-step sorting process. The blue shades refer to the Pareto fronts (PF_i) from the non-dominated sorting algorithm in [52]. **Right**: NSGA-II meta-parameters used in our works.

2.5.3 Fitness Evaluation Strategy

The explored pairs of (`DNN`×`DVFS`) are evaluated regarding prediction accuracy on the targeted task and hardware efficiency of inference as detailed below:

— ***(i)*** The prediction accuracy is the most expensive part as it ultimately involves training and validation steps. We overcome the training problem by using a pretrained supernet and weight inheritance. Nevertheless, the validation step is still a bottleneck as it requires a Batch-Norm weights calibration on the training set [255] and validation on the test set of the large ImageNet-1k dataset. To mitigate this challenge, we employ a surrogate model for DNN accuracy prediction. Specifically, we reuse the surrogate model provided in [220], which is a Random Forest predictor [130] that takes the direct encoding of the DNN (See Figure 2.6) and output the estimated accuracy for

30 epochs training. These estimations are used during the search to accelerate the process. The obtained Pareto optimal `DNN` subnets are derived from the supernet and exhaustively evaluated on the ImageNet-1k validation set to get the accuracy when trained on longer epochs. We note that accuracy values obtained from the surrogate and pretrained supernet are highly correlated, reporting up to $\sim$ 0.88 Kendall-tau correlation rank coefficient as shown in Figure 2.8 from the resoluts reported in [220].

— ***(ii)*** The inference hardware efficiency (latency and power) is directly measured by deploying and executing the sampled `DNN` model on the hardware accelerator under the sampled `DVFS` configuration. The hardware evaluation is not a time-consuming step compared to the accuracy evaluation.

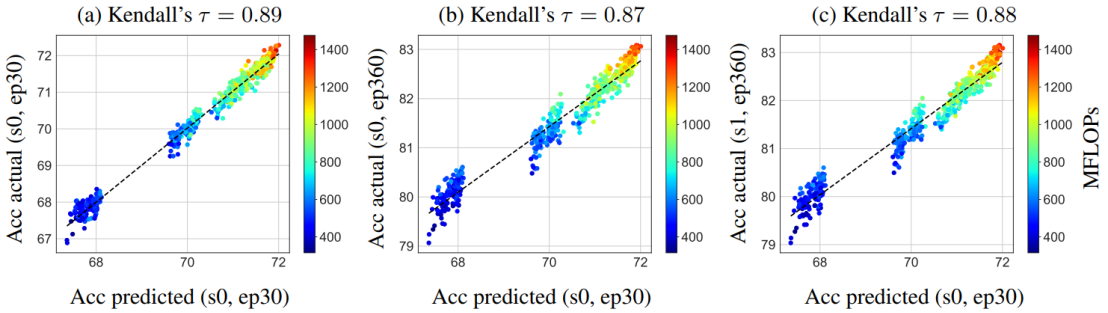


Figure 2.8 Rank correlation between estimated and actual accuracy. ep30 and ep360 denote 30 and 360 training epochs. s0 and s1 denote the random seeds [220]

2.6 Evaluation Methodology

In this section, we provide a comprehensive evaluation of our methodology. We first detail the experimental setup and then discuss and obtained results.

2.6.1 Experimental Setup

① **Task and Dataset**: We primarily target image classification on the ImageNet-1k [115] dataset that contains 14,197,122 annotated images categorized into 1000 classes. We also extend our evaluation by applying transfer learning to other image classification datasets such as CIFAR-10 [112] and CIFAR-100 [113]. All images are preprocessed using data augmentation techniques such as whitening, upsampling, random cropping, and random horizontal flipping before feeding them to the DNN. We note that accuracy measurements have been conducted using the TOP-1 error rate, which pertains to the error of a model's predictions when considering the class with the highest probability. Formally put, the TOP-1 error rate is calculated as follows:

$$\text{TOP-1 error} = \frac{\texttt{Number of Mispredictions}}{\texttt{Total number of predictions}} \times 100\% \tag{2.5}$$

② **Hardware Accelerator**: We use the NVIDIA Jetson AGX Xavier edge GPU as a hardware accelerator[2] equipped with an NVIDIA Carmel Arm-64bit CPU, a high-performance Volta GPU of 512 GPU cores and 64 Tensor cores. The mapping to GPU or Tensor cores depends on the data precision and is performed automatically by the DNN compiler. A native deployment of DNN models on the Jetson

AGX Xavier uses the GPU as a primary HW unit for inference computations. CPU and EMC are more related to data movement and memory operations. The NVIDIA Jetson AGX Xavier also allows the reconfigurability of hardware-related parameters, such as the number of active CPU cores and operating clock frequency of CPU, GPU, and EMC. If not mentioned otherwise, our experiments have been conducted under FP32 data precision using the Pytorch 1.9.0 framework.

2.6.2 Experimental Results

① **Co-optimization Results and Insights**: To showcase the efficiency of our co-optimization approach, we discuss the results regarding the search effectiveness from multiple perspectives as follows:

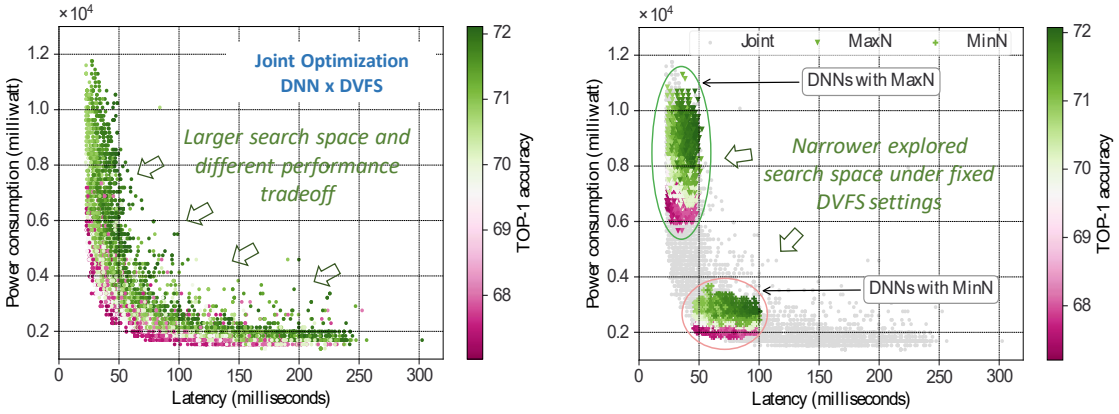


Figure 2.9 *Left*: Co-exploration results on the joint search space vs. *Right* Exploration results of the DNN search space under fixed `DVFS` policy.

② **Efficiency of the Co-exploration**: To showcase the co-optimization's importance, we compare the results obtained when considering the joint search space and performing a typical HW-aware NAS under fixed `DVFS` configurations. In our case, we chose two default `DVFS` settings proposed by the hardware manufacturer, NVIDIA: `MAXN` and `MINN`. The former sets all the clock frequencies at their maximum and enables all the CPU and GPU cores, whereas the latter sets the clock frequencies at the minimum default values and enables only 25% of the CPU cores. Generally, enabled CPU cores remain at an IDLE state when not used by any workload, which wastes the power budget. On the other hand, low utilization of GPU cores on their maximum clock frequencies incurs high power consumption as well, while similar computation speeds can be obtained if the clock frequency is adjusted correctly.
To better understand this impact, we provide the results of the co-exploration on the left of Figure 2.9. We qualitatively compare the former results against those of a typical HW-aware NAS under the `MAXN` and `MINN` `DVFS` policies, marked with different point shapes in the right of Figure 2.9. Each point in the figures corresponds to a combination of DNN and `DVFS` configurations (i.e., (dnn, hc)). As shown on the left of Figure 2.9, we remark that the region explored in the joint space is much larger than the regions explored when fixing the `DVFS` configuration to `MAXN` or `MINN`. Furthermore, the explored regions when fixing the two `DVFS` configurations are included in the co-exploration. Indeed, the joint search space

allows for exploring much larger solutions and, hence, different tradeoffs between accuracy and hardware efficiency (inference latency and energy consumption).

For instance, under the same latency level, an energy saving of up to 53% can be gained while enjoying similar accuracy levels. Similarly, when searching for low power consumption levels (i.e., in case the hardware device is running out of battery), a best-effort configuration can be selected from the obtained Pareto front in which the power consumption is minimal, with improved latency gains. This is due to the `MINN` configuration, which drastically worsens the execution latency. Similar observation can be attributed to the `MAXN`, which wastes the power budget on less resource-demanding workloads. Thus, a compromise between the two extremes is needed to achieve the desired performances by using the just-needed computation and hardware resources.

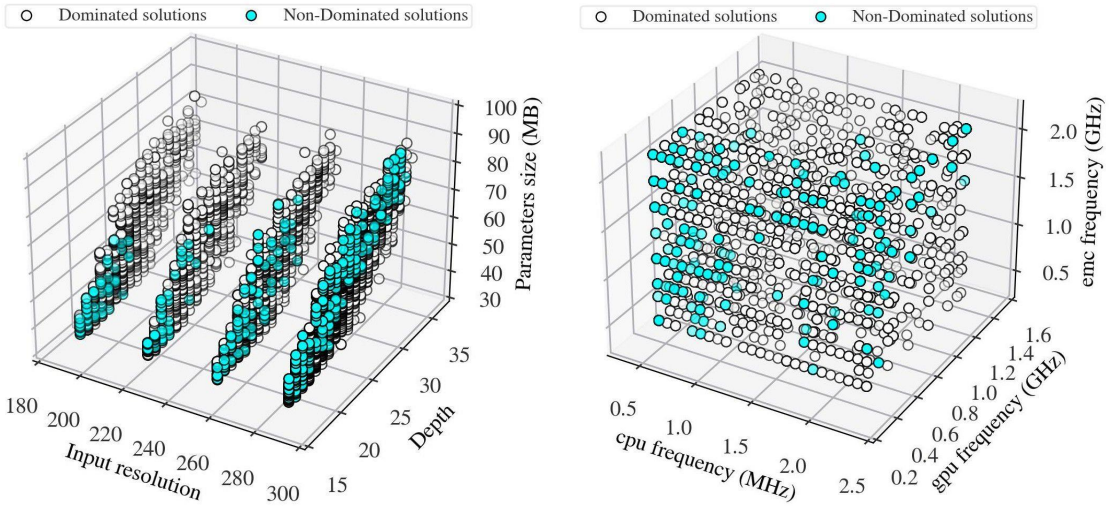


Figure 2.10 *Left*: Characterization of the explored and Pareto optimal DNN models. *Right*: Characterization of the explored and Pareto optimal `DVFS` configurations.

③ **Characterization of the Explored Search Space**: To show the diversity of the explored solutions, we summarize the characteristics of the explored DNN and `DVFS` configurations in Figure 2.10. The points correspond to all the explored solutions. Solutions belonging to the Pareto set are highlighted in blue. On the right of Figure 2.10, we show the explored `DVFS` configuration space. As shown, the Pareto optimal solutions are diverse, well distributed, and do not follow any specific trend. This confirms our earlier observation that no a priori knowledge can be used to choose the best-suited `DVFS` policy without on-device evaluation to get accurate latency and power consumption measurements. On the left of Figure 2.10, we report the main properties of the explored DNN architectures in terms of input resolution, depth (i.e., number of layers), and size of trainable parameters (in Megabytes). Similarly, architectures belonging to the Pareto set are well-distributed and diverse. We also remark that most models have high input resolution, enabling accurate inference. The diversity of the explored architectures supports the importance of the hardware-aware DNN architectural design co-exploration, as no a priori knowledge can be leveraged to get insights on the DNN performance without actual performance measurements.

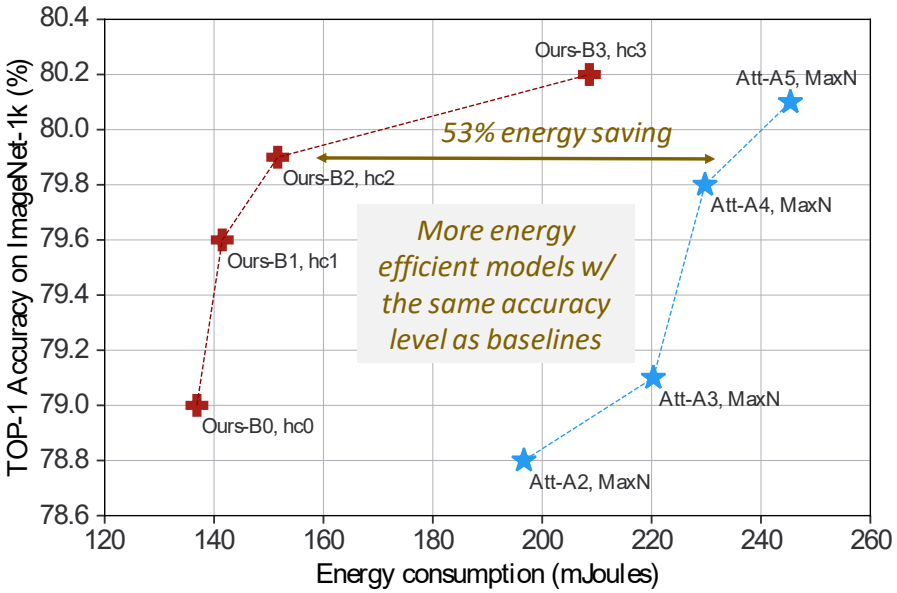


Figure 2.11 Accuracy and Energy of the baseline models proposed by AttentiveNAS [220] compared to non-dominated configurations selected from the Pareto front of our DVFS-NAS the ImageNet-1k dataset.

Table 2.4 Detailed performances of our DVFS-NAS Vs. the baseline from AttentiveNAS [220]

DNN, DVFS	TOP-1 Acc (%)	Latency (ms)	Avg. Power (mW)
Att-A2, MAXN	78.8%	29.91	6575
Att-A3, MAXN	79.1%	33.51	6575
Att-A4, MAXN	79.8%	32.67	7033
Att-A5, MAXN	80.1%	35.66	6881
Ours-B0, dvfs1	**79.0%**	**28.85**	**4744**
Ours-B1, dvfs2	**79.6%**	**30.82**	**4591**
Ours-B2, dvfs3	**79.9%**	**33.03**	**4591**
Ours-B3, dvfs4	**80.2%**	**34.10**	**6118**

④ **Analysis of the Pareto Optimal Configurations**

To further investigate the efficiency of the co-exploration, we select four pairs of (`DNN` × `DVFS`) from the Pareto front obtained from the co-optimization and compare them against state-of-the-art DNN models under the widely used default configuration proposed by NVIDIA, `MAXN`. We specifically compare against the top four (04) models from AttentiveNAS (A2-A5) [220]. We refer to these models as *Att-A(2-5)*. Figure 2.11 details the obtained results and the comparison. As shown, our obtained solutions (i.e., combinations of DNN models and hardware configuration) are referred to as *Ours-B(0-3), hc(0-3)*. Our co-optimization approach identified better solutions in terms of accuracy and hardware efficiency. From the table on the right of Figure 2.4, we notice a power gain of up to 53% under the same latency constraints. Furthermore, we remark an accuracy improvement of up to 0.5% on the ImageNet-1k dataset with better hardware efficiency and less energy demands.

⑤ **Fine-tuning on CIFAR-10 and CIFAR-100** To further investigate the performance of the obtained top models, we apply fine-tuning to CIFAR10 and CIFAR100 datasets. Since the obtained models are sampled from the pretrained SuperNet of AttentiveNAS [220], we do not need to train them from scratch. Instead, we apply a transfer learning from ImageNet to the new datasets. We apply the same fine-tuning strategy proposed in [215] by taking the ImageNet pretrained models checkpoints and fine-tuning on the new datasets. Thus, the fine-tuning is applied to the overall model's parameters (not only the classifier). For training, we apply the same settings used to train the SuperNet in [220].

Table 2.5 Fine-tuning results to CIFAR-10 and CIFAR-100

Models	ImageNet-1k	Cifar-100	Cifar-10
AttentiveNAS-A2[3]	78.8%	86.13%	97.45%
AttentiveNAS-A3[3]	79.1%	86.21%	97.61%
AttentiveNAS-A4[3]	79.8%	86.72%	97.71%
AttentiveNAS-A5[3]	80.1%	86.92%	97.83%
Ours-B0	**79.0%**	**86.12%**	**97.47%**
Ours-B1	**79.6%**	**86.69%**	**97.75%**
Ours-B2	**79.9%**	**86.95%**	**97.62%**
Ours-B3	**80.2%**	**87.03%**	**98.05%**

The models are trained using the SGD as an optimizer, with a weight decay of 1e-5 and momentum of 0.9. The learning rate is initialized at 0.1 and decays by 97% every three epochs. The training is done on eight distributed GPUs with a budget fixed to 100 epochs and a batch size of 128. Table 2.5 summarizes the obtained results. We compare the obtained results with the baseline models of AttentiveNAS.

⑥ **Neural Graph Post-Optimization with TensorRT** We further boost the hardware efficiency of the obtained models by post-optimizing the neural graph using TensorRT[4]. TensorRT is an SDK developed by NVIDIA to accelerate deep learning inference. Figure 2.12 depicts the workflow to create an optimized inference engine using the TensorRT Builder module. First, the network description and other optimization options, such as kernel/data quantization and computation mapping strategy, should be provided as input. Then, TensorRT creates an optimized engine by applying the specified optimization options and other internal optimization strategies, such as layer fusion. The optimized engine creates an execution context to perform the inference. Finally, an execution context is mapped into a single (or multiple) CUDA stream(s) for execution. We optimize the obtained the Pareto optimal models (i.e., Ours-B(0-3), hc(0-3)) by exploring the TensorRT compiler optimization parameters. Specifically, in addition to the implicit horizontal and vertical optimization strategies on the computation graph of the DNN, we enable the FP16 post-quantization, which typically incurs negligible accuracy drop compared to the INT8 quantization that requires a calibration step to recover

3. In the original paper of AttentiveNAS [220], the authors didn't report the results on CIFAR-10 and CIFAR-100. Therefore, we fine-tune the baseline models on the new datasets using our implementation.

the accuracy loss. The quantization is typically expected to minimize memory footprints and thereby reducing the overall latency and energy consumption. We report the results of this post-optimization in Figures 2.13.

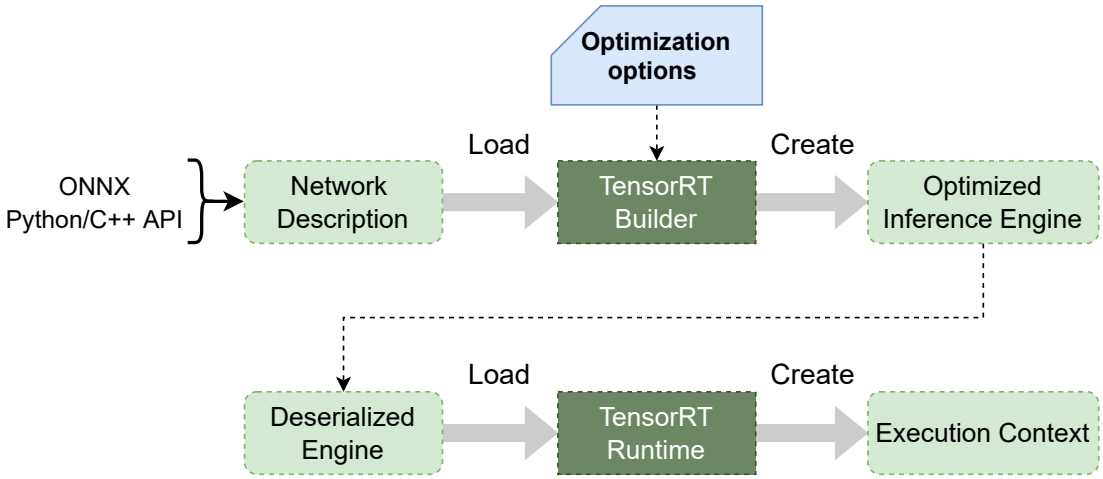


Figure 2.12 Workflow of generating an optimized TensorRT inference engine

The two sub-figures of Figure 2.13 depict the latency speedup and the relative power consumption during the TensorRT inference. We report the speedup in latency by calculating the relative latency improvement of the optimized models with TensorRT compared to the non-optimized models before applying TensorRT. Similarly, the power consumption is normalized by the values of the non-optimized models without the TensorRT optimization. Regarding power consumption, TensorRT incurs higher power consumption compared to the non-optimized implementation. We compare the performance when choosing `MAXN` as hardware configuration (referred to as `MaxN` in the figures) and when choosing the optimal hardware configuration found by our co-optimization approach (referred to as *Co-Opt* in the figures). On the *left* of Figure 2.13, we only apply the internal optimization of TensorRT. On the *right* of Figure 2.13, we apply a post-quantization to FP16.

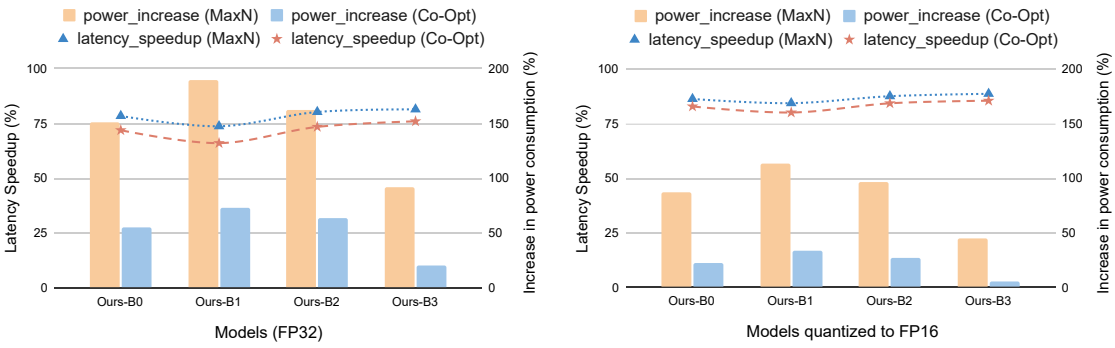


Figure 2.13 *Left*: Graph Neural Post-optimization results using TensorRT without post-quantization. *Right*: TensorRT optimizaion via FP16 post-quantization.

First, when executing our models with the `MAXN` configuration, TensorRT boosts the hardware efficiency of the models with a latency speedup of up to ∼81% and ∼88% for non-quantized and quantized models, respectively. However, this comes

at the expense of increased power consumption up to ∼189% and ∼112% for non-quantized and quantized models, respectively. This is explained by the ability of TensorRT to take advantage of the GPU computation units during the inference. The optimized engine is highly parallel, which maximizes the GPU utilization ratio and thus incurs an increase in power consumption. Second, by optimizing the hardware configuration (Co-Opt)), we reduced the increase in power consumption from ∼189% and ∼112% to ∼73% and ∼33% for non-quantized and quantized models, respectively. Regarding latency, we notice a small average gap of ∼7% in the latency speedups of the `MAXN` and the optimal hardware configurations found by our algorithm. Nevertheless, since the two hardware configurations (i.e., `MaxN` and optimal configurations (hc-(0-3)) speed up the computations latency by a ratio of ∼85% to ∼88%, this small gap becomes negligible when considering the power saving from up to ∼61% achieved by the optimal hardware configuration found by our co-optimization approach. Therefore, co-optimization remains important to balance latency speedup and power saving, even when using high-performance SDK such as TensorRT.

2.7 Discussion and Key Insights

Our results demonstrated that the co-optimization of `DNN` and `DVFS` configurations on edge GPUs is extremely important to balance DNN performance and hardware efficiency. On the one hand, edge GPUs are heterogeneous embedded systems and can execute multiple applications concurrently. In this case, the availability of hardware resources (e.g., computation units and operating frequencies) varies due to other concurrent applications' execution. Thus, the performance of an application changes under different resource availability scenarios. Therefore, knowing beforehand which DNN model is adapted to each scenario can prevent the violation of the application's performance requirements and meet the power and resources budget constraints. On the other hand, the optimal hardware configurations for a single application can be used at runtime to dynamically scale the hardware platform for the performance requirements and the power budget constraints. Therefore, our proposed co-optimization framework can be integrated into a real-time scheduling algorithm to dynamically adjust DNN and `DVFS` configurations according to resource availability. It can also be used as a dynamic frequency scaling governor at runtime instead of the currently used heuristics that have been proven weak in capturing the correlation between the hardware configuration, the application workload, and the availability of hardware resources.
More interestingly, the joint optimization is becoming an important part of typical HW-aware NAS frameworks to co-design Neural Networks and Hardware accelerators for *Edge AI* applications and systems. Emerging DNN models are more diverse, hybrid, and compound atypical computations and dataflow. This diversity opens room for further reconsidering the DNN deployment on edge devices. That being said, more attention should be delegated to answering questions on:

— **(*i*)** *What are the execution requirements and bottlenecks of DNN workloads on existing commodity edge devices ?*

— **(*ii*)** *How can we adjust the hardware accelerator to satisfy those requirements without changing the DNN model or opting for another the HW device ?*

Thus, an a priori and systematic DNN execution characterization and exhaustive exploitation of the HW dimension can help in ensuring the DNN inference requirements on edge devices without drastically compromising accuracy.

2.8 Summary

This chapter investigates the importance of the joint exploration of `DNN` and `DVFS` configurations for edge GPU accelerators to balance performance and energy efficiency. First, we conducted a preliminary analysis of the impact of clock frequency variations on the inference latency and power consumption. From our observation of the non-linear behavior of the correlation between frequency scaling on latency and power, we proposed to integrate the `DVFS` as an additional HW-related parameter that needs optimization alongside the DNN during the HW-aware NAS.

In this context, we propose a co-optimization approach based on an evolutionary algorithm (NSGA-II) to explore these two search spaces. The aim was to minimize three objective functions: DNN TOP-1 error, latency, and power consumption. Experimental results on the Jetson AGX Xavier NVIDIA GPU demonstrated the efficiency of the co-optimization compared to typical HW-aware NAS under fixed hardware configurations. Moreover, the top pairs found by our co-optimization are more energy-efficient with up to 53% gains than solutions found by state-of-the-art models under the same accuracy and latency constraints. We have also demonstrated the importance of co-optimization when using a high-performance SDK such as TensorRT. The results depict a latency speedup of up to 81% and power saving of 61% compared to the MaxN configuration. Future works can extend our co-optimization framework by proposing more efficient selection and recombination operators for the optimization algorithm. It is also worth investigating more hardware architectures, configurations, and DNN benchmarks to further emphasize the importance of co-optimization. More importantly, combining a dynamic execution strategy that would benefit from the dynamic clock frequency scaling can further enhance performance and energy efficiency. These dynamic execution schemes can be drawn from existing works such as dynamic routing in Dynamic-OFA [140], or dynamic neural scaling methods like early-exit and computation skipping as detailed in [82].

Chapitre 3

HADAS: Hardware-Aware Dynamic Neural Architecture Search for Edge Performance Scaling

3.1 Introduction

Neural Networks (NNs) have become integral machine learning techniques that enable intelligence for today's edge computing applications. Edge computing platforms are commonly deployed in real-world environments, which exposes them to significant variations in runtime performance. These variations are mostly attributed to the uneven distribution of received data, which poses challenges in effectively processing inputs within constrained hardware resources. Examples of such constraints include limited power budget and system charge. The relevance of Dynamic Neural Networks (DyNNs) [82] has increased significantly. Unlike traditional static models that have fixed computational graphs, DyNNs have the ability to adapt their model structure and parameters depending on the runtime context (i.e., input complexity and system state). This adaptability allows DyNNs to achieve hardware resource efficiency at the edge while preserving the accuracy of the models.

One prominent DyNN technique is *early exiting*, where dynamic depth variation is applied sample-wise to avoid redundant computations. Early exiting is a technique that enables the efficient processing of "easier" input samples in earlier stages (e.g., layers, channels) of an NN model, hence facilitating the conclusion of their processing within a short time while incurring less energy requirement. As shown in figure 3.1, this DyNN technique is often realized through a multi-exit architecture that integrates intermediate classifiers onto a shared backbone model [216, 169, 19, 188, 166].

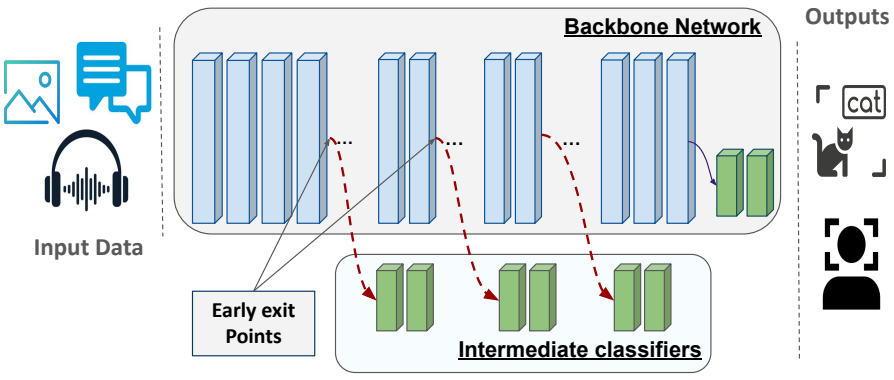


Figure 3.1 Dynamic Inference with early-exit

Typically, the design workflow of multi-exit models initially assumes that the

backbone's 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 [216], or models rendered through the design automation frameworks of Neural Architecture Search (NAS) [220]. 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 neural components, such as early-exit points and intermediate classifiers, are added to serve as the backbone of a dynamic model.**

In this chapter, we investigate another dimension to unleash the potential of inference dynamicity from a neural network perspective. We extend our DVFS-NAS detailed in Chapter 1 [26, 27, 25] and reformulate the problem as a joint optimization of neural network and hardware dynamic features.

1. The former encompasses finding the optimal design of backbones and early-exit positions and branches.
2. The latter involves identifying optimal DVFS configurations for the realized dynamic neural network.

We propose HADAS, a comprehensive HW-aware NAS framework for dynamic neural networks to balance inference performance-efficiency tradeoffs. We make the design of the DyNN fully searchable by **constructing a novel joint search space** for (i) static backbone neural network design and (ii) dynamic early-exit branches and DVFS configurations. Then, **we elaborate an innovative co-optimization framework by hierarchizing the design exploration process** of static and dynamic components using two dedicated optimization engines that inter-communicate using their Pareto sets improvements. We validate our approach using the CIFAR-100 dataset on various hardware configurations: ARM, Denver CPUs, Volta, and Pascal GPUs from the NVIDIA Jetson AGX Xavier and TX2. Evaluation results have seen the superiority of HADAS by providing up to $\sim$57% while sustaining an acceptable level of accuracy. Contributions and results of this chapter have been published in:

— [21] **Halima Bouzidi**, Mohanad Odema, Hamza Ouarnoughi, Mohammad Al Faruque, Smail Niar. "HADAS: Hardware-Aware Dynamic Neural Architecture Search for Edge Performance Scaling" in Proceedings of the Design, Automation and Test in Europe Conference and Exhibition(DATE), April 2023. **Nominated for the Best Paper Award.**

3.2 Related works

3.2.1 Dynamic Early Exit and NAS

Early-exiting has been widely adopted to realize DyNNs on edge devices given their "simple-yet-effective" characteristic. The direct approach to realize Multi-exit networks has been to *branch* intermediate classifiers from the earlier layers of a backbone model and retraining the model to maximize the performance of all classifiers [169, 216, 174, 93]. With an effective input-to-exit mapping policy, multi-exit models achieve computational efficiency as *simpler* input samples can be classified at the earlier classifiers (exits) while maintaining the model's representational power through retaining the full classifier for the *harder* samples. In the

aforementioned works, the multi-exit networks have been manually designed based on heuristic choices of positions, structure, and count conditioned on their respective backbone architecture [121]. This manual design approach highly depends on the choices of backbones and exit branches, overlooking further performance gains from automating the design process to broadly explore the possible variants of dynamic neural network designs.

To leverage design automation for DyNN, recent works [166, 259] 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, [166] instituted a small search space of one exit branch at a fixed position, which is not scalable and may fail to capture the variation of complexity in input data. S2DNAS [259] introduces a large search space for multi-exit networks on depth and width levels for a fixed backbone network. However, despite its effectiveness, S2DNAS is specific to convolutional NNs only and cannot be generalized to new emerging types of NN such as Transformers or Graph Neural Networks. Our approach, on the other hand, expands existing works by making the multi-exit DyNN fully and automatically searchable by enlarging the search space for exit branch positions and jointly optimizing the backbone networks, mitigating the drawbacks of [166] and [259].

3.2.2 Dynamic Hardware Reconfiguration

As demonstrated in Chapter 1, dynamically scaling NNs results in different computational and energy footprints that require adapting the hardware configuration accordingly [25]. To meet the demands of early-exit DyNN, authors in [170, 62] have co-designed the hardware 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 [212, 127]. For instance, Predictive-exit [127] co-optimize the placement of early-exit branches and `DVFS` configurations by lowering the clock frequency after the input data is correctly classified at earlier stages. Nonetheless, when the complexity variance of the received input data is high, this may result in a considerable amount of clock frequency switching by the `DVFS` governor, resulting in a high latency overhead.

Thus, in our work, we mitigate this challenge by searching for a global `DVFS` policy for the entire DyNN that better captures the variations in input data complexity and design composition of early-exit branches.

Table 3.1 illustrates the difference between HADAS and existing multi-exit network design approaches and how it improves upon them through its joint optimization approach while being compatible with existing state-of-the-art NAS frameworks. We note that *Compatibility* refers to the generalization and ease of integration of our design approach onto existing neural network architectures and NAS infrastructure.

Table 3.1 Comparison between Related-works and ours

Work	Early-Exiting	NAS	DVFS	Compatibility
BranchyNet [216]	x			
CDLN [169]	x			
S2dnas [259]	x	x		
Dynamic-OFA [140]		x		x
EExNAS [166]	x	x		
Edgebert [212]	x		x	
Predictive Exit [127]	x		x	
HADAS	**x**	**x**	**x**	**x**

3.3 Motivational Example

Given the intrinsic variance of real-world data, the dynamic behavior of NN is desirable to use NN components and hardware resources efficiently. That being said, easy input data typically requires simple NN and hardware resources. On the other hand, difficult input data require complex reasoning with enormous NN operations, which naturally implies more hardware resource demands. Dynamically activating neural network components according to input data complexity gives rise to *Dynamic Neural Networks*. From a DyNN design perspective:

1. NN backbones need the tuning of both the *Macro* and *Micro* architectures. The former is defined by the high-level structure choice that delineates neural block arrangement and components. The latter is defined by the inner structure of the neural block regarding the depth and width of layers, stride, kernel size, and activation functions.
2. Early exit placement and components must be chosen carefully within the backbone NN. In the context of a depthwise early exit, intermediate classifiers are another design dimension involving operations that need further parameter tuning. In addition, the position of the early exit itself is a design choice that compromises accuracy and efficiency.
3. Considering the hardware landscape in the loop, further considerations of the hardware architecture and parameters (e.g., clock frequency) add another layer of Complexity. In this context, tuning the `DVFS` configuration regarding the DyNN model configuration can enhance energy efficiency while sustaining an acceptable processing latency. The reason being is that compact DyNNs typically require less processing speed as they were originally designed for efficiency. Thus, lowering the clock frequency through `DVFS` is more beneficial for less energy footprint. The same assumption holds when considering a large DyNN, except that operating clock frequency needs to be carefully adjusted so that latency will not decrease drastically in favor of less power consumption.

In Figure 3.2, we showcase how the interplay between the three aforementioned design dimensions impacts the overall performance of DyNNs. We take as baselines the respective *most compact* and *highest performing* image recognition models, **a0** and **a6**. These models are provided through a state-of-the-art NAS framework, AttentiveNAS [220]. However, our method is *compatible* with any other NAS framework could be used instead of AttentiveNAS. We compare their performance

against another model that resulted from co-optimizing backbone and early-exit neural components, as well as the inclusion of DVFS for energy efficiency. In this case, we implement our co-optimization approach on top of AttentiveNAS [220] to ensure a fair comparison by having the models share the same macro-architecture. We also note that AttentiveNAS [220] is only dedicated to backbone design optimization and not initially designed for dynamic inference through early exit. Thus, we include the early exit components and hardware HW-aware NAS algorithm as our new co-optimization framework. In the following discussion, classification accuracy and energy consumption are leveraged as the performance comparison metrics. We use the CIFAR-100 image dataset for models' training and accuracy evaluations and the NVIDIA Jetson TX2 platform for hardware performance benchmarking using the Pytorch framework [1].

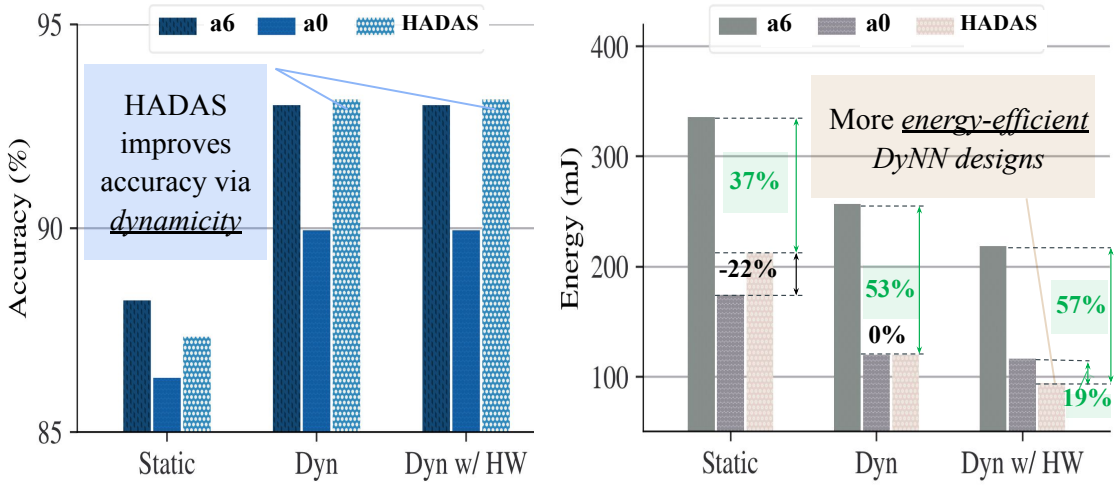


Figure 3.2 Comparing the performance of (a0, a6) from AttentiveNAS and HADAS's model on CIFAR-100 and the Jetson TX2 Pascal GPU hardware.

As shown in Figure 3.2, we designate three stages of DyNN optimizations that can be applied to maximize performance efficiency:

❶ ***Static*** in which we optimize the backbone model design through a typical HW-aware NAS process ;

❷ ***Dyn*** – in which we add dynamicity in the backbone inference process through integrating *early-exiting* classifiers along the depth of the NN ;

❸ ***Dyn w/ HW*** – in which we integrate both early-exit and DVFS through tuning the operating clock of the TX2 hardware units including CPU, GPU, and EMC.

The *left* subfigure compares the accuracy, whereas the *right* subfigure compares the energy efficiency. We note that the same trend has been observed for latency as energy is the product of latency and power consumption.

Firstly, with regards to accuracy, the model obtained from the co-optimization (noted as *HADAS*) outperforms the static **a0** and is on-par with **a6** after applying the *static* and *Dyn* optimizations. The accuracy values are kept the same after integrating the DVFS features as it has no impact on the data representation of the inference results.

1. Pytorch Framework: https ://pytorch.org/

Secondly, though, the energy efficiency of *HADAS*'s model is enhanced considerably with every applied optimization compared to the other models (*right barplot)*. That is, after the first stage of ***Static*** optimization by integrating hardware-related objectives into the optimization process, **a0** is reasonably deemed the most energy-efficient model given its compactness (22% more energy-efficient than ours). Nevertheless, in the *Static* case, *HADAS* depicts the best accuracy and efficiency tradeoff with up to ∼37% energy efficiency compared to **a6** while being +1.01% more accurate than **a0**.

On the other hand, when ***Dyn*** co-optimizations are applied, the *HADAS* model's efficiency improves drastically to reach the *same* level of energy efficiency as **a0**. Even more so, energy efficiency reaches up to ∼19% gains compared to **a0** once ***Dyn w/ HW*** both early-exit and `DVFS` optimizations are in place. In light of the observations, it's mandatory to consider every design dimension into the HW-aware NAS to push the performance boundaries of dynamic neural networks.

Analysis Summary and Conclusions: Through its awareness of the *dynamic* and `DVFS` parameter spaces, the co-optimization approach can balance the accuracy efficiency trade-offs more than the conventional NN design approaches such as AttentiveNAS [220]. Specifically, our proposed co-optimization approach of the backbone neural network, early exiting features, and the hardware settings (i.e., `DVFS`) leads to DyNN model designs that are highly prone to benefit from the static, dynamic, and hardware deployment aspects altogether. These dynamic models can be leveraged to dynamically scale NN models through early-exit and HW devices through `DVFS` at runtime for an effective inference process on resource-constrained systems.

3.4 Novel Scientific Contributions

In this work, we present the following scientific contributions and novelties:

1. We present HADAS, a novel hardware-aware NAS framework that jointly optimizes the design of multi-exit DyNNs and DVFS settings.
2. HADAS is built to leverage the existing infrastructure of pretrained supernets provided through state-of-the-art NAS frameworks, and is also compatible with existing runtime controllers for an effective end-to-end design workflow.
3. We formulate the design space exploration problem for multi-exit architectures as a bi-level optimization problem solved through two nested evolutionary genetic engines. The outer engine identifies optimal backbone designs. Whereas the inner engine co-optimizes the exits' integration and the DVFS settings.
4. On the CIFAR-100 dataset and a diverse set of hardware devices/settings, our experiments demonstrated that HADAS models can realize energy efficiency gains by up to **∼57%** over models designed through conventional methods while preserving the desired level of accuracy.

3.5 Problem Statement

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:

1. (i) ***The backbones*** **($\mathcal{B}$)**; which are models originally designed in a monolithic fashion for *static* inference with no adaptive behavior,
2. (ii) ***The exits*** **($\mathcal{X}$)**; which are the dynamic components to be integrated onto a backbone,
3. (iii) ***The DVFS settings*** **($\mathcal{F}$)**; constituting the space of operational clock frequencies for the underlying hardware components.

For the DyNNs, our reasons for designating $\mathcal{B}$ and $\mathcal{X}$ as separate subspaces are threefold:

1. To maintain the generality of the approach by having the $\mathcal{X}$ subspace indifferent to the "*type*" of candidate backbones in $\mathcal{B}$. Thus, our approach can be readily adopted when a new type of backbones is to be considered (e.g., Transformers).
2. To leverage the existing infrastructure of pretrained supernets from established NAS frameworks (as in [30, 220, 219]) so as to provide high-caliber backbone models for the $\mathcal{B}$ subspace.
3. To ensure the generality of the framework for other *dynamic* inference strategies. For instance, the $\mathcal{X}$ search space can be extended to support width-wise early-exit features through channel-based early exit [93, 259].

To detail the algorithm used for ranking of the architectural designs generated dynamically, we denote $\mathcal{S}$ and $\mathcal{D}$ as generic optimization 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 early exit component. We note that each performance metric is computed in an input-wise manner under the assumption of an ideal mapping of input data to intermediate classifiers. Hence, this implies a bi-level optimization problem with the $\mathcal{B}$ as the outer-level subspace for *Static* components and ($\mathcal{X}$, $\mathcal{F}$) as the inner-level ones for *Dynamic* components. More formally, the overall bi-level optimization problem for DyNN design is formulated as follows:

$$b^* = \underset{b \in \mathcal{B}}{\arg\max}\ \psi[\mathcal{S}(b), \mathcal{D}(x^*, f^* \mid b)] \tag{3.1}$$

$$s.t.\ x^*, f^* = \underset{x \in \mathcal{X}, f \in \mathcal{F}}{\arg\max}\ \mathcal{D}(x, f \mid b) \tag{3.2}$$

Where the global optimization objective to identify the ideal parameter combination (b^*, x^*, f^*) that maximizes an optimality score computed by ψ, given the static $\mathcal{S}$ and dynamic performances $\mathcal{D}$. To solve our bi-level optimization problem, we need to solve two sub-problems: The first sub-problem is given by equation (3.1), where we search for optimal backbones that maximize both of:

— Static performance ($\mathcal{S}(b)$) when no early-exit or DVFS are employed.

— Dynamic performance ($\mathcal{D}(x^*, f^* \mid b)$) when optimal early-exit (x^*) and DVFS (f^*) policies are employed.

For each sampled backbone b, we need to solve a second sub-problem given by equation (3.2), where we search for optimal early-exit and DVFS policies that maximizes the dynamic performance ($\mathcal{D}(x, f \mid b)$). The output of this second sub-optimization problem are optimal early-exit (x^*) and DVFS (f^*) policies that are used to assess the performance in the first sub-optimization problem. Thus, the two sub-optimization problems are highly dependedent to each other. In practice, the underlying optimization objectives are conflicting by nature – e.g., the larger computationally expensive models enjoy higher accuracy scores and vice versa. Thus, the problem can be approached as a multi-objective optimization where we seek a Pareto optimal set of solutions. For instance, in equation (3.2), a solution (x^*, f^*) is said to be Pareto optimal if for all the objective functions $d \in \mathcal{D}$:

$$\begin{aligned} d_k(x^*, f^*) \geq & d_k(x, f) \forall k, (x, f) \\ & \text{and } \exists j : d_j(x^*, f^*) > d_j(x, f) \forall (x, f) \neq (x^*, f^*) \end{aligned}$$

In this scheme, backbones are first assigned to non-dominated sets of solutions (Pareto fronts) and given a Pareto rank depending on their degree of dominance. Backbones with the same Pareto rank are further sorted according to their diversity using a crowding distance [51]. That is, solutions with the highest crowding distance receive the highest scores. The crowding distance metric computes the disparity of candidate backbones in the objective space as shown in (3.3).

$$crow\_distance = \sum_{k=1}^{K} \frac{y_{(i+1),k} - y_{(i-1),k}}{f_{max,k} - f_{min,k}} \tag{3.3}$$

Where K is the number of objectives, $y_{(i+1),k}$ and $y_{(i-1),k}$ are the values of the k-th objective for the objective vector immediately succeeding and preceding the i-th objective vector, respectively, in the sorted population by the Pareto ranking. The fitness evaluation function ψ in equation (3.4) then combines Pareto ranking and crowding distance to assign final optimilaity scores where optimal DyNNs yield the highest degrees of dominance and are more diverse.

$$\psi(\mathcal{S}, \mathcal{D}) = Pareto\_rank(\mathcal{S}, \mathcal{D}) \times crow\_distance(\mathcal{S}, \mathcal{D}) \tag{3.4}$$

3.6 Proposed Approach

Figure 3.3 illustrates the overall framework of HADAS. Principally, we adopt nested genetic algorithms [63] to solve the bi-level optimization problem for the co-design of dynamic neural network (DyNN) on edge platforms.

3.6.1 Outer Optimization Engine (OOE)

The OOE considers two primary tasks:

1. Searching through the design space $\mathcal{B}$ to identify the best backbone candidates in accuracy, latency, and energy consumption.

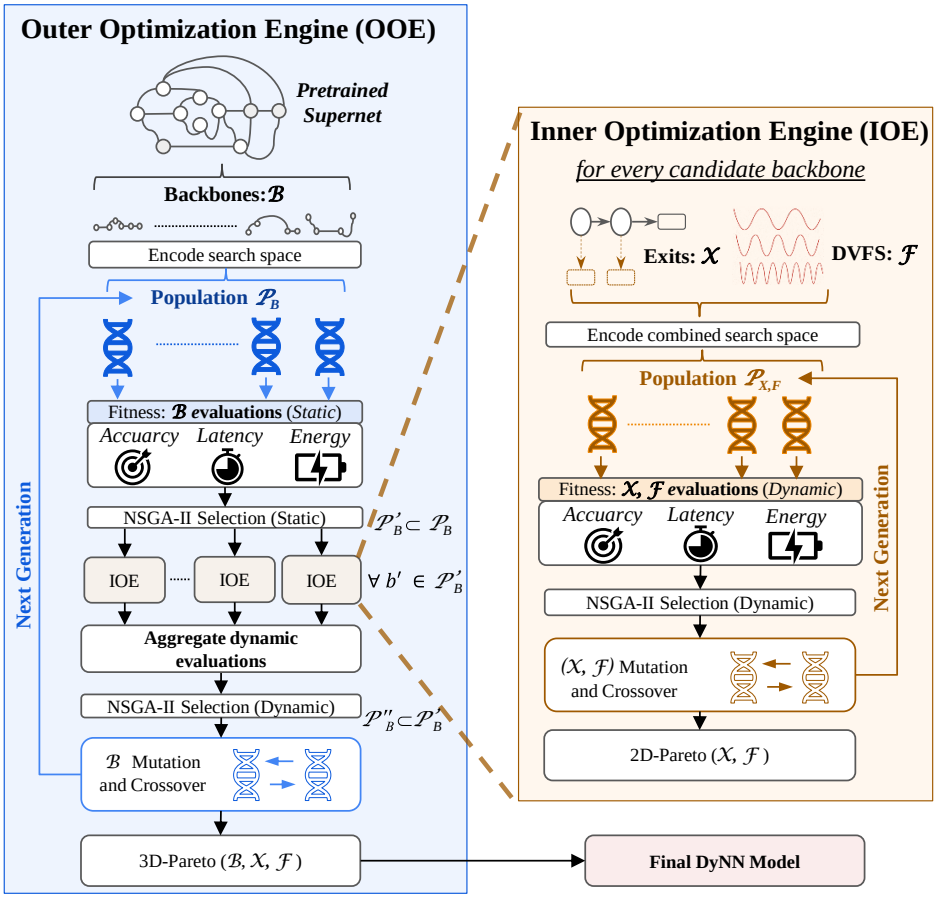


Figure 3.3 HADAS co-optimization framework.

2. Selecting the most promising backbones for early-exit features integration and DyNN optimization.
3. Ranking the resulting DyNNs according to their aggregate $\mathcal{S}$ and $\mathcal{D}$ evaluations. The following details the OOE engine regarding the typical NAS components: search space, strategy, and fitness evaluation.

① Backbone Design Subspace $\mathcal{B}$

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 [30, 220]. Furthermore, the supernet paradigm is highly flexible towards macro- and micro-architecture settings and training approach [48]. Primarily, most existing supernets have fixed macro-architecture as a succession of neural blocks where each is a predefined set of operations (e.g., mobile inverted residual bottleneck block [186] or transformer [57]).
From here, our framework, HADAS, is built to leverage the pretrained supernets of existing NAS frameworks to construct the $\mathcal{B}$ subspace of backbones. Each viable subnet (backbone) can be denoted as $b \in \mathcal{B}$. Figure 3.4 details the employed

encoding scheme. The macro-architecture is fixed to MBConv blocks arranged sequentially. A search space is defined by the micro-architecture parameters set at each block. The parametrical configurations of each block are generally *encoded* into a vector of discrete variables usable by the search algorithm. The encoding vector is referred to as *genome* where each variable is a *gene* corresponding to one of the micro-architectural parameters. A discrete-value encoding is suited to evolutionary algorithm to apply variation operations such as *mutation* and *crossover*.

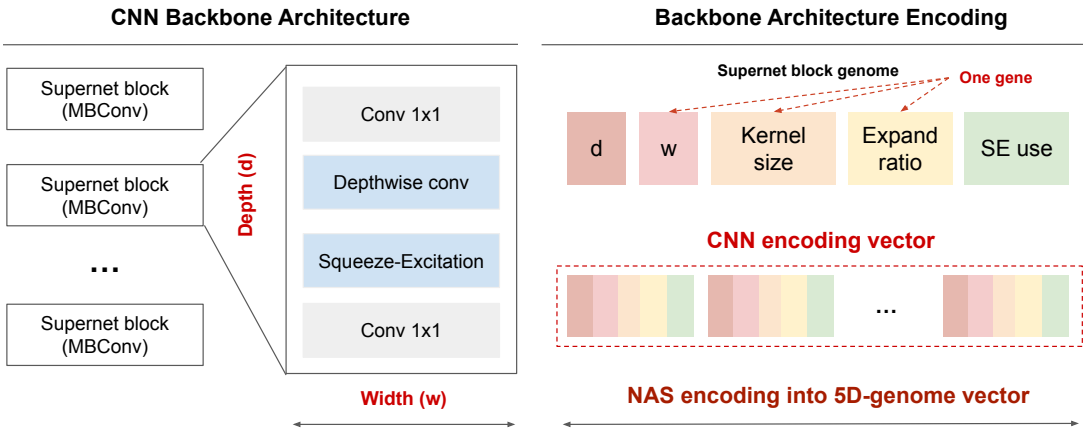


Figure 3.4 Backbone Neural Network Encoding

② Backbone 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. Evolutionary strategy is widely used for NP-hard optimization problems (e.g., NAS). The evolutionary search is typically set to run within a fixed optimization budget. In our case, we run the evolutionary search for G generations. At every generation, g, a population of backbones, $\mathcal{P}_{\mathcal{B}}^{g}$, from which the encoded pretrained subnets can be sampled. Afterwards, $\forall b \in P_{\mathcal{B}}^{g}$, a fitness evaluation under *static* conditions is performed as:

$$\mathcal{S}(b) = Fit(Acc_b, L_b, E_b) \tag{3.5}$$

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) consumption, respectively. Estimates for L_b and E_b are obtained based on hardware measurements, i.e a HW-in-the-loop approaches.

Other solutions are also possible such as : lookup tables or prediction models. We employ an HW-in-the-loop strategy because the search complexity is already dominated by the training time of the intermediate classifiers, incuring an inevitable bottleneck. Thus, accelerating the HW evaluation by proxy estimations using lookup tables or prediction models will not be beneficial and will only compromise the accuracy of measurements.

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 $b \in P_{\mathcal{B}}^{g}$ is ranked using the NSGA-II non-dominated sorting algorithm [51]. If several backbones share the same rank, their diversity scores are used for re-ranking

via measurements of the crowding distance [52]. This early selection procedure enables pruning the population to reach a smaller subset $\mathcal{P}_{\mathcal{B}}^{g'} \subset \mathcal{P}_{\mathcal{B}}^{g}$, where every $b' \in P_{\mathcal{B}}^{g'}$ is mapped to an IOE – inner optimization engine (detailed later) to obtain the overall dynamic neural network 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 $b' \in P_{\mathcal{B}}^{g'}$. 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 $\mathcal{P}_{\mathcal{B}}^{g''} \subset \mathcal{P}_{\mathcal{B}}^{g'}$. To rank the obtained Pareto fronts from the IOE, we employ the hypervolume measurements.

In a nutshell, the hypervolume [191] is a metric used to assess an optimizer's quality by taking into account the optimality, diversity, and spread of the explored solutions. More formally, given m objective functions, the hypervolume is computed as the Lebesgue measure $\Lambda(\cdot)$ of the space that's jointly dominated by the objective vectors in the Pareto Front approximation set obtained by the optimizer, $\mathcal{PF} \in \mathbb{R}^m$, and bound by a reference point $X_{ref} \in \mathbb{R}^m$ that represents the worst values of the objectives (in our case: accuracy, latency, and energy). The mathematical formulation is given as follows:

$$H(X_{ref}, \mathcal{PF}) = \Lambda\left(\bigcup_{x \in \mathcal{PF}} [Acc(x), Acc(X_{ref})] \times [L(x), L(X_{ref})] \times [E(x), E(X_{ref})]\right) \tag{3.6}$$

Lastly, after the second selection, $\mathcal{P}_{\mathcal{B}}^{g''}$ undergoes *mutation* and *crossover* operations to construct a new population $\mathcal{P}_{\mathcal{B}}^{g+1}$ for generation g+1. A uniform mutation is employed on the neural block level of backbones by sampling new resolution, depth, width, kernel size, or expand ratio under a probability threshold of 0.3. The crossover is applied by randomly picking two individuals from the Pareto set and swapping their neural block specifications under a probability threshold of 0.8. This outer loop cycle repeats until generation G at which the Pareto optimal set of backbones, early exit components, and `DVFS` (b^*, x^*, f^*) is returned as the final solution.

3.6.2 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:

① Early Exit Design $\mathcal{X}$ 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 [220]) constitute M sequential computing neural MBConv blocks (i.e., an aggregation of interrelated layers) between which effective placement of the exits can be realized. We illustrate this in Figure 3.5 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 exit positions for

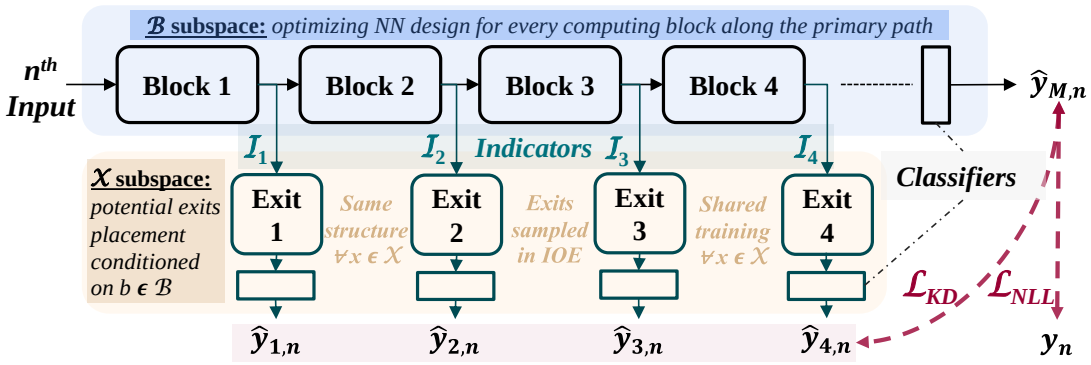


Figure 3.5 The combined $\mathcal{B}$ and $\mathcal{X}$ search spaces

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.

— (iii) Minimizing the training costs of the exits

For our experiments, the exit structure constituted a single sequential computing block of convolutional, batch normalization, and activation layers, which are followed by a final classifier layer. The input features of the exit block are variable and depend on the backbone design and early exit placement. In a typical CNN, early neural blocks extract fewer features than the last ones. Remarkably, the many output features extracted in the last layers result in accurate early classifications while incurring high latency and energy consumption. Thus, the objective of the joint optimization is to perform an effective search to identify the best tradeoff given the design constraints of DyNNs.

② Early-exit Branches 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 one objective [216]. 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 gains. We assume that the neural blocks within b' are already well trained, and we adapt the intermediate classifiers to leverage -to the most extent- the extracted features at different levels of b'.

To effectively and fairly train the exit blocks (i.e., intermediate classifiers), we adopt a knowledge distillation approach [76]. Naturally, exit blocks placed at the last neural blocks of b' would benefit from the maturity of the extracted features and thus correctly classify images of different degrees of complexity. Notably, these last classifiers can also serve as *teachers* to earlier exit blocks *student* for a better

training convergence [174, 247]. For the training loss function itself, we adopt a hybrid loss function ($\mathcal{L}_{total}$) combining **(i)** the Negative log-likelihood [248] as a classification loss (noted as $\mathcal{L}_{NLL}$) to learn from the ground-truth data and **(i)** the Kullback–Leibler divergence [172] as a knowledge distillation loss (noted as $\mathcal{L}_{KD}$) to learn from teacher exit blocks. The loss function to train simultaneously every $x \in \mathcal{X}$ is given as follows:

$$\mathcal{L} = \frac{1}{N}\sum_{n=1}^{N}[\frac{1}{M\text{-}1}\sum_{m=1}^{M-1}(\mathcal{L}_{NLL}(y_n, \hat{y}_{m,n}) + \mathcal{L}_{KD}(\hat{y}_{m,n}, \hat{y}_{M,n})] \tag{3.7}$$

Where N is the total number of training samples and $M-1$ is the total sampled number of exit blocks. 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, $\hat{y}_{M,n}$. We illustrate how these loss components are defined in Figure 3.5, and refer readers to [174] for more details.

③ DVFS $\mathcal{F}$ 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 $\mathcal{F}$. To elaborate more, depending on the backbone design "b" and exit block placement "x", compute-bound DyNNs workloads may benefit from high clock frequency on the main compute unit (CPU/GPU) while lowering the clock frequency of the EMC. The same logic can be applied to memory-bound DyNNs workloads by lowering the clock frequency on the compute units. Thus, careful tuning of `DVFS` settings can further enhance the energy efficiency of the overall DyNN inference while sustaining an acceptable latency.

④ Early Exit and DVFS Evolutionary Search

Similar to the OOE, an IOE also performs an evolutionary NSGA-II algorithm to navigate the combined search spaces of $\mathcal{X}$ and $\mathcal{F}$. We note that the IOE is executed for a fixed backbone b selected by the OOE. The reason for choosing NSGA-II at this stage is the complexity of the combined design space of $\mathcal{X}$ and $\mathcal{F}$. More specifically, the complexity of $\mathcal{F}$ depends on the targeted hardware architecture, which is fixed a priori for the OOE and IOE. However, the complexity of $\mathcal{X}$ varies according to the depth of b, which depends on the selected backbone by the OOE.

As the number of possible exits is a decision variable, the vector encoding the exit positions is dynamic. The combination of `DVFS` comprises the operating clock frequencies of CPU, GPU, and EMC. We also adopt a value-encoding scheme to represent the combinations of $\mathcal{X}$ and $\mathcal{F}$ as one encoding vector to be used by

the search algorithm. Following the logic of the evolutionary algorithm, 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:

$$\mathcal{D}(x, f \mid b') = \frac{1}{\sum_{i=1}^{M\text{-}1} \mathcal{I}_i} \sum_{i=1}^{M\text{-}1} \mathcal{I}_i \cdot [score_i] \tag{3.8}$$

$$s.t. \;\; score_i = \mathcal{N}_i * \frac{E_{x_i,f}}{E_b} * \frac{L_{x_i,f}}{L_b} * (dissim_i)^{\gamma} \tag{3.9}$$

where equation (3.8) 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 (3.9), 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:

$$dissim_i = 1 - \max\left(\mathcal{N}_{0:i-1}\right) \tag{3.10}$$

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 to minimize the overhead of the runtime controller when choosing the optimal exit point for a given input image.

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 [52] 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. A uniform mutation is employed by sampling a new number of exits, exit positions, or DVFS settings under a probability threshold of 0.3. The crossover is applied by randomly picking two encoding vectors from the Pareto set and swapping their exit positions or clock frequencies under a probability threshold of 0.8. 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.

3.6.3 Runtime Controller

When a DyNN design is chosen for the final deployment, a runtime controller must be implemented to provide the effective input-to-exit mapping policies needed for dynamic inference. For HADAS, we consider an *ideal* mapping of inputs to exit branches. Where every input is mapped to the first exit module x_i that can classify it correctly. We use this mapping policy to compute the score of each exit in eq. (3.9), based on $\mathcal{N}_i$ – the *true* fraction of correctly classified samples. Models from HADAS are also compatible with any class of runtime controllers existing in the literature (e.g., entropy-based [169, 216] or reinforcement learning methods [246]). We refer the reader to this survey paper for more information about existing dynamic runtime controllers [82].

Table 3.2 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

3.7 Evaluation Methodology

In this section, we provide a comprehensive evaluation of our methodology. We first detail the experimental setup and then discuss and obtained results.

3.7.1 Experimental Setup

We implement HADAS on top of the AttentiveNAS framework [220]. To construct $\mathcal{B}$, we reuse their search space, which contains more than 2.94×10^{11} neural networks generated by scaling different dimensions such as input resolution, depth, and width as detailed in Table 3.2. 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 (5^{th}) layer to the backbones' last layer (For AttentiveNAS [220], potential exit positions are set after their "MBConv" layers).

We evaluate our approach on four (04) different HW configurations 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` and variations of operating clock frequencies to generate $\mathcal{F}$ subspace as detailed in Table 3.2. 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 an 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 setup replaced the HW-in-the-loop, such as a surrogate prediction model [198] or zero-shot learning [92]

3.7.2 Co-optimization Results

In this section, we provide more details on the obtained results of our HADAS framework. In the following, we discus the results regarding the co-optimization efficiency and DyNN models optimality.

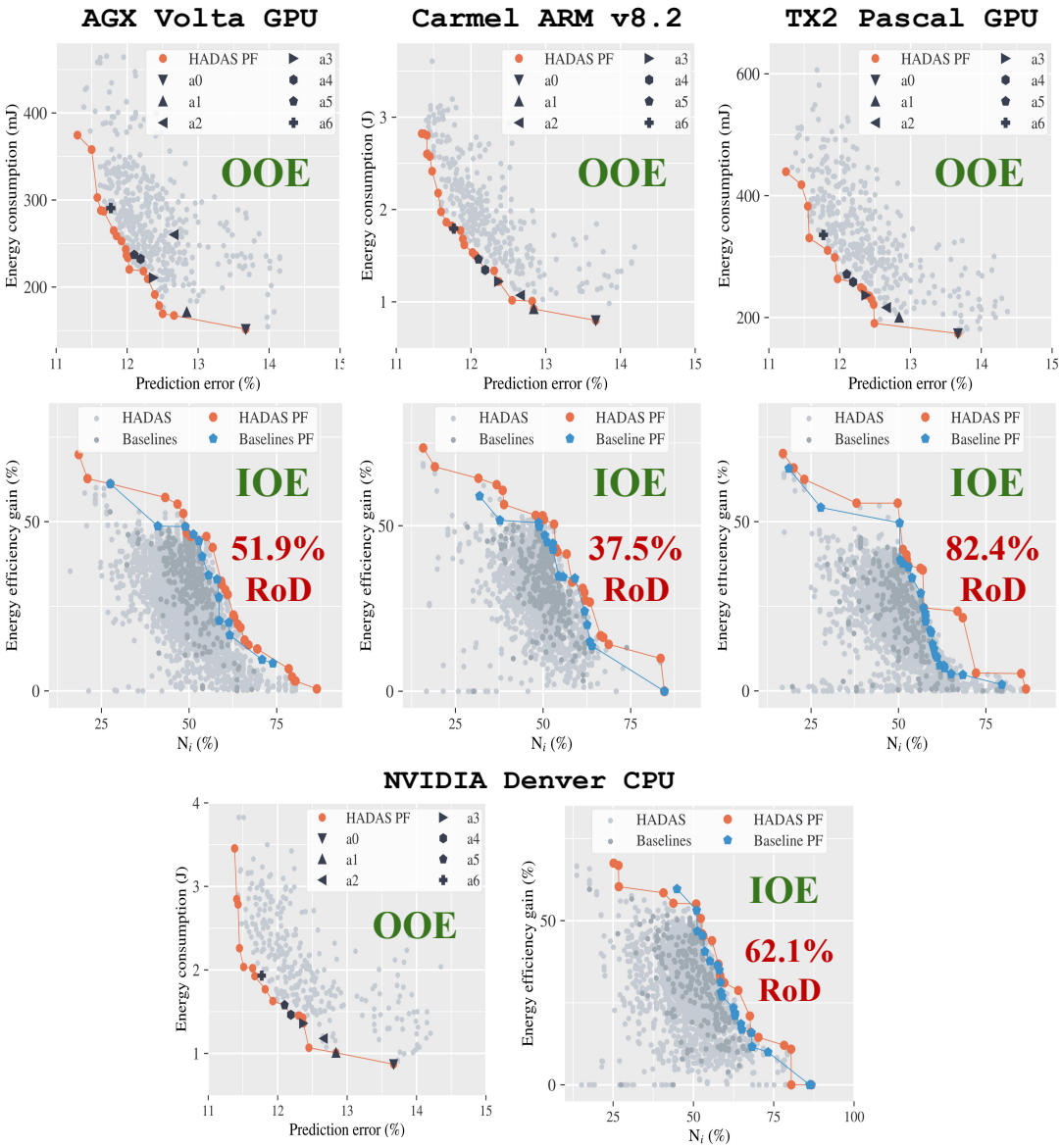


Figure 3.6 The results of the OOE and IOE on 4 hardware settings of (*from left to right*): a) `AGX Volta GPU`, b) `Carmel ARM v8.2 CPU`, c) `TX2 Pascal GPU`, and d) `NVIDIA Denver CPU`. The points in the 'OOE' depict the static performance of the explored backbones in ($\mathcal{B}$) by the OOE, without early-exit or `DVFS`. The points in the 'IOE' represent the performance of the explored combinations of backbones, early-exits, and `DVFS` in ($\mathcal{B}, \mathcal{X}, \mathcal{F}$) by the IOE.

① **OOE Results Analysis**:

The top row of Figure 3.6 compares the static performance results from the OOE of HADAS against those of the top models from AttentiveNAS [220] (denoted as **[a0-a6]**). After a fine-tuning step of the supernet provided by AttentiveNAS [220], we notice that the accuracy ranking across subnets is preserved. That is the reported accuracy values of the baselines (**[a0-a6]**) are positively correlated to the values we found on the CIAFR100 datasets. The rationality behind this is twofold:

— **(i)** Pretrained supernets on the largest classification dataset (i.e., ImageNet-1k) are capable of learning optimal representations on smaller datasets too (e.g., CIFAR100) [109].

— **(ii)** The supernet design provided by AttentiveNAS [220] and the adopted knowledge distillation training approach [219] and Sandwich sampling rule [253] help in preserving the ranking of subnets during the fine-tuning process. Thus, we pick the top-performing baselines of AttentiveNAS on ImageNet-1k (i.e., **[a0-a6]**) for comparison on the CIFAR100 dataset.

On the hardware side, we note that the latency and energy values are reported under a `MAXN` DVFS settings in which the operating clock frequencies on all the hardware components are set to the maximum supported value.

Figure 3.6 shows that 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 same accuracy level. Similarly, **a1** is dominated by another backbone from HADAS with an accuracy improvement of $\sim$ 2.34% under the same energy gain. More interestingly, The obtained Pareto fronts differ from one hardware setting to another, further emphasizing the importance of performing a hardware-agnostic neural architecture search to design backbone networks.

② **IOE Results Analysis**:

The results of the IOE are shown in the bottom row of Figure 3.6. The results of HADAS depict the outputs of the IOE when executed for various backbone designs sampled by the OOE. The results of the baselines are the combined outputs of IOE executed independently on the seven (07) baselines from AttentiveNAS [220]. 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 (3.9).

Across the four (04) hardware settings, HADAS seemingly dominates the majority of the optimized baselines with an average ratio of dominance $\sim$ 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 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 $\sim$ 63% for one of the extreme dynamic models on the Pareto frontier of HADAS, compared to $\sim$ 52% for the extreme dynamic variant from the optimized baselines, under the same level of accuracy.

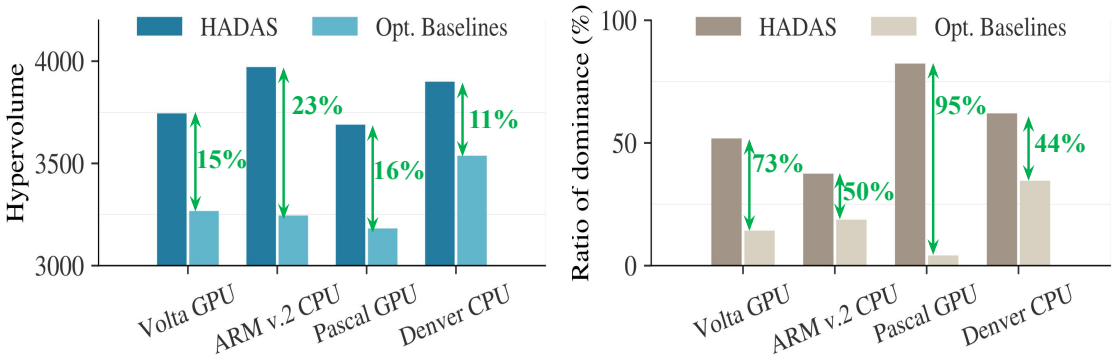


Figure 3.7 Comparing search efficacy for HADAS and the optimized baselines with regards to: a) hypervolume (*left*) and b) ratio of dominance (*right*)

③ **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 based on a reference point that depicts the worst performance values (See equation (3.6)), whereas the latter measures the percentage of solutions found by HADAS that dominate the optimized baselines (and vice-versa). Figure 3.7 shows that HADAS consistently outperforms the optimized baselines with regard to both metrics across the four (04) 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. We also remark that the coverage of the Pareto front is not necessarily correlated to the quality of the solutions. For instance, the PF on the TX2 Pascal GPU has few options that completely dominate the baselines and still cover the two extremes of the baseline Pareto front, thus depicting comparable hypervolume results to the baselines while dominating $\sim$ 95% of them. The opposite can also be observed on the ARM v.2 CPU in which the PF has higher coverage than the baselines ($\sim$ 23%) while only dominating $\sim$ 50% of them. Hence, when analyzing the results of a multi-objective optimization, it's worth reporting different metrics to better understand the observed results.

④ **DyNN Models Comparison**:

In Table 3.3, we compare the top DyNNs obtained by HADAS with 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**.

Table 3.3 DyNNs Comparison on 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

3.7.3 Dissimilarity Ablation Study

We perform an ablation study to investigate the impact of the dissimilarity term ($dissim^{\gamma}$) in equation (3.9) 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 3.8, we compare the results obtained with and without the dissimilarity with different values of γ. As shown, including the dissimilarity term allows the optimization algorithm to focus more on exploring dissimilar early exits with a high contribution to the prediction accuracy. For instance, in the right of Figure 3.8, we find that including dissimilarity improves *RoD* by ∼41%. Moreover, the extreme Pareto models with dissimilarity are ∼43% and ∼52% more accurate and energy efficient than those without dissimilarity.

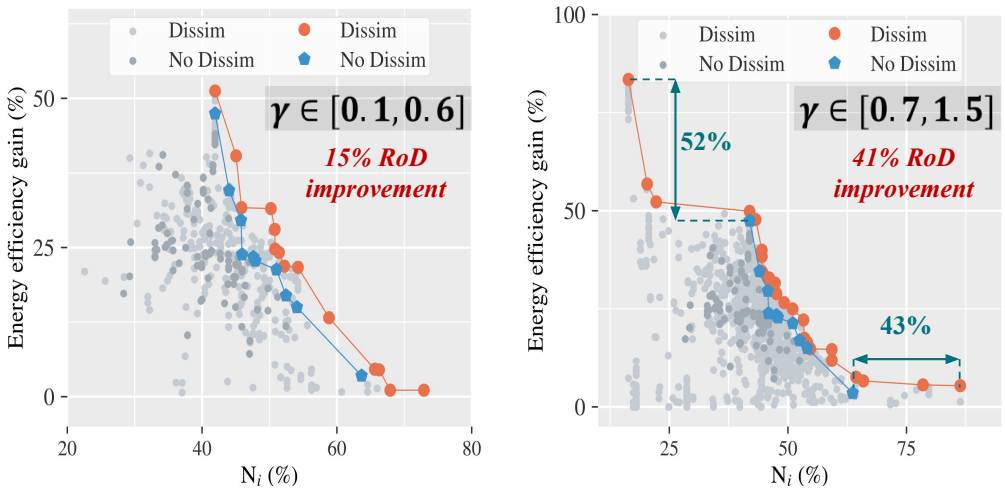


Figure 3.8 Inner optimization improvement by regularizing the exits scores with the dissimilarity function $(dissim)^{\gamma}$ over two ranges of γ values

3.8 Summary

Real-world data are characterized by high entropy ranging from easy to complex data samples. Given the intricate nature of neural networks, complex data samples require processing through many neural operations to extract meaningful features to predict the output results. On the other hand, easy data samples do not require many neural operations, and their outputs can be predicted with low

reasoning and effort. This adaptability to the input data complexity can be translated into an early-exit dynamic neural network (DyNN) where computations can be terminated once the output result can be correctly predicted.

In this chapter, we have shown how critical the design of such networks is as it involves many levels of design parameters:

1. First, we emphasize the importance of the backbone NN design as it primarily extracts the coarse and fine-grained features from the input data. We thus conduct a design exploration for the backbone to find tailor-made networks for early-exit DyNN.
2. Second, We extend existing search spaces for early-exit branches such as the one introduced in [166] by making the number and placement of intermediate classifiers fully searchable. We propose a new loss function to train the intermediate classifiers in a once-for-all fashion using knowledge distillation and dissimilarity metric to select a subset of exits with high dissimilarity to minimize the early-exit decision at runtime.
3. Third, we incorporate the dynamicity on the hardware level through scaling clock frequencies via `DVFS` that can serve as a means to balance latency energy of the DyNN inference or to emulate the availability of computing resources where low frequencies depict less available computational resources and vice-versa.

We believe that our proposed framework, HADAS, is versatile and compatible with any backbone network search, early-exit paradigm, or edge hardware device, as we make our optimization engines less dependent on the input specifications while serving the said purpose. Through HADAS, large agile models can be realized with energy efficiency similar to compact models. We have shown that HADAS DyNNs can achieve up to 57% energy gains while retaining desired accuracy levels. Future works can investigate the generality and flexibility of HADAS with respect to other strategies of dynamic inference. For instance, width-based early-exit [22], computation skipping, and dynamic routing [82]. Furthermore, our framework can also be generalized to other hardware architectures (e.g., FPGA) that provide more freedom regarding reconfigurability.

Chapitre 4

MaGNAS: A Mapping-aware Graph Neural Architecture Search Framework for Heterogeneous MPSoC Deployment

4.1 Introduction

Due to their inherent capacity to learn 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 [232], 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 shown in Figure 4.1. Recent works using 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 [81, 224, 241, 242]. 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 [256, 147].

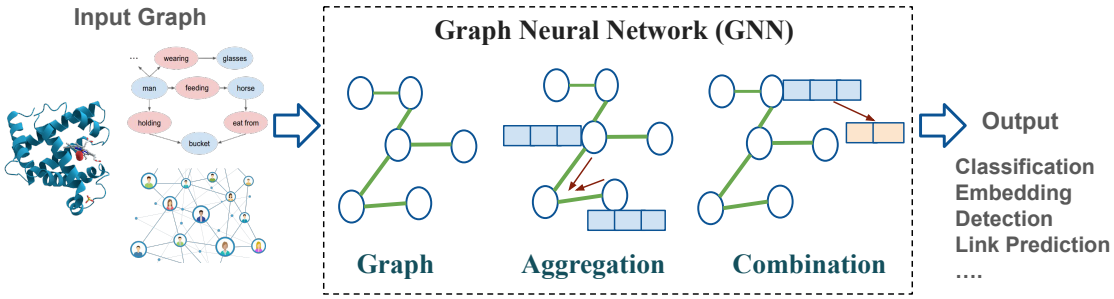


Figure 4.1 An Overview on Graph Neural Netwworks (GNN)

Recent advances have seen a proliferation in heterogeneous Multi-Processor System-on-Chips (MPSoCs) architectures that can balance the low-latency and energy efficiency requirements of compute-intensive workloads. Commercial MPSoC platforms from Nvidia Jetson series [162, 160, 163] and Tesla FSD [211] have successfully integrated a variety of proven hardware computing units (CUs) and industrial Intellectual Properties (IP)s on a single chip. For instance, NVIDIA Jetson AGX Xavier [162] comprises latency-efficient GPU and energy-efficient Deep Learning Accelerator (DLA) within the same die and sharing the same system memory. Other platforms, such as Xilinx Versal FPGA [66], enable even more flexibility in MPSoC solution development by supporting customized hardware design choices. Through such advanced platforms, deep learning-based vision modules

can be run effectively in an edge computing setting to meet stringent application requirements such as object detection for autonomous driving [131]. By extension, any consideration for applying GNNs in these vision modules under the embedded deployment setting must ensure that the execution constraints are still satisfied.

However, this objective is challenging, considering the discrepancy between the GNN workloads and the underlying hardware in heterogeneous MPSoC. Contrary to dense, regular workloads of typical DNNs, GNNs are characterized by an *irregular, multiphase sparse-dense* computational flow [70]. Notably, 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 execution sequence.

A plenty of research works have dedicated efforts to design customized GNN accelerators that can support the *multi-phased* computational flow [36, 250, 205, 240, 10, 108]. Generally, the approach entailed a *hybrid* architecture comprising specialized computing engines to accelerate each of the two phases separately. Unfortunately, these designs are not flexible enough to be consolidated into standard MPSoCs. On the one hand, this is attributed to GNNs belonging to a rapidly evolving field in which highly specialized accelerators may not support new emerging GNN operations and models. On the other hand, physical restrictions and low-power requirements of critical embedded computing platforms at the edge *restrict* the integration of new specialized hardware CUs onto existing MPSoC to the components that best serve the desired target applications. To address these challenges, another research direction involves investigating what optimization opportunities exist – on both the hardware and algorithmic levels – to alleviate the deficiencies of GNNs' computational flow when deployed on conventional CUs.

Researchers in [70] have assumed this perspective by characterizing the design space of *dataflow* choices for running GNNs on conventional re-configurable spatial accelerators, where they studied the costs and benefits of adopting various dataflows for GNNs. In that same spirit, we also believe there are ample optimization opportunities through characterizing the combined design space of *SoC mapping options* and *GNN architectural parameters* together.

In the context of GNNs for vision applications, two considerations motivate this hypothesis:

1. Heterogeneous MPSoCs naturally offer *pipelining parallelism* opportunities, presenting options to run GNN kernels of diverse characteristics on different CUs to potentially yield better performance benefits.
2. The recently proposed VisionGNN (ViG) architecture [81] proposes to transform an image frame into a graph by dividing it into equally-sized patches and constructing a graph out of them to be processed by the model. As will be detailed later, the key advantage of this scheme is that it enables leveraging graph-level features while maintaining a consistent, dense structure for any graphed image throughout the GNN model, which is more amenable to CUs than sparse graphs of *inconstant* dimensions.

Addressing the aforementioned challenges, we propose, *MaGNAS*, a new end-to-end SW/HW co-optimization framework for vision GNNs on heterogeneous MPSoCs. MaGNAS introduces a unique design space for vision GNNs, crafted by altering graph operations like aggregation and combination, and presents a detailed system model for efficiently mapping GNN workloads on MPSoCs. Through hierarchical design space exploration, MaGNAS jointly optimizes GNN design with their workload mappings on the heterogeneous computing units of the MPSoC. We validate our proposed framework on various datasets for image recognition and hardware settings. Evaluation results have demonstrated the effectiveness of our approach by providing ∼1.57× latency speedup, ∼3.38× energy efficiency for several vision datasets executed on the Xavier MPSoC vs. the GPU-only deployment while sustaining an average 0.11% accuracy reduction from the baseline ViG models [81]. Contributions and results of this chapter have been published in:

— [165] Mohanad Odema*, **Halima Bouzidi***, Hamza Ouarnoughi, Smail Niar, Mohammad Al Faruque. MaGNAS: A Mapping-Aware Graph Neural Architecture Search Framework for Heterogeneous MPSoC Deployment. ACM Trans. Embed. Comput. Syst. 22, 5s, Article 108 (October 2023), Special Issue: **CASES Conference at ESWEEK 2023**.

4.2 Related Works

4.2.1 GNNs for Computer Vision

Through learning graph-level features, GNNs achieved remarkable performance on a variety of computer vision tasks, such as activity recognition [241] and point clouds classification [118, 224]. Traditionally, the success of GCNs in computer vision applications relied on the graph construction technique, which in many cases is tailored to suit the input data semantics and downstream task. Scene graph generation [147, 235, 256] 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 [81], a generic architecture upon which our framework is constructed, represents a standard GCN backbone to generate and process graphs from raw images to serve general computer vision applications.

4.2.2 Hardware Acceleration for GNNs

The two phases of GNN *Aggregation* and *Combination* favor different classes of hardware accelerators:

1. GNN acceleration favors MIMD architectures to address the irregularity of graph operations by providing high random access memory bandwidth and small data access sizes.
2. DNN acceleration is achieved through SIMD architectures for exploiting data locality through caches or local scratchpads. Numerous works [240, 10, 250, 205, 36, 108] 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. However, such proposed accelerator designs are acutely specialized ASICs, complicating their integration into numerous hardware platforms and MPSoCs.

3. Since GNNs are becoming increasingly popular, recent research efforts [70] have directed their approach towards characterizing the design space of dataflow for GNNs on reconfigurable spatial accelerators to identify convenient dataflows for various GNN use cases.

The philosophy behind our method follows the latter trend. However, it is complementary to both approaches since it abstracts the underlying accelerator architecture and adds another layer of design space exploration to characterize joint search space of GNN architectures and the inter-phase pipelining across heterogeneous computing components in an MPSoC.

4.2.3 Distributed Computing of GNNs

Distributing DNN workloads across the heterogeneous computing resources of CPU, GPU, DLAs, and FPGAs, is an active field of research [49, 22, 176, 107, 239]. Researchers have recently explored how to distribute GNN workloads to enhance performance by exploiting the underlying heterogeneous hardware composition via task-level, data-level, and pipelining forms of parallelism [37]. For instance, the work in [260] proposed to decouple GNNs onto CPU-FPGA heterogeneous platform to speedup GNN inference.

4.2.4 Graph Neural Architecture Search

Recent research works investigated how to leverage the power of Neural Architecture Search to automate the design process of GNNs. Earlier works adopted search approaches like Reinforcement Learning [68, 67, 268] or Evolutionary algorithms [195]. The work in [251] further proposed a generalized GNNs' design space with a knowledge distillation method from GNN model-task pairs. However, these approaches mostly fall under the training-in-the-loop NAS category. Furthermore, limited or no awareness of the underlying hardware computing platform capabilities was taken. More recent works in [262, 267] proposed to move towards the once-for-all approach [30], which employs a supernet that characterizes the search space of the GNN architectures. Specifically, the training of the supernet can be conducted only once by leveraging the property of weight-sharing.

Table 4.1 Comparison between related Graph NAS works and ours.

Graph NAS work	[68]	[67]	[268]	[195]	[262]	[267]	MaGNAS (ours)
Training-in-the-loop NAS	✓	✓	✓	✓			
Once-for-all NAS					✓	✓	✓
Vision GNN							✓
Hardware Awareness					✓	✓	✓
GNN-Hardware co-design					✓		
Edge Computing Setting						✓	✓
Distributed Mapping							✓

On the hardware side, [262] adopts a co-design NAS approach for GNN and hardware accelerator, whereas [267] optimizes the GNN design to suit underlying edge computing platforms. Our work falls under the same category of HW-aware NAS for GNNs as these two. However, several features distinguish this work from others: (i) Our supernet is designed to consider the emerging class of vision-based

GNNs (ViGs) ; (*ii*) Support for evaluating candidate ViG subnets during the search process based on their best mapping options that leverage pipelining parallelism across diverse computing units within the MPSoC edge platform ; (*iii*) Our two-tier search algorithm implementation allows the inner optimization engine to be extensible to other MPSoCs and GNN supernets serving other tasks. We summarize the key differences in Table 4.1.

4.3 Motivational Example

In Figure 4.2, 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 MPSoC. In this example, the backbone GNN architecture is the ViG-Small (ViG-S) from [81], the target platform is the NVIDIA Xavier AGX MPSoC, and the models are trained on the Oxford-Flowers image dataset. We choose the ViG-S variant from [81] as it provides the optimal compromise between accuracy and efficiency for edge devices. Given the fact that the ViG belongs to the Graph Convolutional Network (GCN) class of GNNs, we construct three (03) additional variants of the baseline ViG-S with different GCN operators. Specifically, the original ViG architecture employs the `Max-Relative` Graph Conv (`MRConv`) graph operation throughout all the model, whereas the variants employ other GCN layer types, namely `EdgeConv`, `GIN`, and `GraphSage`. By using different variants, we aim to showcase how varying the graph operations would impact the accuracy-efficiency balance. 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 component of the MPSoC was used as the target deployment hardware.

In the *right* Figure, we showcase how additional performance trade-offs are attained considering the various deployment options for the ViG variants on the MPSoC. 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, which is 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 can be taken when considering a broader design problem. That is, combining the GNN *architectural* and *mapping* optimizations to

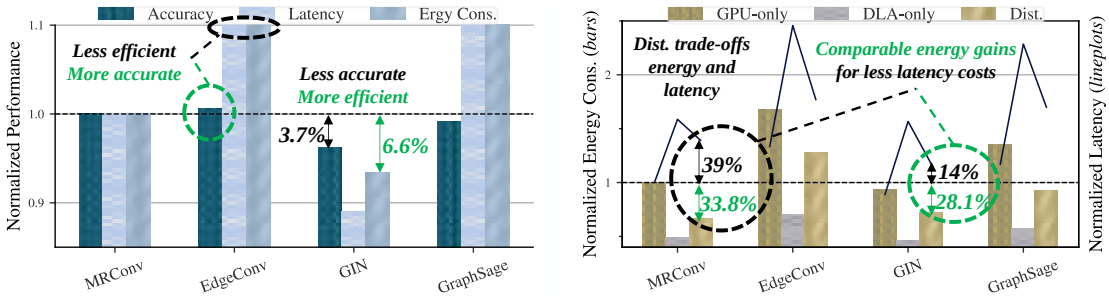


Figure 4.2 Comparing ViG model variants [81] with different graph learning operators when trained on the Oxford-Flowers dataset and deployed onto the NVIDIA Jetson AGX Xavier MPSoC. All values are normalized by the baseline performance evaluations incurred by the original ViG with MRConv layers when 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.

achieve better performance trade-offs. 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, and the question becomes *how can we better characterize this combined architecture-mapping design space to attain better performance trade-offs for vision GNNs given the target task and MPSoC platform.*

4.4 Novel Scientific Contributions

In light of the above challenges, we list the key contributions of this chapter:

— We study how vision GNN (ViG) can leverage distributed deployment across multiple CUs when deployed onto heterogeneous MPSoCs.

— We present **MaGNAS**, a Mapping-aware Graph Neural Architecture Search Framework for *co-optimizing* the design of vision GNN (ViG) and their mappings on Heterogeneous Multi-Processor Systems on a Chips (MPSoC).

— MaGNAS first contributes a self-contained framework for designing ViG supernets to characterize their search space of GNN-based architectural design choices.

— We derive a system model that characterizes the distributed deployment of GNNs onto Heterogeneous MPSoC and the incurred performance overheads.

— To identify optimal *ViG model-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 model design choices ;

an *inner* engine to identify optimal mapping for ViG workloads onto heterogeneous CUs.

— We conduct extensive experiments, in-depth analysis, and ablation studies on MaGNAS using a real MPSoC platform and hardware simulator on four (04) state-of-the-art vision datasets. Our findings have demonstrated the superiority of MaGNAS in designing and mapping ViG architectures onto heterogeneous CUs and its effective scaling capabilities on increasing levels of problem complexity. On the Nvidia Xavier MPSoC, MaGNAS provided on average **1.57**× latency speedup and **3.38**× more energy gains than the GPU-only deployment while sustaining an average **0.11%** accuracy drop from the baseline.

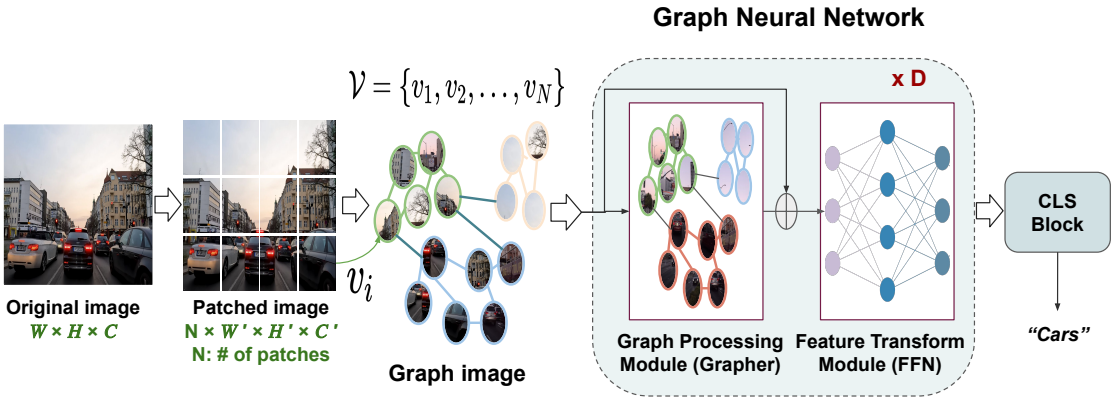


Figure 4.3 An overview of the Vision Graph Neural Network. An image is seen as a graph of nodes. The ViG has two primary missions: **(i)** Capturing the contextual dependency between the image graph nodes through the Grapher module and **(ii)** Extracting the long-range features of the graph nodes through the FFN module.

4.5 Vision Graph Neural Network (ViG)

We briefly describe the main constituents of the ViG architecture [81], which pioneered a generic approach for graph-based image processing through modeling raw input images as graph structures.

① **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.

② **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 operation on $\mathcal{G}$ can be represented by the following abstract formula:

$$\mathcal{G}' = Combine(Aggregate(\mathcal{G}, W_{agg}), W_{comb}) \tag{4.1}$$

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 $\mathcal{G}' = G(X')$.

③ **Grapher and FFN Modules.** To enrich feature representation, graph processing layers can be interleaved with typical DNN layers in a GNN model. The standard ViG architecture comprises a stack of two basic building blocks: *Grapher* and *Feed Forward Network (FFN)* given by:

$$L^{Grapher} = l^{post} \circ l^{comb} \circ l^{agg} \circ l^{pre}, \qquad L^{FFN} = l^{fc_2} \circ l^{fc_1} \tag{4.2}$$

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.

4.6 Problem Statement

In this section, we model the mapping problem of GNN workloads onto heterogeneous MPSoC. We first provide a comprehensive characterization of each computing block involved in the GNN inference pipeline. Then, based on this characterization, we implement performance prediction models to estimate hardware-related metrics (i.e., latency and energy) of each GNN workload. We note that we build our performance models on a layerwise granularity. Finally, Then, we derive a formulation for the design-mapping optimization problem. More details are provided in the following sections.

4.6.1 System Model for Mapping GNNs onto Heterogeneous MPSoCs

In the following, we detail the system models that we adopted to characterize the distributed computing of GNN over heterogeneous MPSoC.

① GNN Workload Characterization

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

$$\alpha = L_n \circ L_{n-1} \circ \cdots \circ L_1,\ s.t.\ L_i \neq L_{i-1},\ L_i \in \{L^{FFN}, L^{Grapher}\}, \\ L^{FFN} \in \{L^{FFN}, \phi\}\ \forall 1 \leq i \leq n \quad (4.3)$$

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.

Now, 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:

$$X_{j+1} = L_j(X_j)\ \ s.t.\ \ x_k^j \in \mathbb{R}^{D'}\ \forall\ x_k^j \in X_j \quad (4.4)$$

where the condition ensures that feature embedding dimensions remain consistent at 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. This consistency in the feature embedding dimensions is typical of GNNs as it preserves the integrity of graph operations with regards to feature aggregation from farther nodes across multiple consecutive layers and facilitates supporting residual and dense connections [251]. Note that D' can either be equivalent to D or a downsampled version of it as some architectures (e.g., Pyramid in [81]) can include additional downsampling layers in-between stacks of 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 with varying degrees of support for DNN and graph operations. Considering a *blockwise* granularity, we can define a mapping vector, m, to characterize the workload distribution for each GNN computational block as follows:

$$m = [\pi_1, \pi_2, \cdots, \pi_n],\ \ s.t.\ \ \pi_i \in \mathbb{CU}\ \forall\ 1 \leq i \leq n \mid support(\pi_i, L_i) == True \quad (4.5)$$

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

② Performance Modeling

For a mapping strategy m, the total latency and energy consumption overheads, T_{total} and E_{total}, experienced by a GNN model when deployed in a distributed, pipelined fashion can be modeled as the sum of the overheads incurred by

its individual blocks:

$$T_{total}(m) = \sum_{i=1}^{n} T_i(m), \quad s.t. \;\; T_i(m) = \tau_i^{comp} + \mathbb{I}[\pi_{i-1} \neq \pi_i] \cdot \tau_i^{in} + \mathbb{I}[\pi_i \neq \pi_{i+1}] \cdot \tau_i^{out} \tag{4.6}$$

$$E_{total}(m) = \sum_{i=1}^{n} E_i(m), \quad s.t. \;\; E_i(m) = e_i^{comp} + \mathbb{I}[\pi_{i-1} \neq \pi_i] \cdot e_i^{in} + \mathbb{I}[\pi_i \neq \pi_{i+1}] \cdot e_i^{out} \tag{4.7}$$

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}, τ_i^{out} and e_i^{in}, e_i^{out}, are the latency and energy overheads sustained when loading and writing back graph features from and to the *shared system memory* on the MPSoC, 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.

③ Mapping Problem Formulation

Define $P(m) = f(T_{total}(m), E_{total}(m))$ 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 MPSoC platform as follows:

$$m^* = \arg\max_{m \in \mathbb{M}} P(m), \quad s.t. \;\; T_{total} < T_{TRG}, \; E_{total} < E_{TRG} \tag{4.8}$$

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.

4.6.2 Nested Search Formulation

As the application of graph learning on embedded hardware is a relatively nascent field, the lack of standardization in GNN architectures for edge deployment settings adds another dimension to this design optimization problem. Together with the mapping formulation derived above, a natural question arises as follows: *Given an awareness of the ideal mapping strategy for a GNN onto a heterogeneous MPSoC, can we leverage this information to guide further architectural design optimizations such that the target task accuracy and resource efficiency are enhanced ?*

In light of this proposition, we refine our formulation to an *architecture-mapping* co-optimization problem, where the goal is to identify the optimal set of design choices for the GNN architecture and its mapping strategy. Since a Cartesian product of their combined search parameters can result in a large search space, we designate two separate subspaces to be managed through a bi-level optimization approach as follows: a) GNN architecture subspace ($\mathbb{A}$) ; which describes the set of architectural design choices associated with the GNN model, and b) 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:

$$\alpha^* = \arg\max_{\alpha \in \mathbb{A}} \psi[Acc(\alpha), P(m^*|\alpha, \mathbb{CU})] \tag{4.9}$$

$$s.t.\ m^* = \arg\max_{m \in \mathbb{M}} P(m|\alpha, \mathbb{CU}) \tag{4.10}$$

where the outer optimization equation (4.9) 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 contingent upon the results from the inner optimization equation (4.10). That is, energy and latency performance evaluations used for scoring a candidate architecture, α, are those obtained for an optimal mapping strategy, m^*. Due to the conflicting nature of the involved objectives, the problem can be solved as a multi-objective optimization providing a Pareto-optimal set of solutions. For instance for the outer optimization objective, an architecture α^* is said to be Pareto-optimal iff for every objective $u \in U$:

$$u_k(\alpha^*) \geq u_k(\alpha) \forall k, \alpha \text{ and } \exists j : u_j(\alpha^*) > u_j(\alpha) \forall (\alpha) \neq (\alpha^*) \tag{4.11}$$

4.7 Proposed Approach

To solve the above GNN *architecture-mapping* co-optimization problem, we present MaGNAS, a mapping-aware Graph Neural Architecture Search framework for heterogeneous MPSoC deployment. MaGNAS employs two phases: ❶ the construction and training of a ViG supernet to attain a design space of diverse GNN architectural design choices; ❷ the development of a two-tier evolutionary search framework to *identify* optimal *architecture-mapping* pairings.

4.7.1 Supernet Construction and Training

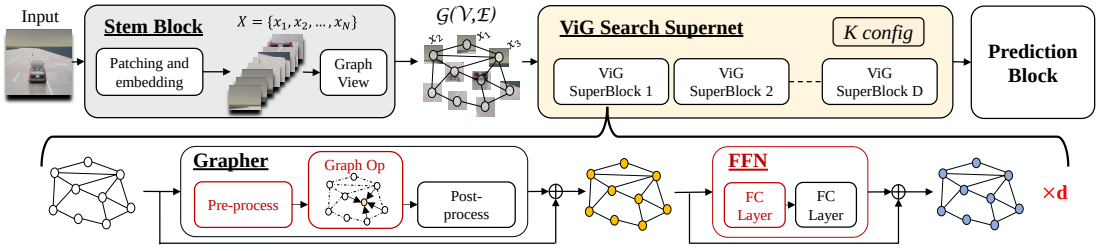


Figure 4.4 The ViG supernet implementation for MaGNAS co-search framework. The supernet comprises D ViG search super blocks, each of which constitutes a sequence of d_i Grapher and FFN computing modules. Architectural search parameters characterizing $\mathbb{A}$ subspace are highlighted in *red* and detailed in the text.

We extend the ViG architecture introduced in Section 4.5 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 [30]. 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 identify an ideal subnet that meets the target specifications. The ViG supernet is illustrated in Figure 4.4, where the choice of architectural search parameters for $\mathbb{A}$ is based on observations from both related works [251, 81, 67, 250] as well as from our initial experiments. The supernet construction is detailed in the following:

① ViG Superblocks

The backbone ViG-S architecture in [81] 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. Superblocks are inspired by the concept of neural computing blocks in popular architectures (e.g., ResNets), where the same architectural parameter value can be repeated for a stack of consecutive layers. Figure 4.4 illustrates the composition of our ViG superblock and what architectural parameters are searchable within it. 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.

② $\mathbb{A}$ 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 module.

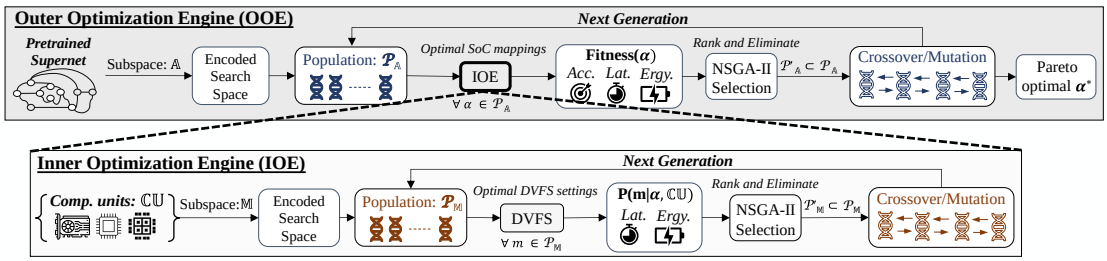


Figure 4.5 MaGNAS two-tier evolutionary search framework

We do not include the Grapher's post-processing layer as part of $\mathbb{A}$ since, in the ViG backbone, it additionally contributes to maintaining the consistency of feature embedding dimensions.

③ 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 [252, 21]. This training is performed from scratch due to: (i) The ViG is a relatively new GNN architectural concept, and the availability of pretrained weights is still limited, and (ii) loading the exact pretrained model weights from the original ViG backbone [81] can introduce a bias towards certain design choices during training. For instance, the original ViG architecture employed `MRConv` *Graph Op* throughout its graph processing layers. Thus, loading their pretrained weights gives MRConv operations an edge over the remaining *Graph Op* choices. 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 [252] as follows:

— *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 architectural features.

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 inducing a bias towards specific implementations.

4.7.2 Nested Evolutionary Search: Outer Optimization Engine (OOE)

In order to solve the bi-level *architecture-mapping* optimization problem formulated in equations (4.9) and (4.10), we construct the two-tier evolutionary search framework illustrated in Figure 4.5 to identify optimal GNN architecture and workloads mapping pairings. 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 according to their $Acc(\cdot)$ and $P(\cdot)$ evaluations.

① Subspace $\mathbb{A}$ Description

By adopting a Once-For-All (OFA) NAS approach [30], 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 one specific ViG superblock listed in 4.7.1, facilitating the sampling of subnets as GNN design candidates, $\alpha \in \mathbb{A}$ (See Figure 4.6).

② OOE Evolutionary Search

The next step is to employ a search algorithm to solve the optimization objective in (4.9) 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:

$$F(\alpha) = f(Acc_\alpha, T_\alpha, E_\alpha) \tag{4.12}$$

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 $g+1$. A uniform mutation is employed on the superblock level by sampling new depth, width, graph operators, etc., under a probability threshold of 0.4. The crossover is applied by randomly picking two individuals from the Pareto set and swapping their superblocks under a probability threshold of 0.5. This iterative search continues until the search budget expires (e.g., a given total number of generations). At the last iteration, a Pareto-optimal set, $\{\alpha^*|m^*\}$, is provided. To provide some

perspective based on 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 $\mathcal{P}'^g_{\mathbb{A}}$ for the following mutation and crossover processes.

4.7.3 Nested Evolutionary Search: Inner Optimization Engine (IOE)

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 the underlying MPSoC ($\alpha \rightarrow \mathbb{CU}$) and evaluate performance accordingly.

① Subspace $\mathbb{M}$ Description

The mapping configuration, m, defined in equation (4.5) 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 (see Figure 4.4).

② IOE Evolutionary Search

Given how the mapping decision space is at least $|\mathbb{CU}|^n$ (see equation (4.3)), a brute-force search to determine the ideal mapping, m^*, can be costly. Thus, 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 $\mathcal{P}^g_{\mathbb{M}}$, 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:

$$P(m|\alpha, \mathbb{CU}) = \left(\frac{E^m_\alpha}{max\{E^{\mathcal{CU}}_\alpha\}}\right)^{\gamma_1} \times \left(\frac{L^m_\alpha}{max\{L^{\mathcal{CU}}_\alpha\}}\right)^{\gamma_2} \quad \forall \mathcal{CU} \in \mathbb{CU} \tag{4.13}$$

where E^m_α and L^m_α 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^{CU}_α and L^{CU}_α, 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. For instance, if mapping the entirety of α onto a *GPU* component is the best option with respect to latency, then all latency evaluations are normalized by L^{GPU}_α. 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 accessible lookup tables by benchmarking computing blocks of varying architectural configurations onto the target CUs, allowing low-overhead estimations of latency and energy during the search.

Based on these evaluations, another non-dominated sorting algorithm is instantiated to rank mapping configurations, retaining the top-ranked configurations to provide population subset $\mathcal{P}'^g_{\mathbb{M}} \subset \mathcal{P}^g_{\mathbb{M}}$. Afterwards, subset $\mathcal{P}'^g_{\mathbb{M}}$ undergoes mutation and crossover to provide $\mathcal{P}^{g+1}_{\mathbb{M}}$ as the new population for the next generation.

The mutation is uniformly applied by flipping the CU for each GNN computing block under a probability threshold of 0.4. The crossover is applied by randomly selecting two individuals from the Pareto set and interchanging their CUs mapping under a probability threshold of 0.8. 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.

③ Constrained Search

To support specifying L_{TRG} and E_{TRG} as search constraints during the search procedure as in equation (4.8), we designate an additional option for the selection procedure of the IOE non-dominated sorting algorithm to filter out mapping options from $\mathcal{P}_\alpha^m$ that do not conform to the pre-specified constraints, allowing only compliant mapping options to proceed to the next stage of mutation and crossover. If there were no compliant mappings, the standalone evaluations are returned for E_α and T_α. In general, L_{TRG} and E_{TRG} can also be instated at the selection process of the OOE, where α architectures whose E_α and T_α do not meet target performance scores are eliminated from the population before the OOE's mutation and crossover stage.

④ 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 (4.4), 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 due to inconsistent graph structures.

⑤ 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. Predominantly, numerous standard heterogeneous MPSoC components integrate this feature to support a diverse set of operational modes serving different execution contexts, as in to enable switching between *low-power* and *high performance* modes. Here, to better capture the fine-grained effects of altering `DVFS` settings, 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 $\mathbb{M}$ and does not incur as much search overhead. In typical real-time operational contexts, `DVFS` settings are kept the same across all the computing blocks of α. This made a direct brute-force search through `DVFS` options sufficient to identify configurations that maximize the IOE fitness score in objective (4.13). Formally, if we denote a single set of `DVFS` configura-

tion settings as ϑ and the overall DVFS search space as Ψ, then the DVFS search objective is given as:

$$\vartheta^* = \arg\max_{\vartheta \in \Psi} P(m|\alpha, \mathbb{CU}, \vartheta) \tag{4.14}$$

where the performance evaluation of m also depends on the choice of $\vartheta \in \Psi$.

4.8 Experiments and Evaluation

In this section, we conduct extensive experiments, in-depth analysis, and ablation studies using a real MPSoC platform and SoC simulator on four(04) state-of-the-art image classification datasets. Through our experiments, we aim to assess the merit of MaGNAS in designing optimal ViG architectures and mapping them onto heterogeneous CUs. We also demonstrate the ability of MaGNAS to scale with an increasing degree of design and mapping problem complexity.

4.8.1 Experimental Setup

① GNN Supernet Design and Search Space

We build our supernet on top of the ViG-S variant [81] 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. We transform each ViG superblock into a *slimmable* neural network following [255] to support dynamic width and depth configurations. To support varying graph operations, we specify a dynamic graph processing layer in the *Grapher* with four (04) concurrent branches reflecting different GCN operational choices for *Graph Op*: 1) `EdgeConv` [224], 2) `GIN` [236], 3) `GraphSAGE` [80], and 4) `Max-Relative` GraphConv [125]. As mentioned in Section 4.7.1, the GNN search space also includes options to skip the *Grapher*'s pre-processing layer and the entirety of the *FFN* module throughout a given ViG superblock.

Table 4.2 Joint Search space of GNN architectures, Mapping, and DVFS.

Decision variables	Values	Cardinality
Supernet Search Space ($\mathbb{A}$)		
ViG Superblock depth (d)	{2, 3, 4}	3
Graph Operation (Graph Op)	{Max-Relative, EdgeConv, GraphSAGE, GIN}	4
Skip Grapher pre-process (fc_use)	{False, True}	2
Skip Grapher 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

② Datasets and Training Settings

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 cluster of 20 GPUs of Nvidia RTX 2080 Ti (11 GB).

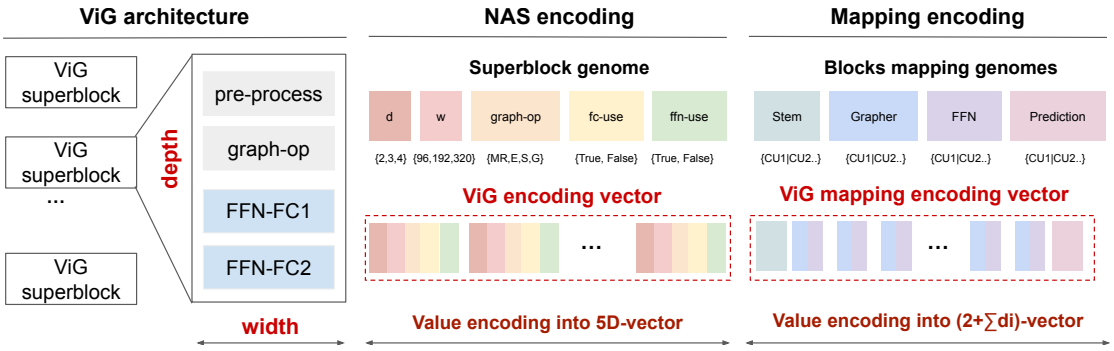


Figure 4.6 Dynamic encoding scheme employed in MaGNAS for the GNN architectural-mapping specifications.

③ Evolutionary Search Settings

Table 4.2 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. We adopt uniform mutation and crossover with respective probabilities of 0.8 and 0.4. As depicted in Figure 4.6, we employ a dynamic encoding scheme in which the IOE evolutionary algorithm changes the size of the genome vector -for the mapping strategy encoding- according to the architectural parameters of the sampled GNN to avoid sampling meaningless decision variables (e.g., mapping choices for skipped FFN and FC-pre layers). 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. The search process takes around **~1-2** GPU days to complete, depending on the complexity of each dataset.

④ Hardware Experimental Settings

We evaluate our approach using two hardware experimental setups presenting a variety of computing units and architectural features: (i) NVIDIA Jetson AGX Xavier [162], as a real target MPSoC platform; (ii) MAESTRO [116, 117], as a hardware simulator.

— **NVIDIA Jetson AGX Xavier:** We employ the NVIDIA Jetson AGX Xavier MPSoC [162] as our primary experimental testbed. The platform is equipped with a high-performance Volta GPU of 512 GPU cores and 64 Tensor cores and an energy-efficient DLA. We specify both components as the usable

computing units of $\mathbb{CU}$ and characterize them as the feasible deployment options of $\mathbb{M}$. Both components share the same 16 GB 256 bits LPDDR4x 136,5 GB/s system memory and are orchestrated by the same CPU NVIDIA Carmel Arm 64 bits. To run workloads on GPU/DLA, we use the TensorRT 8.4 compiler running on top of CUDA 11.4 and cuDNN 8.3.2. As TensorRT is limited by the set of operations that can be executed on DLA, we consider this limitation in our performance characterization by enabling the *GPU fallback* feature for the non-supported operations. The AGX Xavier also supports hardware reconfiguration of the clock frequencies of CPU, GPU, EMC, and DLA to emulate different hardware settings and power budgets, which we use to implement the `DVFS` search space Ψ. Unless otherwise stated, performance evaluations in our experiments are performed under the high-performance `DVFS` setting (`MaxN`).

— **MAESTRO:** For the hardware scalability analysis, we leverage the MAESTRO tool [116, 117] to simulate a use-case of an MPSoC with three (03) heterogeneous CUs. We express the heterogeneity by varying the dataflow configuration on each accelerator given how different neural network workloads exhibit different affinities towards dataflow choices. For example, a weight stationary dataflow (like *kcp_ws* from MAESTRO and that of the DLA in the Nvidia Xavier) maximizes filter weights' reuse which is useful for layers whose same filters are used to compute multiple outputs [43]. We use the native dataflows in MAESTRO of *kcp_ws*, *ykp_os*, and *dpt* for our 3 CUs, denoted by **DSA-k**, **DSA-y**, and **DSA-d**.

⑤ Baseline GNNs and Mappings for Comparison

The efficacy of our approach is assessed regarding the following GNN architectural and hardware mapping baseline:

— **GNN architectures baselines**: These include the original isotropic ViG-S model in [81] 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. We implemented PyramidViG-M to follow the feature dimensional reductions across stages as in [81] and fixed four (04) blocks within each superblock in the supernet (recall 4.8.1).

— **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 proposed in [49].

— **MAESTRO GNN baseline**: We use the PyramidViG-M mentioned above GIN-variant for our hardware scalability experiments using the MAESTRO simulator. For the convenience of MAESTRO, we define the GIN operation by its low-level implementations of the *aggregation* and *combination* phases.

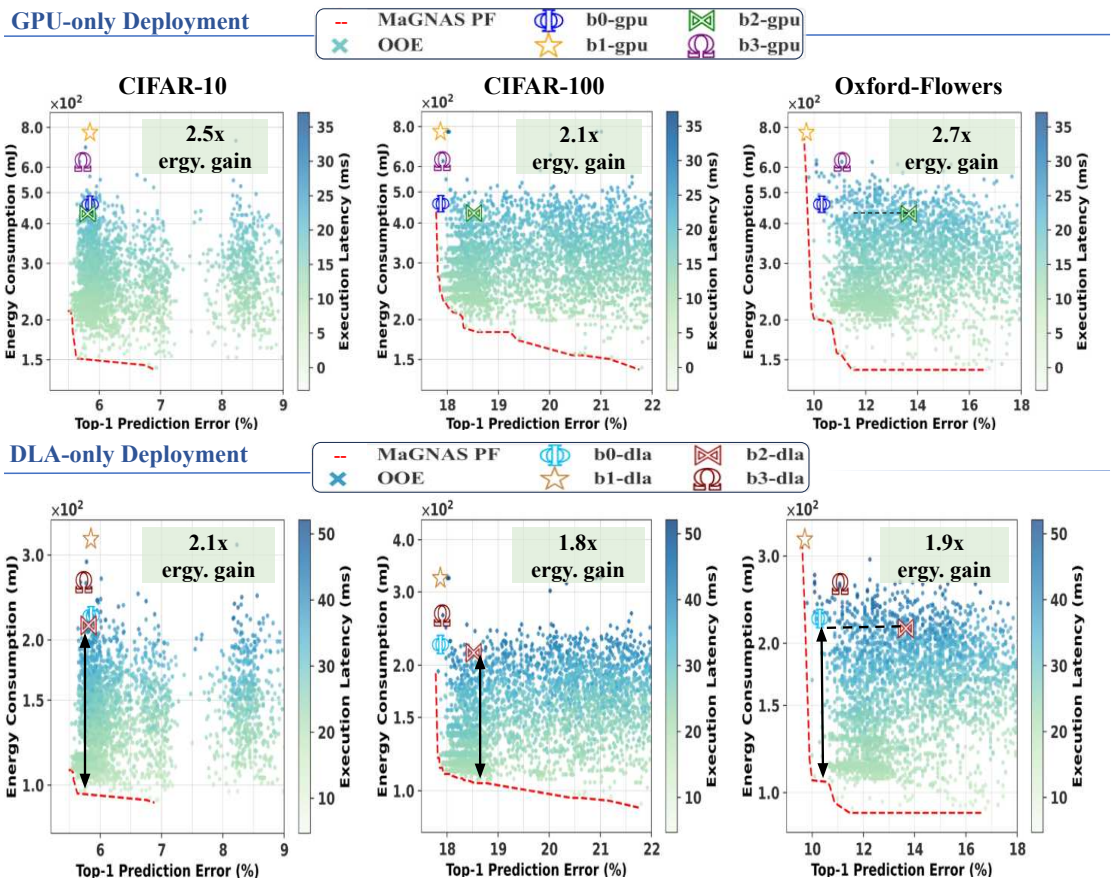


Figure 4.7 The first two rows show the performance of the explored GNNs in ($\mathbb{A}$) by the OOE on three datasets (*from left to right*: a) CIFAR-10, b) CIFAR-100, c) Oxford-Flowers. The Hardware metrics (i.e., latency and energy) are shown for *GPU-only* and *DLA-only* deployment options.

That is, the *aggregation* entails a matrix multiplication between the adjacency matrix and the feature embedding matrix, whereas the *combination* entails another matrix multiplication to transform the aggregated graph features to another representation for the following layer.

4.8.2 OOE Results: GNN Architecture Optimization

We first examine the merit of the OOE in identifying GNN architectures that can achieve favorable performance trade-offs compared to the baselines. In Figure 4.7, 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 $\sim$**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 4.8.4,

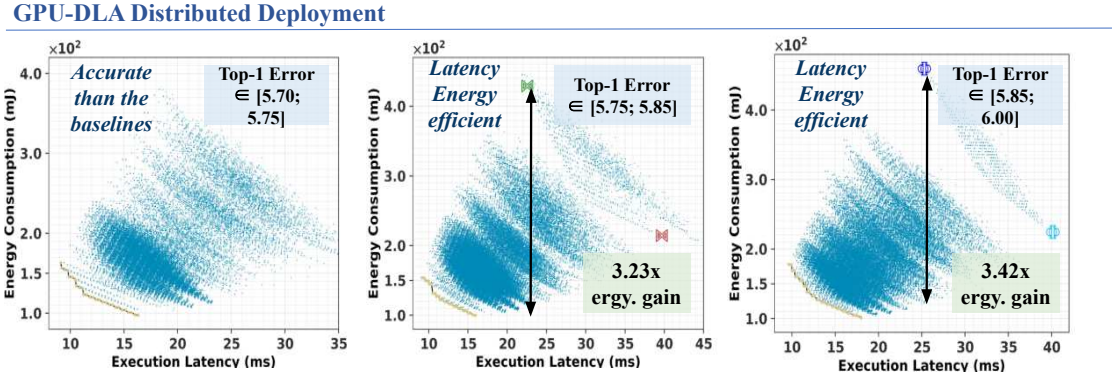


Figure 4.8 The IOE results on CIFAR-10 grouped by prediction error intervals.

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 (as specified in $\mathbb{A}$ from Table 4.2), 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 *FFN* and FC pre-processing layers.

4.8.3 IOE Results: Hardware Mapping Optimization

We further assess the efficacy of the IOE in identifying effective mapping configurations for provided GNN architectures. The bottom row of Figure 4.8 shows the optimization results when exploring mapping strategies from $\mathbb{M}$ for the *top-performing* GNN architectures (as ranked by equation (4.12)) 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 from the standalone deployment options, that is, between the GPU-only and DLA-only mapping options' latency/energy consumption values, as depicted by the middle sub-figures. Remarkably, the explored configurations form distinguishable contours, each showing a specific GNN architecture alongside its explored mapping options – represented by the different latency-energy trade-off values. Specifically, the GPU-only and DLA-only mapping configurations for each GNN architecture are located at the boundaries of its curved line. The intermediate points between the extremes illustrate the performance of the distributed deployment settings and show how each mapping configuration results in different latency-energy trade-offs.

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 $\sim$**3.42**$\times$ 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 prove that rank is preserved across GNN architectures and mapping schemes in this joint search space.

Table 4.3 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 MPSoC platform.

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

Dataset Abbreviations: **C10**: Cifar-10, **C100**: Cifar-100, **F**: Oxford-Flowers, **Ti**: Tiny-ImageNet.
Graph-Ops Abbreviations: **M**: `Max-Relative`, **E**: `Edge-Conv`, **G**: `GIN`, **S**: `Graph-Sage`.
Computing Units Abbreviations in 'Latency' and 'Energy' Columns: **G**: GPU, **D**: DLA.

4.8.4 Analysis of Search and Pareto Optimal Models

In Table 4.3, we provide a detailed analysis of performances, architectural parameters, and mapping strategies of the ViG baselines **[b0-b3]** and a selection 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 achieve better speedups and energy efficiency results. For instance, our models achieve on average ∼**1.57**× and ∼**2.49**× latency speedups; ∼**3.38**× and ∼**1.65**× more energy efficiency when compared to 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 4.3, further balance the latency-energy trade-offs by effectively utilizing different CUs.

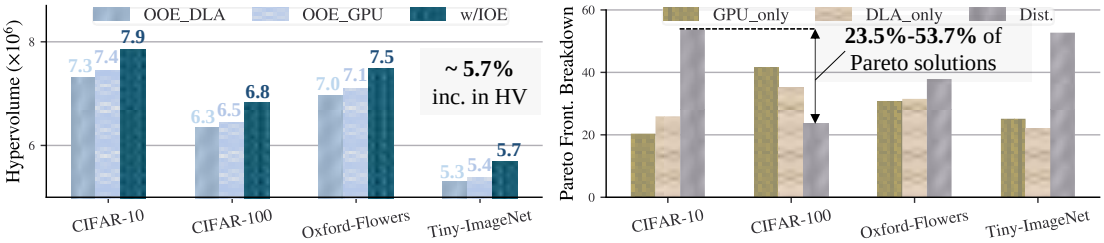


Figure 4.9 *Left*: Hypervolume analysis when including the IOE against those of the standalone OOE for the DLA and GPU. *Right*: Breakdown of the combined Pareto Fronts constituents on the basis of mapping options.

4.8.5 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 [191] 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 4.9 (*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 4.9 (*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.

4.8.6 Analysis of GNN Workload Distribution

In this subsection, we showcase how different GNN workload assignments across the GPU and DLA influence the latency-energy tradeoffs. In Table 4.4, we select one of the Pareto-optimal models, **Ours-a3** on CIFAR-100, and compare three mapping configurations: ❶ Standalone options in which the model is fully deployed on either GPU or DLA. ❷ Constrained transition options (as introduced in [49]) 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). ❸ Ours (IOE) are the mapping options provided through our IOE with unconstrained inter-CU transitions.

As no single optimal solution exists for any distributed mapping strategy, we ensure a fair comparison between our approach and the constrained transition strategies by comparing evaluations of one objective function (energy) while fixing the other (latency). Thus, for each constrained transition option, we use two (02)

Table 4.4 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. (ms)	Enrg. (ms)
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-G-D-D-D	G-G-G-G-G-G-D-D-D	D	1	17.42	226.79
constr-transit2	D	D-G-G-G-G-G-G-G-D	D-G-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**

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 4.4, we can observe that with our unconstrained mapping strategy, a single inference sustains 197.8 mJ on average, which is more 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: ❶ graph feature sizes are relatively small throughout the ViG models, leading to low inter-CU transmission overhead penalties to be experienced on the Xavier MPSoC. Our IOE optimization strategy exploited this property to identify more efficient mapping configurations with a larger number of transitions. ❷ 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 as possible before transmission costs become non-negligible.

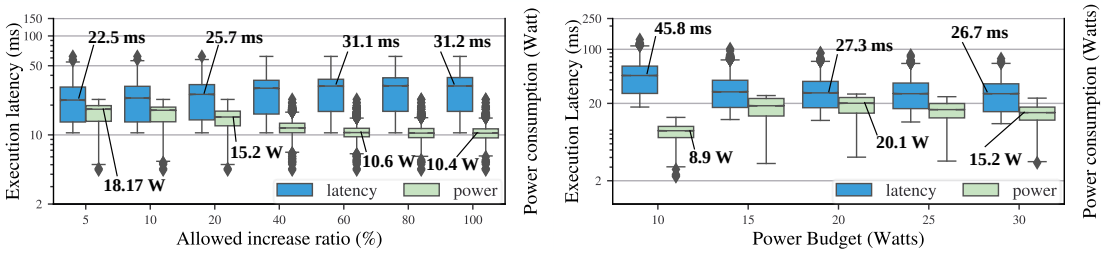


Figure 4.10 Results of the two constrained optimization: Latency and power consumption numbers are reported under variation of (*Left*) the allowable latency increase ratio compared to GPU-only, and (*right*) the available power budget. Numbers indicate median values.

4.8.7 Constraint-aware Optimization

As many embedded systems employ real-time execution requirements, we test the effectiveness of our framework when the search algorithm is performed under strict latency and power constraints. In particular, we specify two experiments, each associated with one of the following constraints: ❶ Latency, in which the constraint specifies the max allowable increase in latency compared to the *standalone* deployment option on the fastest MPSoC component (i.e., GPU-only). ❷

Power budget ; by fixing low values of clock frequencies and a limited number of CPU cores and memory speed transmission [1]. The first constraint is common for real-time systems governed by strict execution deadlines, whereas the second constraint is more common for battery-powered systems operating on limited power budgets. We conduct the two constrained optimization on the IOE using baselines [**b0-b3**] and our models [**a0-a3**] on the CIFAR-100. We report the absolute latency and average power consumption values in Figure 4.10.

Table 4.5 GNN Workload distribution under different latency constraints.

Workload Distribution	**Allowable latency increase ratio (%)**						
	5	10	20	40	60	80	100
Avg. GPU utilization	0.97	0.91	0.74	0.56	0.50	0.50	0.50
Avg. DLA utilization	0.03	0.09	0.26	0.44	0.50	0.50	0.50

① **Latency-constrained search** We characterize the latency constraint by enforcing a max *allowable increase ratio* from the fastest CU (i.e., the GPU). As shown in left Figure 4.10, low latency increase ratio ($\leq 20\%$) leads the IOE towards delegating more computation kernels to the GPU, resulting in more power-demanding mapping strategies. Compared to the soft-constraint case (i.e.,w/ tolerance of 100% increase in latency), the power demands at an allowed increase ratio of 5% are 1.75× more.

As the tolerable increase ratio rises ($\geq 30\%$), the constraint on the search is gradually relaxed. As shown in Table 4.5, the optimizer gains more freedom in exploring mapping options and favors delegating more computation kernels to the DLA for energy efficiency. The power efficiency gains start to plateau around a 50% increase ratio, indicating that the IOE has converged onto mapping strategies that maximize the fitness formula (as defined in (4.13)) by balancing latency and power efficiency. This convergence is sensible given how between the GPU and DLA, one component is roughly twice as effective as the other with regards to one performance objective, i.e., execution latency on the GPU is almost 2× less than the DLA, and the DLA incurs 2× less power consumption than the GPU (see Table 4.3) ; given that we assigned equivalent weights for the objectives in the fitness evaluation formula in (4.13), i.e., $\gamma_1 = \gamma_2 = 1$.

② **Power-constrained search** The second experiment depicted in the right Figure 4.10 shows that at tighter power budget constraints, the IOE focuses on identifying power-efficient mapping options at the expense of a slight decrease in latency, resulting in mappings that assign more GNN workloads to the DLA as depicted in Table 4.6.
We note that in this experiment, we also maintain the latency minimization as objective, which also explains the low DLA utilization ratio values reported in Table 4.6. For instance, to satisfy the 10 Watts power constraint, the IOE specifies mapping settings with a median latency of 45.8% – 1.71× more than the latency

1. Power management and clock frequency scaling : `https://docs.nvidia.com/jetson/archives/r34.1/DeveloperGuide/text/SD/PlatformPowerAndPerformance.html`

Table 4.6 GNN Workload distribution under different power budgets.

Workload Distribution	Available Power Budget (mW)				
	10	15	20	25	30
Avg. GPU utilization	0.74	0.76	0.88	0.88	0.81
Avg. DLA utilization	0.26	0.24	0.13	0.13	0.19

experienced at a power budget of 30 Watts. More latency-efficient mappings are identified with refined workload distribution as the power budgets are relaxed.

4.8.8 Ablation study on the impact of DVFS

In this experiment, we assess the merit of including `DVFS` optimization within the IOE. We reuse the baselines [b0-b3] and our models [a0-a3] from the CIFAR-100 experiment. Their mappings are kept fixed, and we run the models through the `DVFS` optimization engine to assess how performance can be further enhanced. Specifically, we consider the following `DVFS` settings: ❶ `MaxN`, which resembles the high-performance mode on the Jetson Xavier MPSoC with clock frequencies set to the maximum. ❷ `MinN`, which is an opposing best-effort mode for low-power operation in which clock frequencies are set to the minimum. ❸ `Searched`; in which `DVFS` settings are searchable within the IOE (see Table 4.2 for the values). ❹ `Default`; in which we use the default dynamic `DVFS` heuristic with CPU and GPU governors fixed to *Schedutil*, *nvhost podgov*, respectively. In this last setting, clock frequencies are dynamically adjusted at runtime depending on the underlying resources utilization, where clock frequencies are ranged from 0 to the maximum value on each component. We note that in addition to the GPU and DLA frequency variations, we also scale the CPU and EMC clock frequencies as both influence data transmissions between the shared system memory and private memories of GPU/DLA. We run the IOE with the same optimization parameters to ensure a fair evaluation.

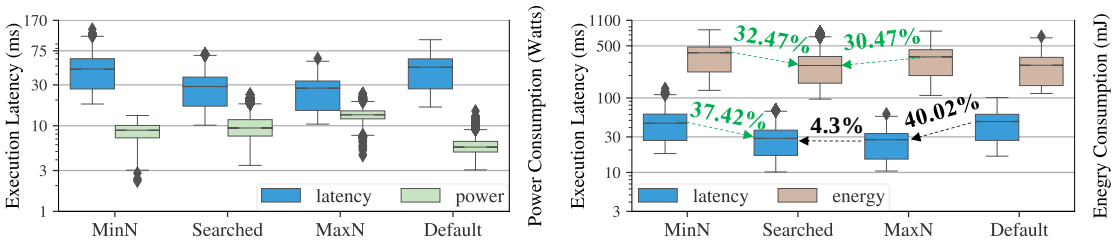


Figure 4.11 Ablation on the impact of including `DVFS` optimization within the IOE. **Searched** settings is compared against the `MinN`, `MaxN`, and `Default` settings with regards to (*Left)*: Latency-Power trade-offs, and (*Right*): Latency-Energy trade-offs. Numbers in the right Figure indicate percentage change in values.

In Figure 4.11, we illustrate the performance trade-offs as incurred by the explored (*GNN architectures* × *HW mappings*) under the 04 (four) `DVFS` settings. As expected, the left subfigure shows that the Searched mode exhibits a balanced trade-off between latency and power compared to the `MinN` and `MaxN` modes. More interestingly, however, the *Searched* setting is able to identify configurations that

yield superior energy gains to the fixed DVFS modes. In particular, the median latency and energy consumption values of Searched are **37.42**% and **32.47**% *less* than MinN, respectively. On the other hand, though Searched incurs a **4.3**% increase in its median latency compared to MaxN, it can achieve an order of magnitude more energy savings reaching **30.47**%. This implies that the IOE identified the DVFS as a viable tuning knob to enhance energy efficiency by scaling clock frequencies across the different components. Moreover, latency in Searched is improved by **40.02%** compared to the default DVFS governor. This is explained by the underlying logic of the dynamic heuristic, which only considers the hardware utilization and overlooks workload properties such as computation and memory requirements. For instance, memory-bounded workloads may benefit from GPU/DLA core downscaling with reduced energy at the same latency level. These properties are captured in our Search mode as we adjust the frequencies according to the GNN and mapping configurations. In addition, The default governors are set to avoid the idle state when the computing unit is not used, by lowering the frequency to 0, which helps in minimizing the power consumption (as shown in the left subfigure) but also worsens the execution latency as computing units usually need a warm-up stage to operate steadily after swapping between low and high frequencies.

4.8.9 Generality and Scalability

Employing an evolutionary algorithm (EA) for the IOE may seem excessive when the backbone ViG is an isotropic one that does not experience feature map sizes change and when the mapping is performed across merely 02 CUs. Thus, we perform an additional set of experiments in which we assess the scalability and generality of the IOE on the search-space levels of: ❶ *the ViG architectural backbone* ; where the supernet's backbone is implemented as a pyramid variant that allows dimension reductions from one superblock to the next (recall Figure 4.4), unlike the aforementioned isotropic one, and ❷ *the hardware CUs* ; by simulating a case with 03 heterogeneous CUs.

① On the ViG Architectural Level

Using the Nvidia Xavier MPSoC with GPU and DLA, we compare the mapping results from the IOE between the isotropic (ViG-S) and pyramid (PyramidViG-M) variants (recall Section 4.8.1). As we analyze the effectiveness of the inner EA, we fix the GNN from the OOE for both variants 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 4.12, 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. Given such a setting, a scalarization method can be sufficient to determine the Pareto front by varying the ratio of mappable workloads on either

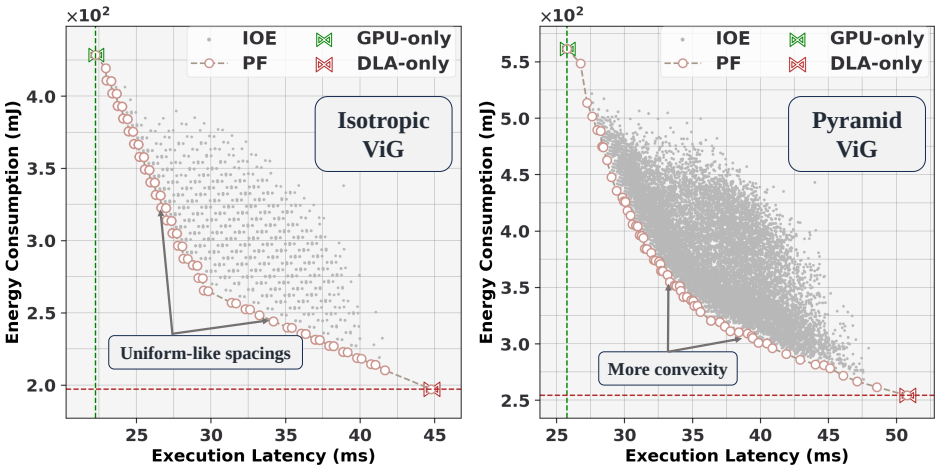


Figure 4.12 The results of the IOE EA optimization on the Isotropic Vision GNN (*left*) and Pyramid Vision GNN (*right*).

CU. 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. We observe that the sampled mapping options are more diverse in their energy and latency characterizations and that the Pareto front exhibits stronger convexity than its isotropic counterpart, reflecting a diverse, more complex mapping space.

② 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 [116] to specify 3 DSAs of diverse dataflows for CU heterogeneity (see the details in 4.8.1). As every layer within MAESTRO is defined via low-level implementations (including *aggregation* and *combination* layers), we can characterize processing overheads within PyramidViG-M on a *layerwise* basis and combine them to characterize larger blocks (e.g., Grapher). At this point, we find that each 'layer' rather than 'block' can exhibit different performance characteristics at different ViG stages. For instance, the *aggregation* sustains a substantial overhead when processing the sizable graph feature matrices at earlier blocks. This is predominantly due to the DSAs in MAESTRO not being implemented initially to support graph acceleration – similar to how numerous MPSoC platforms (e.g., the Xavier) do not widely integrate specialized graph acceleration engines. As such, we can simulate an additional case to study the mapping on a *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 an increasing level of problem complexity.

In Figure 4.13, we demonstrate how the inner EA scales effectively as the search space is expanded from the *blockwise* to the *layerwise* mapping granularity. We first specify a fixed optimization budget of 6×10^4 evaluations for both. Moreover,

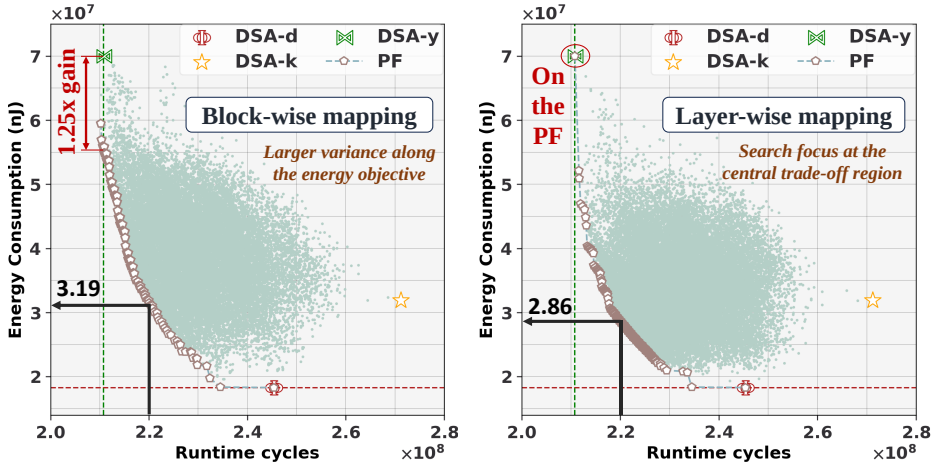


Figure 4.13 The results of the IOE on MAESTRO [116] with: i) *Block-wise mapping granularity* (*left*) and ii) *Layer-wise mapping granularity* (*right*).

although fully deploying the architecture on DSA-d completely dominates DSA-k deployment, the latter is still included since it represents the optimal mapping option for some individual layers. In the blockwise case (*left*), we observe that 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 distributed mapping options that dominate the standalone extreme, i.e., the EA identifies a distributed mapping configuration that achieves 1.25× energy gains over DSA-y for the same latency level. The opposite occurs for the *layerwise* search, where despite the much larger optimization space, 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 $\sim 2.2 \times 10^8$ cycles, the layerwise search by the IOE identified a mapping option that incurs 28.6 mJ compared to 31.9 mJ from the blockwise search.

4.9 Discussion and Key Insights

① **Key Takeaways.** Hardware-software design optimizations and workloads mappings are increasingly studied in the literature [63, 21, 49, 22]. What distinguishes this work is its specialization in considering the details of: (*i*) GNNs' computational flow irregularity; (*ii*) workload distribution across heterogeneous CUs with varying degrees of support for graph operators. Furthermore, ViG is a relatively emergent class of GNNs, and there remains room for improvement along the design, characterization, and training of ViG supernets, which can only improve as the application of ViGs – and GNNs in general – at the edge continues to proliferate. All things considered, MaGNAS has demonstrated encouraging results that can help pave the way for future lines of research.

② **Generality and Scalability.** In analyzing the generality of MaGNAS (Section 4.8.9), we have demonstrated the benefits of heterogeneity in hardware accelerators through varying dataflows across HW accelerators. In practice, heterogeneity can also occur through varying other factors such as processing engines per accelerator,

shared buffer size, off-chip memory bandwidth, etc., All of which can influence the hardware efficiency of the workloads. MaGNAS has been shown capable of generalizing to the different forms of heterogeneity as it relies on high-level performance characterization that abstracts underlying hardware compositions. Furthermore, experiments on real MPSoCs with different HW accelerators and levels of heterogeneity from that of the Nvidia Xavier is still needed to corroborate that MaGNAS can scale effectively to diverse platforms.

③ **Graph Operation Support Limitations.** As MAESTRO does not natively support the sparse matrix multiplications, we implemented GNN operations within the simulator as generic matrix multiplications, which has led to considerable execution overheads for the *aggregation* phase regarding latency and energy. This is indeed a situation akin to the case when GNN workloads are to be run on generic, uncustomized edge devices that lack proper support for specialized accelerators for GNN operations. In such cases, mapping optimizations can be particularly beneficial in mitigating the impact of such hardware deficiencies. Furthermore, as GNNs grow in popularity, promising steps are being taken towards developing new dataflows for reconfigurable spatial accelerators to support irregular graph computational sequences, which will also bring about the need for new architectural simulators to effectively model their performance overheads.

④ **Other Application Domains.** Vision-based applications provided practical, tangible use case motivations for the *GNNs-on-SoCs* scenario, and accordingly, they have become the primary target application of this work. The manner in which MaGNAS has been developed enables it to generalize to other emerging applications on edge MPSoCs that employ GNNs for their primary computational workloads. For instance, the support for mapping on both the blockwise and layerwise levels of granularity within MaGNAS enables it, with some fine-tuning, to serve other types of emerging GNN-based applications, such as scene parsing/extraction and anomaly detection, at the edge by maximizing GNNs' efficiency across a broad range of diverse CUs integrated onto the same chip.

4.10 Summary

This chapter presented MaGNAS, a mapping-aware Graph Neural Architecture Search framework for distributed deployment of vision GNN onto heterogeneous MPSoCs. MaGNAS characterizes a GNN architectural design space bound with prospective mapping options, enabling the identification of model designs optimized to the distributed deployment scheme. MaGNAS employs a two-tier evolutionary search framework to identify optimal *architecture* and *mapping* pairings that provide the best performance trade-offs. Extensive experimentation, in-depth analysis, and ablation studies using a real MPSoC platform and hardware simulation have showcased the merit of MaGNAS. Evaluation results have shown a $\sim$1.57$\times$ latency speedup, $\sim$3.38$\times$ energy efficiency for several vision datasets executed on the Xavier MPSoC vs. the GPU-only deployment while sustaining an average 0.11% accuracy reduction from the baseline ViG models [81]. We believe that our work can serve future research in SW/HW co-optimization for GNNs on the edge by investigating more application use cases, hardware paradigms, e.g., System-in-Package (SiP), and by expanding the GNN search spaces.

Chapitre 5

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

5.1 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 those of Deep Neural Network (NN) workloads, MPSoCs are designed to incorporate heterogeneous computing units (CU) within the same die, typically sharing the same system memory (DRAM) and possessing different hardware capabilities. This hardware architecture paradigm enables the collaborative usage of multiple CUs to accelerate different operations of the same or multiple applications, 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 (e.g., DLA) 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 and energy gains through maximizing the utilization of the MPSoC's resources and making better use of the HW capabilities of each CU.

Latest research has shed light on the *computation mapping* problem for MPSoC by providing comprehensive modeling methodologies in [203, 152, 237, 49] to characterize workloads performances on MPSoCs. The resulting models are generally used to map computations onto CUs in a sequential pipeline fashion. However, for workloads exhibiting a high degree of parallelism, such as NN, there is 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 NN layers such as convolutional layers in Convolutional Neural Networks (CNNs) [79] and multi-head self-attention layers in Transformers[54]. Particularly, parallelism is leveraged within NN from three (03) perspectives:

— *(i) Depthwise partitioning*: By splitting layers into less deep micro-workloads.

— *(ii) Width-wise partitioning*: By splitting the NN along the channels (for CNNs) or attention heads (for Transformers) and generate micro-workloads that are as deep as the original NN but less wide.

— *(iii) Hybrid partitioning*: By combining the two techniques mentioned above to realize less deep and large micro-workloads.

The common aim of each parallelism technique is to provide less hardware-demanding micro-workloads that are to assigned to different CUs within the context of edge-cloud systems [104, 58], distributed edge-devices [88, 78], or single MPSoC [150, 49]. By extension, prior works [149, 190, 79, 88] have also considered leveraging computation parallelism on data and task levels. Nevertheless, most works focus on model training rather than inference [58]. Although substantial studies exist for distributed edge devices [182], there is a lack of a comprehensive and holistic framework for commodity edge and tiny MPSoC. Moreover, assigning multiple workloads of the same NN application incurs high communication and synchronization overhead that sets critical constraints on the adoption of the *distributed computing* scheme for real-time execution settings [146].

On a separate note, after the success of dynamic neural Networks (DyNNs), recent works have started to explore the prospect of partitioning the NN model itself into separate computing stages that can be invoked in a *dynamic* manner. The dynamicity manifests *on-the-fly* during the inference where simpler inputs can be classified in earlier model stages, whereas the latter stages are instantiated for more complex inputs. Many dynamic inference strategies are employed in literature, such as early-exiting [21, 120], computation skipping [246, 192], and dynamic routing [155, 140]. These NN models typically exhibit a high opportunity for parallelism, especially when multiple inference stages are a priori designed and instantiated to serve inferring on various degrees of input data complexity. For instance, S2DNAS [259] demonstrated the benefits of partitioning a CNN model along its width dimension (i.e., layer's channels) and deploying the model as a multi-exit neural network supporting parallelism. ENAS4D [258] followed up by enlarging the design space of S2DNAS on the model level to encompass various architectures of the CNN backbone in order to find well-tailored backbone designs for the dynamic inference scheme introduced in S2DNAS. Nonetheless, in both works, the hardware dimension was missing, and only proxy metrics (e.g., FLOPs and Parameters) were reported, which limits their integration for existing edge devices. Furthermore, studies of mapping such parallel neural network components onto a heterogeneous MPSoC for dynamic inference have not been done in prior works.

Addressing these challenges, we propose *Map-and-Conquer*, a novel framework that leverages dynamic inference and computation parallelism and mapping on heterogeneous MPSoCs. Our approach synergistically splits an NN along its width to derive inference stages capable of inferring on input data of varying degrees of complexity. Additionally, we contribute a novel mapping technique by enabling computation parallelism between the inference stages on the different CUs of heterogeneous MPSoC. Our approach has been validated using Transformer and CNN models on the CIFAR-100 dataset, employing NVIDIA's Jetson AGX Xavier MPSoC, which includes a GPU and DLA. Our findings indicate that our dynamic configurations are 2.1x more energy-conserving than GPU-only setups and experience 1.7x reduced latency than DLA-only configurations. Contributions and results of this chapter have been published in:

— [22] **Halima Bouzidi**, Mohanad Odema, Hamza Ouarnoughi, Smail Niar, Mohammad Al Faruque. 2023., "Map-and-Conquer: Energy-Efficient Mapping of Dynamic Neural Nets onto Heterogeneous MPSoCs," 2023 60th ACM/IEEE Design Automation Conference (DAC), July 2023.

5.2 Related Works

5.2.1 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 (i.e., layers) [216] or width (i.e., channels) [259]. Recently, early-exiting is emerging to Vision Transformers (ViT) as they exhibit many computation redundancies [180, 257]. The dynamicity in vision transformers has been leveraged on tokens, heads, and encoder levels. For instance, HeatViT [54] introduced a token selector operation that progressively conserves informative tokens conditioned on features from the previous layer. Similarly, MP-ViT [122] introduced a multi-path routing strategy for patches exhibiting different information scales to be processed by adequate Transformer encoders. MIA-Former [257] dynamically adapts the number of heads in attention layers. This latter approach can also be exploited for model partitioning, as it represents the width in ViT. However, most existing works still need to catch the hardware dimension when designing a *dynamic* ViT, which is a vital factor given their complexity.

5.2.2 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 NN. AxoNN and MEPHESTO [152, 237, 49] propose modeling strategies to characterize execution latency and energy consumption for computation mapping on the AGX Xavier MPSoC. Jedi [98] provides a framework built upon TensorRT to accelerate NN via model parallelism to maximize throughput for batched inference. [103, 105] proposes evolutionary-based scheduling for NN layers on heterogeneous MPSoCs with `DVFS` by exploiting both data and model parallelism to optimize the throughput. DistrEdge [88] 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.

Table 5.1 Comparison between Related-works and ours

Related Work	Early Exiting	Model Parallelism	Collaborative execution	DVFS	Training free
AxoNN [49]			x		x
Jedi [98]		x	x		x
DistrEdge [88]		x	x		x
Kang et al. [103]		x	x	x	x
S2DNAS [259]	x	x		x	
HADAS [21]	x			x	
Edgebert [212]	x			x	x
Ours	**x**	**x**	**x**	**x**	**x**

To the best of our knowledge, our work is the first to address the problem of dynamic NN design and mapping onto heterogeneous MPSoC in a collaborative manner. Thus exploiting NN dynamicity, MPSoC heterogeneity, and reconfigurability (`DVFS`) for an energy-efficient execution on MPSocS. Table 5.1 highlights the key differences between related works and Ours.

5.3 Motivational Example

To illustrate our motivation in a more concrete context we provide Figure 5.1 that illustrates the underlying performance tradeoffs of deploying an NN onto a heterogeneous MPSoC. Specifically, the example compares different mapping approaches for a *Visformer* architecture [44] for image classification -from the Vision-Transformers category [57]- onto an AGX Xavier MPSoC with a single GPU and two deep-learning accelerators (DLAs).

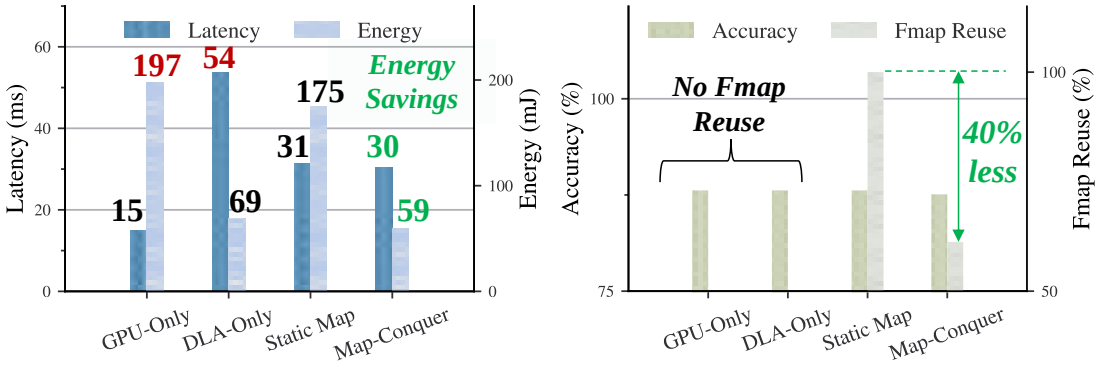


Figure 5.1 Performance comparison between different mapping and deployment options for *Visformer* [44] on Cifar100 and AGX Xavier MPSoC

As shown in the left subfigure, mapping the *Visformer* entirely to either hardware computing unit, namely *GPU-Only* and *DLA-Only*, yields a sub-optimal performance with regards to energy consumption for the former, and with regards to execution latency for the latter. As an alternative, we implemented a *distributed static mapping* strategy that aims to harvest the best of both worlds – GPU's speed and DLA's energy efficiency. More so, we implement the mapping strategy to exploit the underlying parallelism through *partitioning the Visformer* along its width dimension (i.e., the attention layer heads) and distributing them along the CUs. Mildly, the *static mapping* strategy leads to performance improvements over its single-mapping counterpart with regards to each component's deficient metric (42.6% speedup over *DLA-Only* and 11.1% energy gains over *GPU-only*). Accordingly, we alter our implementation to attain a *dynamic* version of this mapping, namely *Map-Conquer*, where the *Visformer* is deployed as a multi-exit neural network on the MPSoC, leading to substantial performance gains due to the nature of dynamic inference. In fact, this *dynamic* mapping strategy dominates the DLA with respect to both the latency (44.4% speedup) and energy efficiency (14.5% gain). Still, one deficit from such *distributed* mapping strategies is the additional inter-CU overheads experienced across the MPSoC. In the right sub-figure, we show that adopting a dynamic strategy can also alleviate such burden compared to the static mapping approach. Particularly, our approach identifies the key feature

subset from each stage and only involves those in any needed inter-CUs exchanges, denoted by *Fmap Reuse*. This scheme leads to 40% less *Fmap Reuse* compared to static mapping (which exchanges all needed features) at the expense of 0.5% accuracy drop.

5.4 Novel Scientific Contributions

In the realm of our observations and motivations summarized in Section 5.3, we introduce the following novel contributions:

- We present *Map-and-Conquer*, a novel energy-efficient execution scheme for Dynamic Neural Networks (NN) on heterogeneous MPSoCs that makes the best of both worlds of dynamic inference and distributed parallel computing paradigms.
- We leverage model-parallelism along the *width* dimension to partition the NN to multiple inference stages that can be run dynamically and concurrently on the MPSoC. The inference stages are meant to serve input data of increasing complexity, where simple samples are assigned to earlier stages.
- We derive a comprehensive system model to characterize the performance of concurrent inference stages on heterogeneous CUs with support for Dynamic Voltage and Clock Frequency Scaling (`DVFS`) for energy efficiency.
- We design an optimization framework based on evolutionary algorithms to search for the best width-based partitioning and workload mapping for Dynamic NN on the available heterogeneous CUs of the MPSoC.
- We conduct our experiments on CNN and Transformer based NN architectures for image classification on the CIFAR-100 dataset.
- We demonstrate the merit of *Map-and-Conquer* on the NVIDIA Jetson AGX Xavier MPSoC using various NN architectures. *Map-and-Conquer* can achieve up to $\sim$ **2.1x** more energy efficiency than the GPU-only mapping while incurring $\sim$ **1.7x** less latency than DLA-only mapping, all while preserving the desired level of accuracy.

5.5 Problem Statement

5.5.1 System Model for Mapping DyNN onto Heterogeneous MPSoCs

In this section, we comprehensively model the components needed to conduct a *static-to-dynamic* transformation of a given pretrained NN model and characterize its performance overheads when executing on the heterogeneous MPSoC accordingly.

5.5.2 Dynamic Transformation of Neural Networks

Consider an unaltered pretrained basic neural network, $\mathcal{NN}$, constituting a sequence of n computing layers (i.e., workloads) as follows:

$$\mathcal{NN} = \mathcal{L}^n \circ \mathcal{L}^{n-1} \circ \mathcal{L}^{n-2} \circ ... \circ \mathcal{L}^1 \tag{5.1}$$

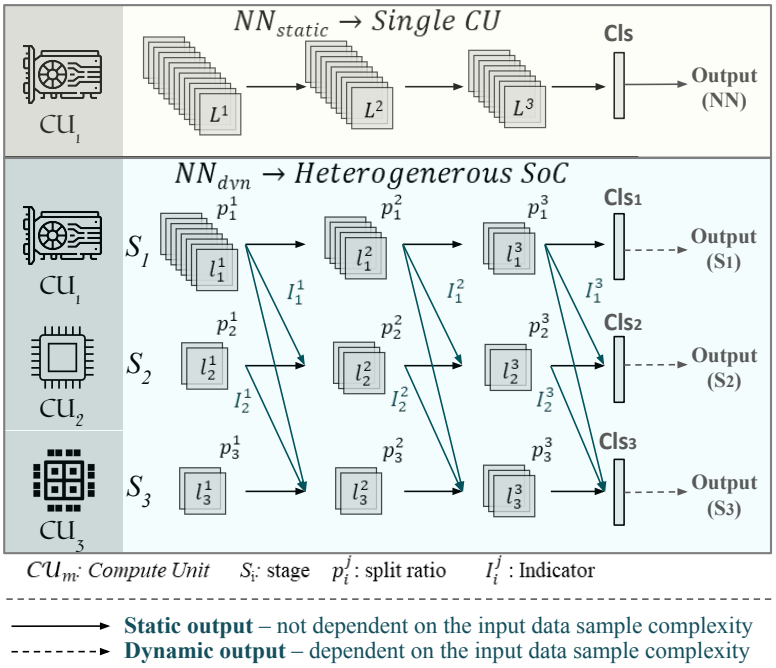


Figure 5.2 Transformation of NN_{static} into NN_{dyn} based on s and I, and mapping NN_{dyn} onto a MPSoC with multiple $\mathcal{CU}$s

in which each computing layer, $\mathcal{L}^j$, consists of pretrained weight parameter matrices (e.g., filters) whose count W_j represents the 'width' of the layer. Without losing generality, we refer to these weight matrices here as 'channels' such as those in a convolutional NN, whereas the same terminology can be applied to the 'heads' in a Transformer NN. Therefore, we can define the j^{th} layer within the NN as:

$$\mathcal{L}^j = \{C_1^j, C_2^j, C_3^j, ..., C_{W_j}^j\} \tag{5.2}$$

where C_i^j represents the i^{th} channel in the j^{th} layer. The output result of a computing layer is a feature map of W_j channels denoted as F^j.

Now, consider an MPSoC that comprises M CUs $\mathbb{CU} = \{\mathcal{CU}_1, \mathcal{CU}_2, ..., \mathcal{CU}_M\}$, the goal is to derive a strategy to partition every L^j into M subsets according to its width dimension (i.e., the channels), and thus, $\mathcal{L}^j$ and F^j are redefined as:

$$\mathcal{L}^j = \{l_1^j, l_2^j, l_3^j, ..., l_M^j\} \ \ and \ \ F^j = \{F_1^j, F_2^j, F_3^j, ..., F_M^j\} \ \ s.t. \ \ W_j \geq M \tag{5.3}$$

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 three operations to characterize this mapping problem: ❶ ***Partitioning*** ($\mathbb{P}$); to divide layers and generate the subsets l_m^j, and ❷ ***Concatenation*** ($\mathbb{I}$); 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:

$$\mathbb{P} = \begin{bmatrix} p_1^1 & \cdots & p_1^n \\ \vdots & \ddots & \vdots \\ p_M^1 & \cdots & p_M^n \end{bmatrix}, \ \mathbb{I} = \begin{bmatrix} I_1^1 & \cdots & I_1^n \\ \vdots & \ddots & \vdots \\ I_M^1 & \cdots & I_M^n \end{bmatrix} \tag{5.4}$$

where $\mathbb{P}$ is the *partitioning matrix* in which every p_i^j represents the fraction of channels in a layer L^j (equation (5.2)) 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 5.2 provides an illustration of how these matrices govern the partitioning and concatenation operations of a neural network. As shown, each $\mathcal{CU}_m$ on the MPSoC can host a unique sequence of channel subsets, which we denote as a stage, S_i, given as:

$$S_i = l_i^n \circ l_i^{n-1} \circ ... \circ l_i^1 \tag{5.5}$$

thus, ultimately, we obtain the following set of stages:

$$\mathbb{S} = \{S_1, S_2, ..., S_M\} \tag{5.6}$$

Suppose we augment each stage S_i with an exit at its tail (e.g., a classifier block) ; 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). In our analysis, we focus on delivering the optimal dynamic transformation of the NN under the assumption of optimal mapping of input samples to the corresponding inference stages. With that being said, each input sample is assigned to the earlier inference stage that correctly predicts its output class label.
Lastly, we define the ❸ ***Mapping Vector*** ($\mathbb{M}$) to parameterize the computing workloads assignment of the obtained inference stages onto the MPSoC:

$$S_i \rightarrow \mathcal{CU}_m \; \forall \; S_i \in \mathbb{S}, \mathcal{CU}_m \in \mathbb{CU} \tag{5.7}$$

where the mapping vector, $\mathbb{M}$, can by given as:

$$\mathbb{M} = [\pi_1, \ldots, \pi_M] \;\; s.t. \; \pi_k \neq \pi_{k'} \; \forall \; 1 \leq k \leq k' \leq M \tag{5.8}$$

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 $\mathcal{CU}_m$.

5.5.3 Distributed Performance Modeling for Dynamic Inference

Here, we model the dynamic inference execution overheads given the partitioned deployment of a model on a heterogeneous MPSoC regarding *execution latency* and *energy consumption*. We recall that we assume *ideal input mapping* in which the number of stages needed to process an input sample i is known a priori. Accordingly, we compute the overall accuracy, latency, and energy per input data sample, where each input data sample needs at least one inference stage to be correctly classified. For input data samples that fail to be correctly classified by every inference stage, we assume they must pass through all the inference stages. In practice, input mappings can be determined using runtime controllers as stated in [246, 21]. Thus, we compute both latency and energy in a *dynamic* manner for each input sample from the validation set of the target task (e.g., image classification).

① **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 CUs.

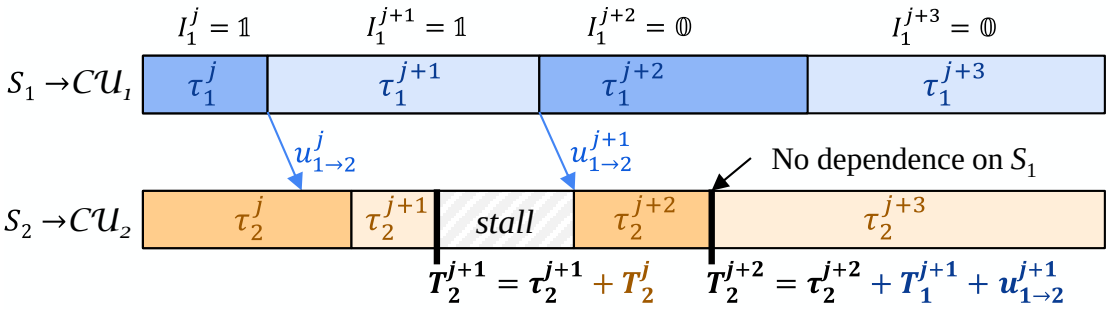


Figure 5.3 Concurrent execution of S_2 and S_1 considering timing dependencies

To avoid the demerits of a sequential execution model, we leverage the underlying separation of the compute units and propose a *concurrent* execution scheme 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 characterize the *cumulative* latency overhead at l_i^j as follows:

$$T_i^j = \tau_i^j + \max\{T_i^{j-1}, T_k^{j-1} + u_{k\rightarrow i}^{j-1} \mid I_k = \mathbb{1} \; \forall \; 1 \leq k < i\} \tag{5.9}$$

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 and the need for synchronization, while $u_{k\rightarrow i}^{j-1}$ is the data transmission overhead of the features F_k^{j-1} to the local buffer of the computing unit assigned to S_i (See Figure 5.3 for an illustrative example). Given n layers in S_i, the execution latency of S_i is equal to those of the last layer (i.e., nth) of the ith inference stage (i.e., S_i) as following:

$$T_{S_i} = T_i^n \tag{5.10}$$

② **Energy Consumption.** By definition, the energy consumption is the average power consumption per unit of time. Thus, for every $\mathcal{CU}_m \in \mathbb{CU}$, we first characterize its power consumption as follows:

$$P_m = P_m^s + P_m^d(\vartheta_m) \approx \alpha + \beta \cdot \vartheta_m \tag{5.11}$$

P_m^s and P_m^d are the static and dynamic components, respectively. The latter is parameterized by the scaling of the ❹ ***DVFS policy*** (ϑ_m) through varying the operating clock frequency 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:

$$e_i^j = \tau_i^j \cdot P_m \tag{5.12}$$

Now by extension, the total energy consumed by S_i is:

$$E_{S_i} = \sum_{j=1}^{n} e_i^j \tag{5.13}$$

③ **Overall Characterization.** Under the underlying concurrent execution scheme, the overall performance characterization for the end-to-end latency and energy of the dynamic $\mathcal{NN}$ are given by the following equations, (5.14) and (5.15), respectively:

$$T_{\mathbb{P},\mathbb{I},\mathbb{M},\vartheta} = \max\{T_{S_i} \; \forall \; S_i \in \mathbb{S}\} \tag{5.14}$$

$$E_{\mathbb{P},\mathbb{I},\mathbb{M},\vartheta} = \sum_{i=1}^{M'} E_{S_i} \; s.t. \; 1 \leq i \leq M' \leq M \tag{5.15}$$

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

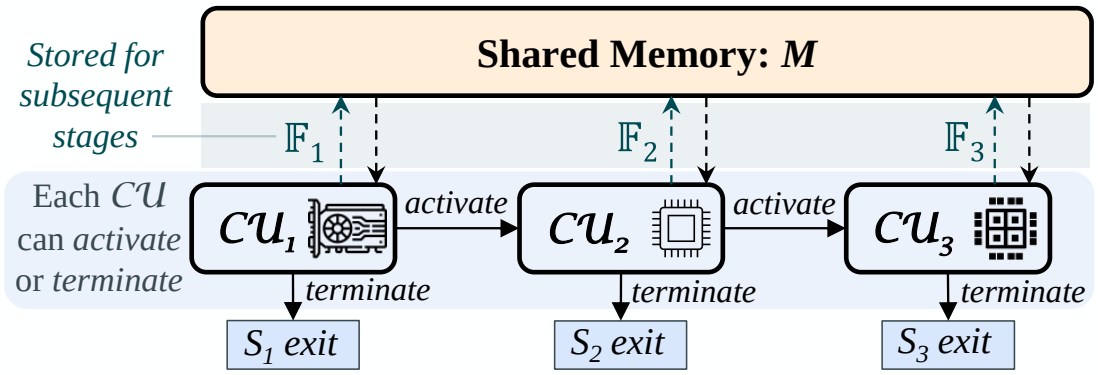


Figure 5.4 Illustration of data movement and feature storage on the MPSoC

5.5.4 Problem Formulation

Let $\Pi = (\mathbb{P}, \mathbb{I}, \mathbb{M}, \vartheta)$ combine all parameters that characterize a dynamic neural network's mapping onto an MPSoC. In this grand scheme, $\mathbb{P}and\mathbb{I}$ are hardware-agnostic and only depend on the $\mathcal{NN}$ architecture, whereas $\mathbb{M}$, ϑ are hardware-specific. However, the interplay of the chosen parameters of Π directly impacts the overall performances (i.e., accuracy, latency, and energy). Owing to this observation, we formulated the problem as a multi-objective joint optimization in which we investigated network- and hardware-related design parameters to three objectives at once. In a nutshell, this joint optimization aims to find the ideal set of parameters, Π^*, that can enhance a performance objective, $\mathcal{P}$, given a set of constraints:

$$\Pi^* = \arg\min_{\Pi} \mathcal{P}(\Pi) \tag{5.16}$$

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

Where T^{TRG} and E^{TRG} are the respective target latency and energy constraints to be set by the practitioner according to the system and application requirements. These parameters generally serve as control knobs to balance latency and energy and keep them within specific ranges. The constraint $\mathbb{size}_{\Pi}(\mathbb{F}, \mathbb{I}) < M$ is set to limit the size of the intermediate feature maps (denoted as $\mathbb{F}$) that need to shared and reused between parallel stages. Their overall size must be less than the MPSoC's shared memory size, M (see Figure 5.4), so as to minimize communication and synchronization overheads. We note that minimizing the size of feature maps that must be communicated across inference stages is essential to reduce the synchronization overhead for subsequent stages. $\mathcal{P}$ is kept generic and can be tuned to the designers' objectives.

5.6 Proposed Approach

In this section, we propose an evolutionary-based optimization framework to solve the partitioning and mapping problem of Dynamic NN on MPSoCs. Figure 5.5 gives an overview of our framework, whose key components are detailed below.

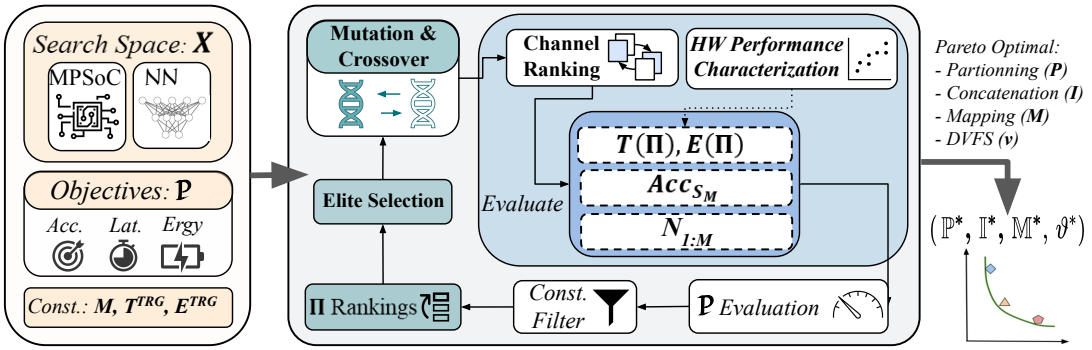


Figure 5.5 Overview of the workflow of our proposed optimization framework

5.6.1 Search Space (X)

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 a heterogeneous MPSoC with M CUs, we can generate X based on the $\mathcal{NN}$'s layer specifications and the MPSoC's underlying hardware components. 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. We assume that each inference stage will be assigned to one CU but different inference stages can not share the same CU to avoid memory contention overhead [237]. Lastly, ϑ is specified through the hardware reconfiguration parameters (`DVFS`). For instance, the mapping search space complexity of one layer from the *Visformer* [44] $\mathcal{NN}$ model is $\mathcal{O}(1.5 \times 10^5) = \mathcal{O}(8^3 \times 3! \times 50)$, considering 8 channel partitioning ratios, $M = 3$, and $|\vartheta| = 50$.

5.6.2 Performance Objectives ($\mathcal{P}$)

Performance objectives need to be designated as $\mathcal{P}$ for the main optimization function in equation (5.16), to be specifically used for the candidate mapping evaluation. For our case, we used the following weighted product function for $\mathcal{P}$:

$$\mathcal{P} = (\frac{Acc_{base}}{Acc_{S_M}}) \times (\sum_{i=1}^{M} T_{S_i} \cdot N_i) \times (\sum_{i=1}^{M} E_{S_{1:i}} \cdot N_i) \tag{5.17}$$

In which Acc_{base} is the baseline static accuracy of the pretrained NN model ; Acc_{S_M} is the accuracy of the last stage of the dynamic version of NN as its base accuracy. The terms above are included to ensure that no accuracy drops ensue when a model's structure changes through the $\mathbb{I}$ matrix from cutting-off paths of feature maps reused from previous inference stages. N_i represents 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 (5.10) ; $E_{S_{1:i}}$ is the energy consumed by the system as the result of executing i stages of the model – each E_i is evaluated as in (5.13).

5.6.3 Evolutionary Search Algorithm

We employ an evolutionary-based search algorithm to effectively explore the design space defined by X. Following the workflow in Figure 5.5, every new search iteration entails a new population, say $X'_i \subset X$. Then, for every sampled configuration $\Pi \in X'$, its corresponding dynamic $\mathcal{NN}$ -issued from the *static-to-dynamic*-transformation and hardware settings -from mapping vector and `DVFS`- are evaluated using a predefined objective function, $\mathcal{P}$. Based on the obtained results, configurations that do not meet the search constraints (e.g., memory usage via feature maps reuse ratio) are omitted, whereas the remaining ones are ranked according to $\mathcal{P}$. Afterward, the $1st$ quartile (25%) of elite configurations $\Pi' \in X''$ is selected for a mutation and crossover step to generate the new population X'_{i+1} for the next generation of the evolution. Once the search budget expires (i.e., the maximum number of generations), a Pareto set is calculated from all the generated populations from which the ideal configurations $\Pi*$ of dynamic $\mathcal{NN}$ and workloads mappings are extracted.

5.6.4 Channel Partitioning, Reordering, and Arrangement

Before a candidate configuration $\Pi \in X'$ is evaluated on the objective function P, the $\mathcal{NN}$ should be partitioned according to the channel parameters ratios in $\mathbb{P}$. Nevertheless, the width channels in each model layer are arranged according to their *importance scores* to maximize performances when partitioning. The logic is 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 earlier inference stages for dynamic inference so that to maximize the value of N_i. This would enable numerous samples to terminate processing prematurely if deemed feasible, consequently enhancing the *dynamic inference* performance of the $\mathcal{NN}$ regarding 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 for network pruning, and we follow the approach in [151] to estimate each channel's importance.

5.7 Experiments and Evaluation

In this section, we conduct extensive experiments to showcase the efficiency of our proposed framework in realizing effective width-based dynamic NN along with mapping computing workloads on the MPSoC. We study the case of two distinct neural network architectures, ❶ *Visformer*, a Transformer-based architecture, and ❷ *VGG*, a CNN-based architecture. Both architectures are designed for vision-related tasks (e.g., image classification). We use a real MPSoC platform from NVIDIA, namely Jetson AGX Xavier [162] to assess the merit of *Map-and-Conquer* in finding optimal dynamic NN workload distribution given the heterogeneity between GPU and DLA.

5.7.1 Experimental Setup

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 memory. To run the NN workloads on the DLA, we use TensorRT and ONNX to build inference engines from the PyTorch model. As NNs, we use *Visformer* [44] as Transformer-based architecture and *VGG19* [199] 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 overall optimization process.

5.7.2 Search Efficiency 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 5.6, 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*). Hence, defining the optimal inter-stages concatenation strategy that determines the feature maps reuse ratio is crucial to maintaining the desired level of accuracy while minimizing inter-CUs dependencies.

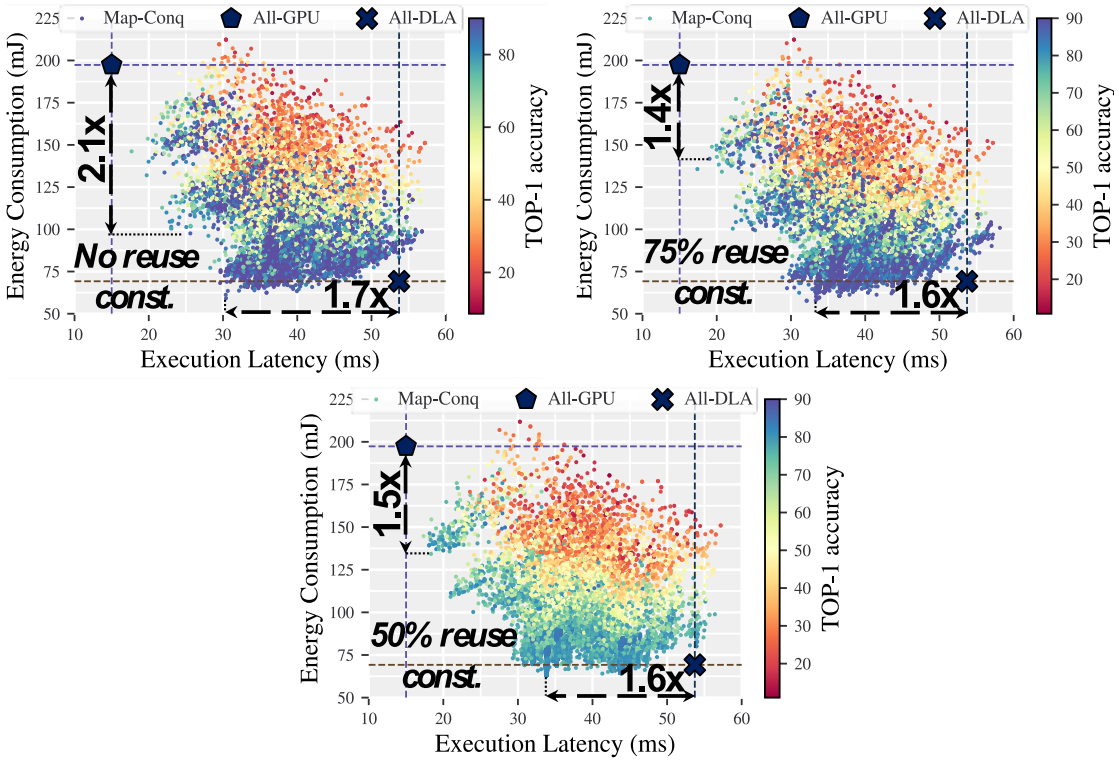


Figure 5.6 Results of three search strategies (**From left to right**): *No constraint* is set on the *Fmap Reuse*. *Under a constraint of reusing only less than 75%* of feature maps. *Under a constraint of reusing only less than 50%* of feature maps. All the results are reported for *Visformer* on the 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

5.7.3 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 5.7 and Table 5.2 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 $\sim$ **1.83x** and up to $\sim$ **14.4%** of energy gain as shown in the left sub-figure. In addition, the correlation between feature map 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). We note that for our case, we assume the positive correlation between the ratio of feature maps reuse between inference stages and memory contention overheads. According to the findings in [237] and by employing their analytical model or our case, we observed in our preliminary that the latency slowdown due to memory contention overhead in '*No reuse const.*' case was around 20-30%. This is explained by the fact that inference stages from the same NN generally exhibit low memory bandwidth

Table 5.2 Performances Breakdown of the Pareto optimal models obtained by *Map-and-Conquer* and the baselines

Opt. Strategy	NN Implment.	TOP-1 Acc (%)	Avg. Enrg. (mJ)	Avg. Lat. (ms)	Fmap. reuse. (%)
Visformer (Transformer-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**

demands compared to running concurrent NNs with high memory demands.

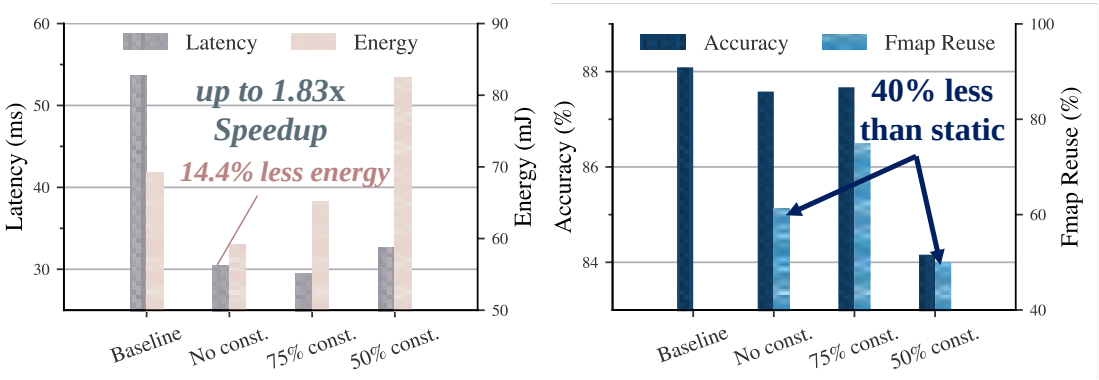


Figure 5.7 Comparison between the most energy-oriented models selected from the obtained Pareto sets by each search strategy and the baseline on DLA

5.7.4 Generalization to Other Neural Architectures

To further demonstrate our approach's applicability, we evaluate our framework on a typical CNN architecture, *VGG19*. Table 5.2 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 high redundancy. Our approach has exploited these two properties of *VGG19* well, resulting in up to $\sim$ **4.62x** energy gain and $\sim$ **4.44x** latency speedup. Furthermore, according to our analysis, more than 80% of the input data samples were correctly classified in earlier stages with fewer channels, resulting in considerable latency and energy gains.

5.8 Discussion and Key Insights

① **Main Takeaway.** Hardware-specific co-optimization of Neural Networks has been thoroughly studied in literature [221, 254, 100, 69]. Recently, there has been an increasing interest in dynamic pruning techniques as first introduced by [139]. Following its success, recent works have focused on enhancing dynamic pruning from a hardware perspective [69, 65]. However, most of these works are built upon highly specialized hardware settings. What distinguishes *Map-and-Conquer* from related works is its general optimization landscape that allows it to operate on any MPSoC –general-purpose or specialized ones. Notably, abstracting low-level hardware details in the mapping problem allows for further extrapolation on FPGA, ASIC, or brain-inspired hardware devices. *Map-and-Conquer* has demonstrated promising results that can serve as a stepping stone for future research endeavors.

② **On the Power of Dynamic Neural Networks.** Recently, more efforts have been dedicated to developing new techniques to maximize dynamicity and flexibility within neural networks [21, 69]. The particularity of *Map-and-Conquer* is its novel method of early-exiting along the width dimension that relies on splitting and reorganizing the channels to of an already pretrained NN model to spawn submodels (i.e., inference stages) capable of operating on less complex input data. Our approach can be generalized to any neural network architecture by operating on the parameters that scale the width of the networks (e.g., the number of attention heads in Transformers or Graph size in GNNs). Our dynamic approach can be enhanced by incorporating more dynamic techniques such as depth-wise early exit and layer skipping. However, one challenge arises from the need to unify the structure of the dynamic inference stages to enable the particularity of feature maps to preserve the original accuracy without the need to train or fine-tune the dynamic Neural Net.

③ **The Importance of Exploiting Heterogeneity within MPSoCs.** Heterogeneous MPSoCs are emerging as an attractive solution to provide more flexibility in deploying computing workloads of different characteristics on diverse computing units within the same die. The particularity of memory sharing has made it possible to manage data communication between CUs with low to negligible overhead, offering favorable prospects to exploit the collaborative execution paradigm

on these MPSoCs. *Map-and-Conquer* seize this opportunity to the fullest by incorporating the dynamicity and mapping of inference stages, enabling the dynamic usage of CUs. In our approach, MPSoC hardware resources are used dynamically according to the complexity of the input data. As such, simple input data requiring one inference stage will be processed by either the fastest CU (i.e., GPU) or the energy-efficient CU (i.e., DLA). As the input data complexity increases, the processing will require the collaborative usage of more CUs (combination of GPU-DLA or DLA-DLA) to balance latency and energy efficiency. Following this novel scheme of *Map-and-Conquer*, both the workloads of the dynamic neural network and Hardware resources are used dynamically, providing more scalability and flexibility for runtime applications, which also opens doors for further development of an end-to-end self-adaptive system, especially for critical application domains (e.g., autonomous cars and unmanned aerial vehicle).

5.9 Summary

In this chapter, we have explored the prospect of dynamically mapping intricate neural networks onto various CUs of heterogeneous MPSoCs. Specifically, we combine the concept of early-exit NN along the width dimension and computation parallelism to make the best of both worlds. Consequently, we introduce *Map-and-Conquer*, a novel framework crafted to align dynamic neural networks (NNs) with heterogeneous MPSoCs, thereby maximizing processing parallelism. Our approach focuses on identifying the optimal way to divide a dynamic NN across its 'width' dimension, enabling the parallel and concurrent execution of NN inference stages on different computing units. Additionally, we propose a unique method to allocate these divided NNs onto the MPSoC, utilizing dynamic multi-exit networks to enhance performance. We validate our framework using Transformers and CNNs on the CIFAR-100 dataset and employ NVIDIA's Jetson AGX Xavier MPSoC, which includes a GPU and Deep Learning Accelerator (DLA), for our hardware experiments. Our findings have shown that our dynamic configurations are 2.1x more energy-conserving than GPU-only setups and experience 1.7x reduced latency than DLA-only configurations. Future works can extend our *Map-and-Conquer* framework from the following perspectives:

— Applicability of our approach to other applications, tasks, and datasets (e.g., anomaly detection, scene parsing, and extraction).

— Leveraging other dynamic inference techniques that provide an edge for computation parallelism. For instance, dynamic routing and mixture-of-experts.

— Innovating a new problem-solving approach that can minimize the search time. For instance, dynamic programming and reinforcement learning.

In addition, we believe that the novel computing paradigm of *Map-and-Conquer* can also be beneficial for new emerging hardware architectures such as heterogeneous integration in Chiplets and heterogeneous multi-core RISC-V systems.

Chapitre 6

Performances Modeling of Computer Vision-based Convolutional Neural Networks on Edge GPUs

6.1 Introduction

Machine Learning (ML) algorithms have demonstrated their efficacy across several fields of application, including Computer Vision (CV) and Natural Language Processing (NLP). The pairing of reliable hardware (HW) platforms with large and diverse datasets enables machine learning algorithms to tackle intricate challenges across various applications, including autonomous driving and healthcare. Deep Learning (DL) techniques, such as Convolutional Neural Networks (CNN), are often regarded as cutting-edge methods in CV applications. For instance, CNNs are extensively employed for scene comprehension and perception in autonomous driving cars [97]. Processing data near sensors makes edge computing mandatory for privacy and network congestion concerns. CNNs can also leverage the potential benefits offered by edge computing through the use of ASIC, FPGA, GPU, and MCU. Particularly, edge GPUs exploit massive data parallelism that characterizes CNN models at the cost of high power consumption. However, these devices offer fast and flexible product development and short time-to-market compared to other HW architectures [123, 101].

As CNNs continue to advance in accuracy and complexity, their computational and memory demands set significant challenges for edge GPUs. Investigating different CNN design alternatives to pursue hardware efficiency is time-consuming. Conventional methods propose quantifying CNN's complexity by using measurements of FLOPs and weights [177]. However, CNN's performances are not strongly correlated with proxy estimations of FLOPs or weights [15]. In order to save design and deployment time, a quick performance estimation strategy becomes essential. This strategy can assist in selecting the optimal CNN for a specific HW platform or comparing other HW platforms for a target CNN. Additionally, it may serve as a surrogate-based model for directing the design optimization process toward the most favorable CNN/HW designs.

In this chapter, we address one of NAS's primary challenges: The *evaluation* step, which typically involves extensive performance measurements. To accelerate this process, we propose an end-to-end characterization and modeling approach to elaborate prediction models for rapid performance estimation in NAS. Our work includes a comprehensive analysis of CNN workloads for image classification on edge GPU devices. We study and explore the correlation between NN features and critical metrics such as execution time, power consumption, and memory usage. Accordingly, we employ ML methods like linear regression, support vector ma-

chine, random forest, and XGBoost to model these correlations and develop prediction models. We validate our methodology on state-of-the-art and synthetic CNN architectures using three NVIDIA Jetson series edge GPUs: AGX Xavier, TX2, and Nano. Our elaborated prediction models achieved an average error rate of approximately 11%, 6%, and 8% for execution time, power, and memory usage estimations, respectively. Contributions and results of this chapter have been published in:

— [23] **Halima Bouzidi**, Hamza Ouarnoughi, Smail Niar, and Abdessamad Ait El Cadi. Performance prediction for convolutional neural networks on edge GPUs. The ACM International Conference on Computing Frontiers (2021)

— [24] **Halima Bouzidi**, Hamza Ouarnoughi, Smail Niar, and Abdessamad Ait El Cadi. 2022. Performance Modeling of Computer Vision-based CNN on Edge GPUs. ACM Trans. Embed. Comput. Syst. 21, 5, Article 64 (2022)

Our proposed framework is currently in use and has been commercialized by the Research Institute Technologique SystemX, based in Toulouse, France. This institute has begun employing and reusing our modeling methodology within their automotive systems to assess the execution overheads of CNNs on edge GPUs [60].

6.2 Related Works

6.2.1 Benchmarking and Performances Analysis

These strategies are based on designing CNNs specifically tailored for the edge [230, 73, 214, 229]. However, these techniques aim to manually reduce the overall computational load and memory footprint. Thus, the performances of the resulting CNNs may vary depending on the hardware platform, as they are not designed for specific hardware.

6.2.2 Execution Time Modeling

In the literature, prediction models are used either to estimate CNN training and/or inference execution time. In [7], the authors characterize inference execution time on GPU platforms using different ML-based approaches: linear regression, SVR, and RF with a Bulk Synchronous Parallel (BSP) based analytical model. They exploit profiling results obtained from nine (9) benchmarks executed on nine (9) different GPUs. However, the proposed approach considers general-purpose applications that cannot be adapted to DL applications.

In [223], the authors propose analytical models to characterize DL training workloads in large computing clusters. Authors in [196] propose a set of prediction models for training time with Stochastic Gradient Descent (SGD) optimization considering the communication time between GPU nodes, I/O processing time, and GPU processing time. However, the authors only evaluated their prediction models on three CNNs: AlexNet, GoogleNet, and ResNet50. In PALEO [177], the number of Floating-point OPerations (FLOPs) required for an epoch is multiplied by a scaling factor to obtain the training phase's execution time. However, PALEO does not consider numerous other operations that do not scale linearly with the number of FLOPs and significantly impact execution time.

Authors in [102, 28, 204, 218] leverage ML approaches to model CNN inference execution time on a layer-wise granularity. However, layer-wise modeling approaches are not adapted to complex CNN architectures with dependencies between layers. For instance, ResNet [84, 85] and DenseNet [94] are characterized by skip and dense connections between layers. Another example is the parallelism in CNNs such as GoogleNet [209] where layers within the inception block can be evaluated simultaneously on the GPU [177]. These dependencies can, therefore, lead to higher prediction errors when estimating execution time.

6.2.3 Power Consumption Modeling

In [183], the authors propose a multi-variable linear regression to predict the energy consumption of the inference phase for CNNs based on the number of SIMD instructions and DRAM memory accesses, as they are both high-energy consumers. The authors used Nvidia Jetson TX1 GPU and obtained about 20% of an average relative error. However, modeling power consumption using only the features mentioned above is insufficient as it neglects further optimization of the neural network computational graph and dataflow.

The idea behind tools proposed in [28, 204, 218] is to explore the hyperparameter space of the most common layers of CNNs, such as convolution, pooling, and fully-connected layers. The power consumption is then estimated layerwise, and an average is taken as the end-to-end power consumption of the CNN inference. However, it is difficult to capture the exact power consumption of each layer as its execution time is very short, leading to high prediction errors of the overall model power consumption. Authors in [145, 206, 148] use analytical models to estimate the average power consumption on FPGA or ASIC platforms. Nevertheless, analytical models lack flexibility as they require calculating low-level details of the CNN execution, such as memory transactions and processing elements utilization, which is not disclosed information for every HW device.

6.2.4 Memory Usage Modeling

HyperPower [204] proposes to model memory usage from structural hyperparameters of the CNN, such as the number of hidden units. The authors employ linear regression to model memory usage. However, they have trained their prediction models on only variants of AlexNet [114]. In [143], the authors propose a modeling methodology for CNN memory usage on CPU and GPU platforms. Their approach is based on characterizing the memory requirements of convolutional and fully connected layers to predict the entire CNN's memory usage. Authors in [8, 106] propose prediction models of cache memory hierarchies for DNNs on modern discrete GPU platforms to investigate further optimization of memory performance on GPUs. Some works also leverage performance estimators to reduce CNN memory footprint during training [32] or inference [136]. However, they did not consider other performance metrics, such as execution time and power consumption.

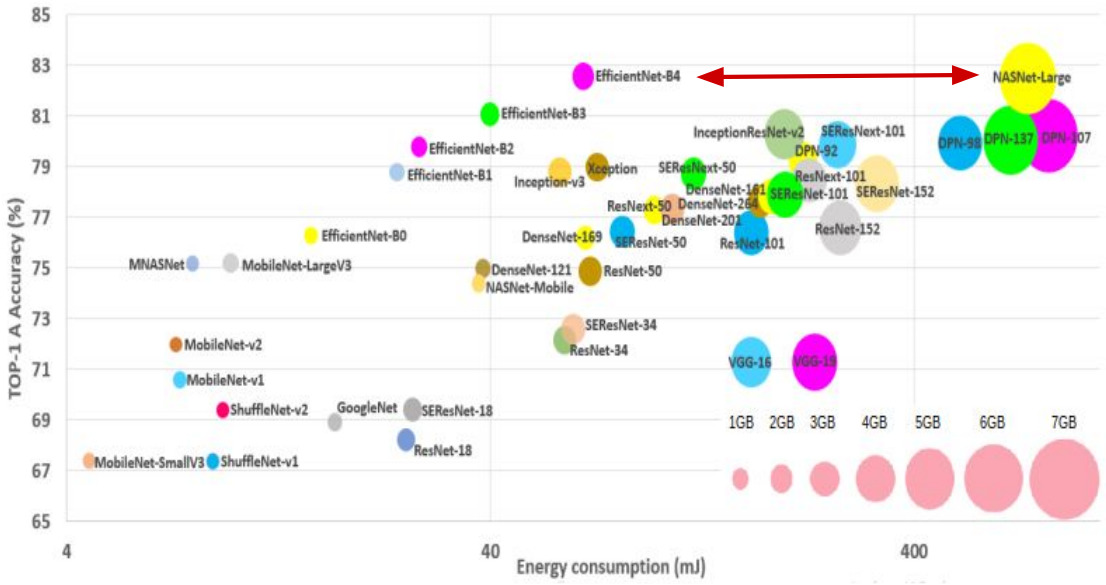


Figure 6.1 Correlation between CNN accuracy, on ImageNet dataset, and their energy consumption and memory usage. The size of the circle corresponds to the total memory usage from 1 Gigabyte to 7 Gigabytes.

6.3 Motivational Example

Figure 6.1 depicts the tradeoff landscape observed in CNN architectures for image classification. Each circular shape depicted in the diagram represents a SOTA CNN model. The size of the circles correspond to the maximum amount of memory required for processing the inference. In this context, memory usage refers to the allocation of memory for storing weights and intermediate calculations [143]. The x-axis indicates the energy consumption, which is determined by the product of the inference time and the average power consumption measured on an edge GPU device, namely the NVIDIA Jetson AGX Xavier. The y-axis represents the TOP-1 accuracy on the ImageNet validation dataset [53]. The following observations are derived analyzing the reported values:

1. The TOP-1 accuracy is not correlated with either energy consumption or memory usage. EfficientNet-B4 [215] and NASNet-Large [271] both demonstrate a TOP-1 accuracy of around 83%, however, EfficientNet-B4 consumes ∼89% less energy and requires ∼53% less memory than NASNet-Large.
2. Memory usage is often correlated with energy consumption. NASNet-Large , DPN-98, DPN-107 and DPN-137 [41], for instance, have significant memory footprints and consume a large amount of energy.

These findings highlight how difficult it is to strike the ideal compromise between CNN's workload requirements and the hardware limitations of edge platforms. Furthermore, conducting a comprehensive investigation of CNN design choices on various hardware platforms can be arduous due to the time-consuming MLOPs process [266]. Therefore, a viable approach to address this concern is by abstracting the physical limitations and adopting a performance prediction approach for CNN without the need to execute them on the hardware device.

6.4 Problem Statement

Predicting CNNs performance metrics on edge GPUs can be formulated as follows: As inputs, we have a CNN (cnn_i) characterized by a vector of n features $(f_1, f_2, \ldots, f_n)$. For instance, the number of convolutional and fully-connected layers, the input image size and the number of neurons are all considered CNN features. A performance metric prediction function $\boldsymbol{T_{k,edgpu_j}}$ where $k \in \{execution\ time,\ memory\ usage,\ power\ consumption\}$ We note that in this chapter we use latency to interchangeably refer to execution time. $edgpu_j$ is an edge GPU, is a mapping function from cnn_i to $\mathbb{R}_+$. The function $\boldsymbol{T_{k,edgpu_j}}$ is defined below:

$$T_{k,edgpu_j} : cnn_i \rightarrow \hat{y_{ij}} \quad \text{where} \quad \hat{y_{ijk}} = T_{k,edgpu_j}(f_{i1}, f_{i2}, \ldots, f_{in}) \tag{6.1}$$

where $\hat{y_{ijk}}$ is the estimated value of the performance metric k of cnn_i on $edgpu_j$.

We study the case of three different NVIDIA edge GPUs, namely Jetson AGX Xavier, Jetson TX2 and Jetson Nano. From the problem formulation, the same modeling approach is applied on each edge GPU. Given that each metric is modelled by a specific mapping function, three sets of prediction models, for each performance metric k, are associated to each edge GPU.

Table 6.1 Summary of notations.

Name	Description
n	# of CNN input features
f_i	CNN input feature (e.g., FLOPs, number of layers, parameters..etc)
cnn_i	Vector of CNN input features
$edgpu_j$	Edge GPU indexed by j
k	Performance metric: execution time, power consumption, or memory usage
$\hat{y_{ijk}}$	Predicted value of the performance metric k of cnn_i on $edgpu_j$
y_{ijk}	Measured value of the performance metric k of cnn_i on $edgpu_j$
D_{jk}	Dataset of input features and measured values of k on $edgpu_j$

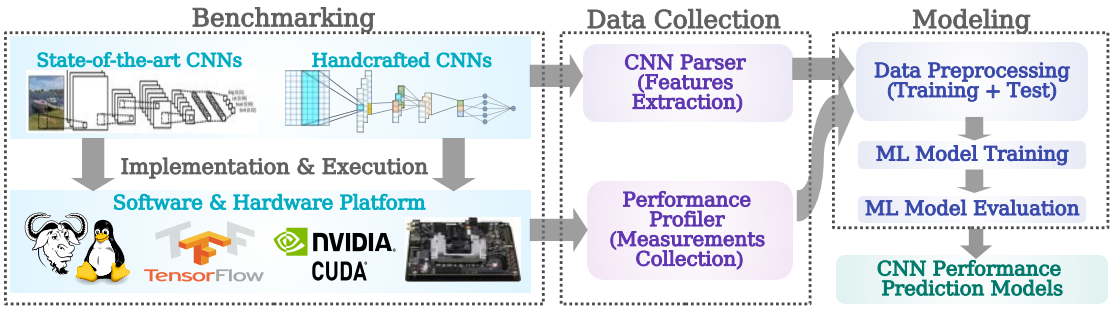


Figure 6.2 Modeling methodology for CNNs performances prediction models.

6.5 Proposed Approach

Our contribution entails presenting a modeling methodology to assess the performance of CNNs on edge GPUs. This methodology examines the correlation between key CNN characteristics (e.g., image input size, hidden layers, neurons, activations) and performance metrics (e.g., execution time, power consumption,

and memory usage). Our proposed methodology is depicted in Figure 6.2 and comprises three main steps:

1. **Benchmarking** where we measure the execution performances of several CNNs on various edge GPUs. The aim is to leverage correlation and regression analysis to quantify the impact of CNN model-level features on the performance metrics.
2. **Data Collection**: In this phase, we gather two distinct types of data from the underlying benchmarks. Firstly, we collect performance measures, which provide empirical data on the benchmarks' execution. Secondly, we collect characteristics of CNNs, which will be used as input variables for the purpose of prediction. The data were acquired through the use of a CNN architecture parser and performance profilers. The model parser takes a CNN model description as an input and generates a comprehensive representation of its architectural characteristics. The performances profiler takes a CNN architecture description, generates a ready-to-deploy CNN, deploys and executes the CNN on the target edge GPU, then returns the execution performance metrics values (execution time, power consumption, and memory usage).
3. **Modeling** elaborates a set of ML-based prediction models for each performance metric and edge GPU. This step also involves the data pre-processing to perform feature transformation (i.e., data encoding and transformation) and feature selection for the ML-based prediction models. The prediction models are trained on the collected data from the previous step. The training process encompasses the prediction model's hyperparameters tuning and internal parameters (i.e., weights and biases) learning.

6.5.1 CNN Characterization

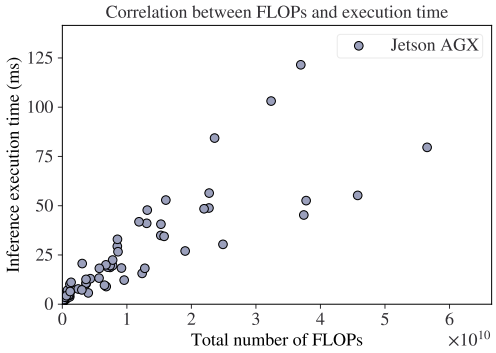
Our proposed methodology characterizes CNN execution performances on a model-level granularity. Regarding the CNN architecture and the target edge GPUs, we assume that the following factors impact the performances of the CNN:

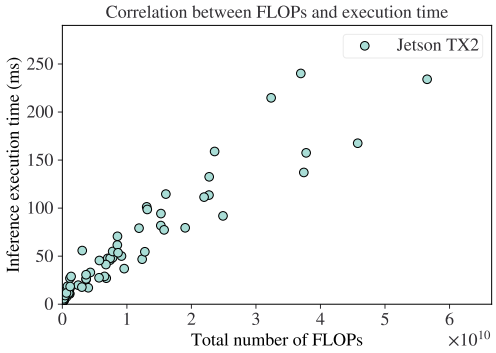
1. Computational complexity, which directly impacts the GPU activities.
2. Memory workload, which corresponds to read and write memory operations for input, intermediate and output activations and weights ;
3. CNN hyperparameters, which corresponds to the structural dependencies between computation and memory operations.

Considering the above factors, we search for the most correlated features with CNN performance. Since we target to model different performance metrics, the impact of the features may differ from one target performance metric to another. For instance, CNN features related to memory requirements are the most relevant to model CNN memory usage. We further discuss the CNN features related to each impact factor and the followed process to select the most relevant features for the prediction models.

6.5.1.1 Computational Complexity

The computational complexity is commonly quantified by the theoretical total number of FLOPs (Floating-point Operations). This characteristic depicts the number of calculations needed to execute a single inference. According to the





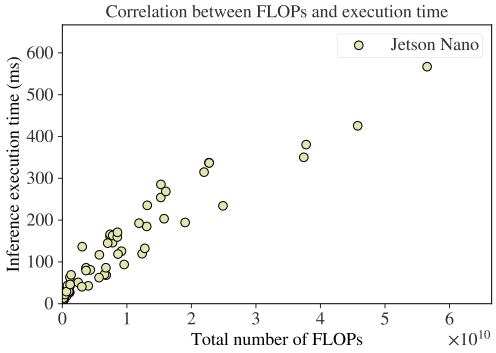


Table 6.2 FLOPs breakdown.

CNN	Conv2D	Add	Mul	Pool
ResNet-50	7.71B	31.02M	25.58M	1.81M
DenseNet-121	5.67B	7.89M	8.02M	1.98M
DPN-98	23.34B	70.54M	61.63M	2.71M
GoogleNet	3.00B	6.61M	6.64M	12.55M
ResNet-101V2	14.38B	52.32M	44.59M	2.16M
Inception-v3	5.67B	23.80M	23.85M	12.18M

Figure 6.3 Correlation between FLOPs and CNN execution time on three edge GPUs (*from left to right*): NVIDIA Jetson AGX Xavier, TX2, and Nano.

FLOPs breakdown analysis shown in Table 6.2, most FLOPs are attributed to the calculations carried out in convolutional layers, accounting for almost 98% of the overall computations. As anticipated, the computational barrier arises from these layers due to their involvement in several Multiply-accumulate (MAC) operations [124]. Nevertheless, the optimization strategies employed by GPU on the CNN computational graph hinder the accuracy of determining the end-to-end performances based only on the theoretical total number of FLOPs. To further demonstrate this, Figure 6.3 depicts the relation between the total number of FLOPs and the inference execution time on three (03) different edge GPUs. As shown, there is no strong correlation between execution time and FLOPs. For instance, two models may have the same FLOPs number but different end-to-end execution times. Furthermore, as depicted in Figure 6.4.a, there is no linear correlation between FLOPs and power consumption. Two CNN models may have the same number of FLOPs but consume different power budgets.

6.5.1.2 Memory Requirements

For General-Purpose GPU (GP-GPU) devices, memory activities significantly impact execution performances. Based on our experimental findings, it has been shown that the substantial memory demands of CNN inference may primarily be attributed to three key aspects.

1. Memory read of model's weights and biases,
2. Memory read/write of input data and output results,

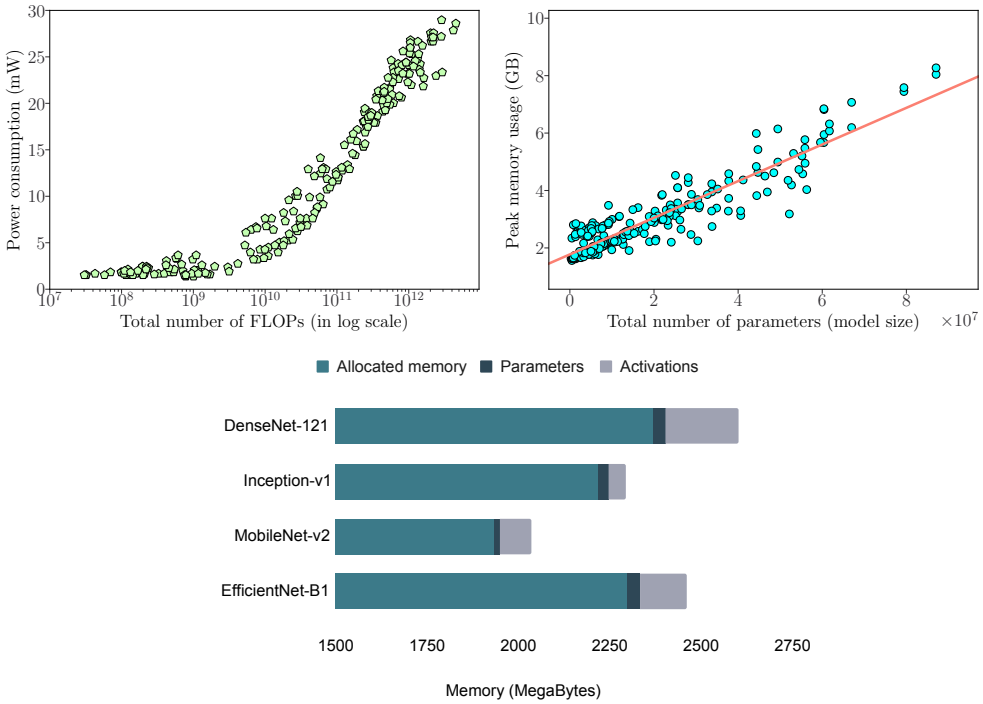


Figure 6.4 (a) Correlation between power consumption and the total number of FLOPs (in log scale). **(b)** Correlation between memory usage and the weights size. **(c)** Memory usage breakdown for a single CNN inference on an edge GPU.

3. Memory read/write of the intermediate activations.

To provide more context, Figure 6.4.c details the memory requirements for CNN parameters, activations, and the measured system memory allocated for the DL framework (TensorFlow in our case). As shown, the allocated memory for the DL framework is significant and varies from one CNN to another. As mentioned in [143], existing DL frameworks, such as TensorFlow, PyTorch, and Caffe, are not built with awareness of the unified memory of edge GPUs. Consequently, the data transfer between the CPU and GPU creates redundant data copies on system memory. Moreover, to speed up the computation, the memory allocated to the DL framework can only be released at the end of CNN inference, which requires a significant amount of memory for the DL framework. According to our experiments depicted in Figure 6.4.b, the memory usage of CNNs is strongly correlated with the sum of weights and activations. However, the high memory usage does not necessarily increase its execution time and power consumption. Indeed, data are not accessed with the same frequency during the inference. For instance, convolutional weights and activations are constantly accessed, whereas fully-connected weights are accessed once for each activation [202]. In addition, when activations and weights cannot be fully loaded in the GPU cache memory, execution time and power consumption may increase. In order to comprehensively evaluate the influence of memory access on execution time and power consumption, it is important to initially investigate the CNN dataflow through several levels of the memory hierarchy. Nevertheless, examining memory operations without profiling the CNN on the target edge GPU at runtime is intricate. Hence, this approach is

not viable due to its complexity and the substantial increase in prediction latency it would entail. In order to address this issue, we employ CNN characteristics that have a strong correlation with memory-related activities. Therefore, it is hypothesized that the properties of weights, input, output, and intermediate activations significantly influence memory operations.

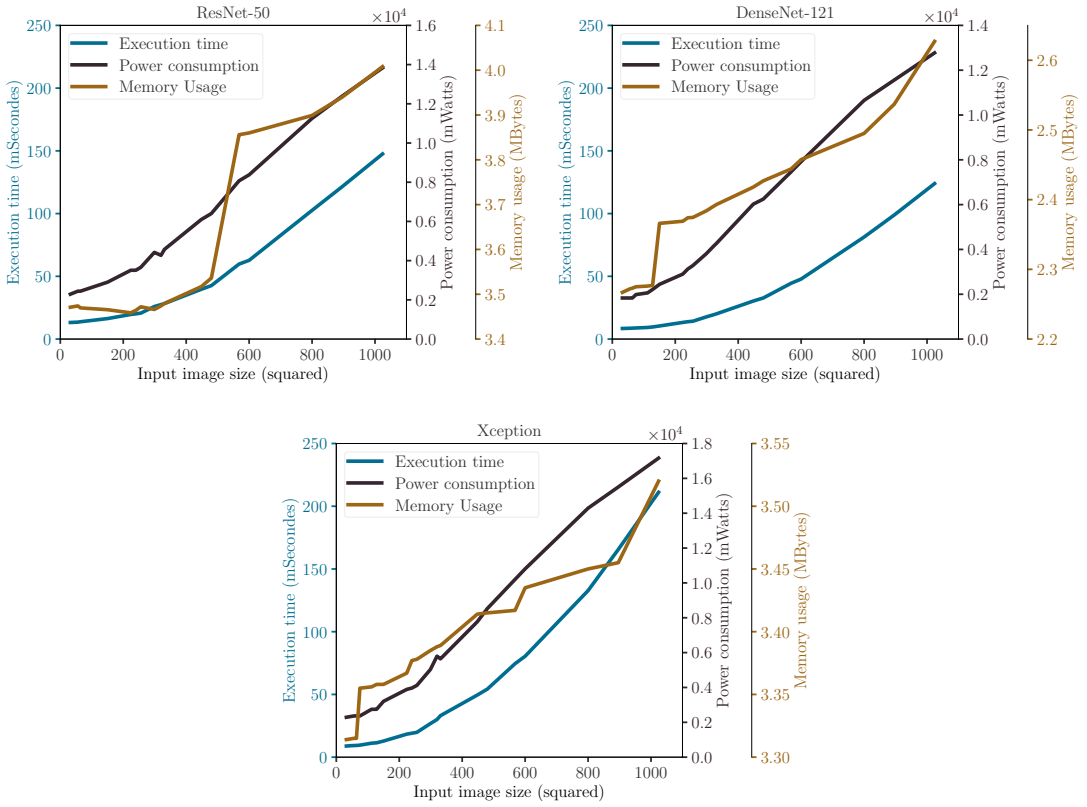


Figure 6.5 Correlation between image input size and the performance metrics: execution time, power consumption, memory usage on the NVIDIA AGX Xavier.

6.5.1.3 CNN Architectural Hyperparameters

In addition to the aforementioned characteristics, the architectural attributes of CNNs are also incorporated. We consider the number of convolutional, fully connected, batch normalization, and pooling layers. Furthermore, we examine the influence of different image input dimensions on CNNs. The procedure of up or down the input size can be employed in conjunction with fine-tuning to adjust the CNN model for novel training tasks and datasets. Firstly, as shown in Figure 6.5, execution time, power consumption, and memory usage strongly and positively correlate to the CNN input image size. Additionally, to characterize the neurons within the CNN, we introduce a novel feature called "the weighted sum of neurons." This feature is computed by aggregating the total number of neurons in convolutional and fully-connected layers. The number of neurons in convolutional layers is then multiplied by the filter size: $height \times width \times depth$, to give more importance to neurons with large filter sizes. Nevertheless, The number of neurons in fully-connected layers is not weighted because the input neuron is associated with a single scalar and not a multidimensional filter.

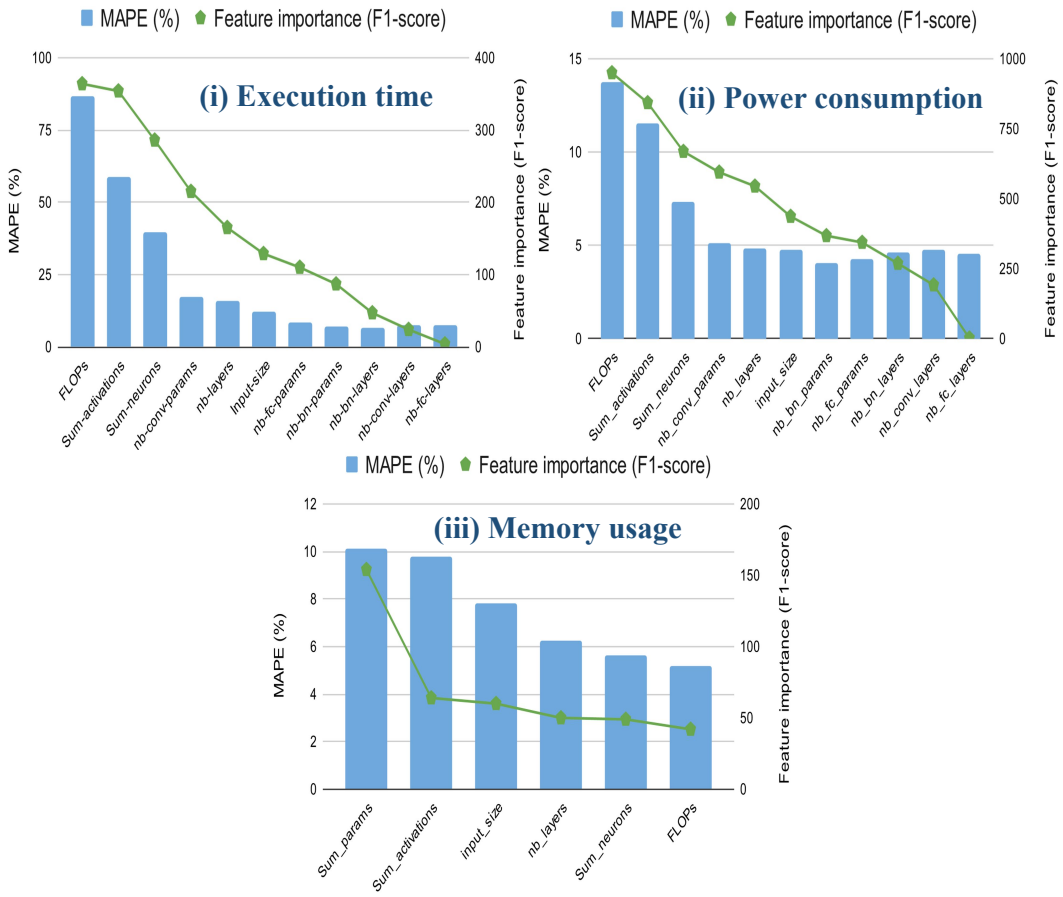


Figure 6.6 The forward stepwise selection of features for CNN performances modeling on Jetson AGX Xavier. High F-score correspond to high feature relevance.

6.5.2 Input Features Selection

A forward stepwise technique is employed to select the most relevant CNN features as prediction variables. For each prediction model, we begin with an empty set of features. The feature with the highest F-score is added at each step as a prediction variable. The F-score is calculated via the XGBoost [39] algorithm to rank the features from the most to the least important. Figure 6.6 reports the results of the stepwise feature selection based on the calculated feature importance with the corresponding improvement in MAPE for each performance metric on the Jetson AGX Xavier. Features are ranked and added to the prediction model based on their relevance (i.e., high F-score values). On the left y-axis we report the improvement of the Mean Absolute Percentage Error (MAPE). On the right y-axis we report the corresponding feature importance calculated by XGBoost. A similar trend has also been observed for the two other edge GPUs (Jetson TX2 and Nano). As shown, including more features in the prediction model continues until a point is reached when there is no discernible enhancement in the Mean Absolute Percentage Error (MAPE). The feature's importance depends on performance metrics. However, the same trend has been noticed for execution time and power consumption. The most crucial features for estimating memory usage are those that pertain to memory needs.

6.5.3 Prediction Algorithms

We leverage several ML-based methods for regression tasks such as Polynomial regression (Poly) [167], Support Vector Regression (SVR) [11], Multi-Layer Perceptrons (MLP) [156], Random Forest (RF) [130], and eXtreme Gradient Boosting (XGBoost) [39]. We also list and report Table 6.4 the hyperparameters choices used for tuning and training each of the aforementioned ML-based methods.

6.6 Evaluation Methodology

This section details our evaluation methodology and the obtained results. As shown in Figure 6.2, our modeling process is subdivided into three main steps: ① Benchmarking, ② Data collection, and ③ Modeling. We first detail each step, give details about the experimental setup, and finally present and discuss our obtained results.

6.6.1 CNN Benchmarking

The benchmarking phase primarily involves the deployment and execution of the CNNs inference on the targeted edge GPUs. The benchmarks have been designed based on state-of-the-art CNNs for image classification. The taxonomy of the CNN architectures investigated in this study is depicted in Figure 6.7. As seen, we have considered different architectures from the depth and width-based CNN, such as ResNet and Inception, to multi-path and multi-connection-based CNN, such as ShuffleNet and MobileNet. Furthermore, in order to enlarge the scope of our datasets to characterize various types of correlations, we employ the following data augmentation techniques:

1. Input Image Size: The impact of the input image size on the CNN performances is studied. For this purpose, we test the frequently used image resolutions, from 32*32 to 2400*2400 pixels, with three channels for the RGB representation.
2. CNN Variants: Different variants of the same CNN architecture are considered. For instance, we deploy different variants of the ResNet architecture [84, 85] by varying the number of layers and residual blocks. Hence, we obtained eight (8) widely used variants of ResNet, such as ResNet18, ResNet34, ResNet50.
3. CNN Architectures: Finally, we consider different CNN architectures to quantify their impact on inference performances.

6.6.2 Data Collection

This stage involves gathering all the necessary data required to characterize and model the CNN performances. The process entails the acquisition of four (04) distinct sets of measurements. The initial set consists of the CNN features employed as independent variables for prediction. Extracting this data is rather straightforward since it does not need the additional deployment of a CNN on edge GPU. However, the remaining three sets consist of actual measurements of execution time, power consumption, and memory usage. As a result, these cannot be acquired without consistent profiling of the CNN inference on the targeted edge GPU.

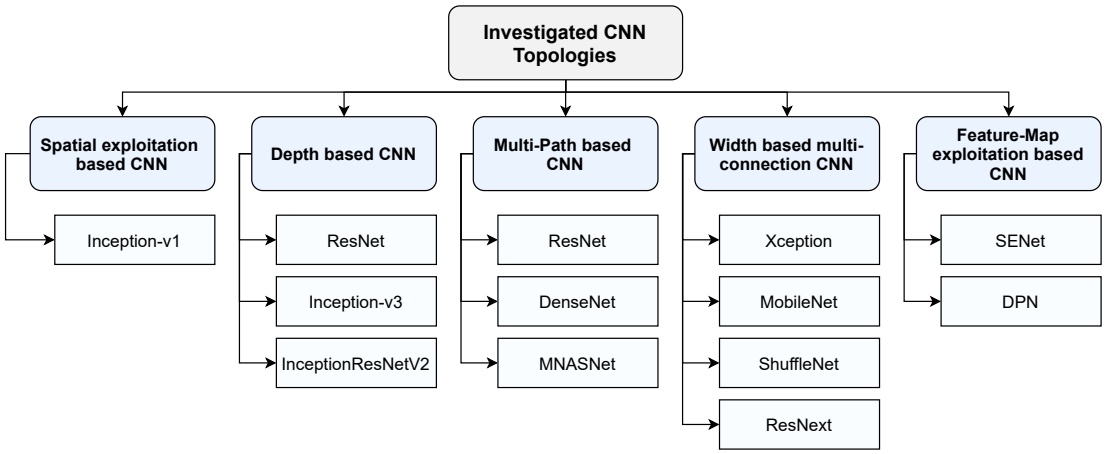


Figure 6.7 Taxonomy of the investigated CNN architectures for the Benchmarking.

The workflow of the CNN profiling procedure for collecting performance metrics is illustrated in Figure 6.8. In the following part, we provide further information pertaining to the data collection process:

① **CNN features extraction**: We have developed a parser that takes the CNN architecture description as input and gives its essential features in output. We give more details about the extracted features and their ranges in Table 6.5. From this table, we can notice that a wide range of values has been considered for each CNN feature. Thus, the studied CNNs range from small to large models according to the reported values of FLOPs, parameters, activations, and the number of layers in Table 6.5. Our benchmark covers different types and values of CNN features.

Table 6.3 Details of the CNN benchmarks used in the experiments.

Attribute	# variants	# input image sizes	Range of image sizes (from.. to..)
GoogleNet [209]	1	15	[(224x224x3),..,(1024x1024x3)]
Inception (v3) [210]	1	20	[(75x75x3),..,(1024x1024x3)]
InceptionResNet (v2) [208]	1	20	[(75x75x3),..,(1024x1024x3)]
DPN [41]	4	23	[(32x32x3),..,(1024x1024x3)]
DenseNet [94]	5	23	[(32x32x3),..,(1024x1024x3)]
Xception [47]	1	23	[(32x32x3),..,(1024x1024x3)]
EfficientNet [29]	4	25	[(32x32x3),..,(1600x1600x3)]
MNASNet [213]	5	25	[(32x32x3),..,(1600x1600x3)]
ResNet [84, 85]	19	27	[(32x32x3),..,(2400x2400x3)]
MobileNet [90, 187, 89]	13	27	[(32x32x3),..,(2400x2400x3)]
ResNext [234]	2	27	[(32x32x3),..,(2400x2400x3)]
SENet [91]	6	27	[(32x32x3),..,(2400x2400x3)]
ShuffleNet [261, 144]	7	27	[(32x32x3),..,(2400x2400x3)]

② **Execution time profiling**: CNN inference execution time measurements have been collected using the Nvidia profiling tool Nvprof [159]. We measure the CNN execution time by summing up the execution times of the GPU kernels that have been invoked during the CNN inference. To minimize the impact of the profiling overhead on the measurements, we repeat each experiment 100 times on 100

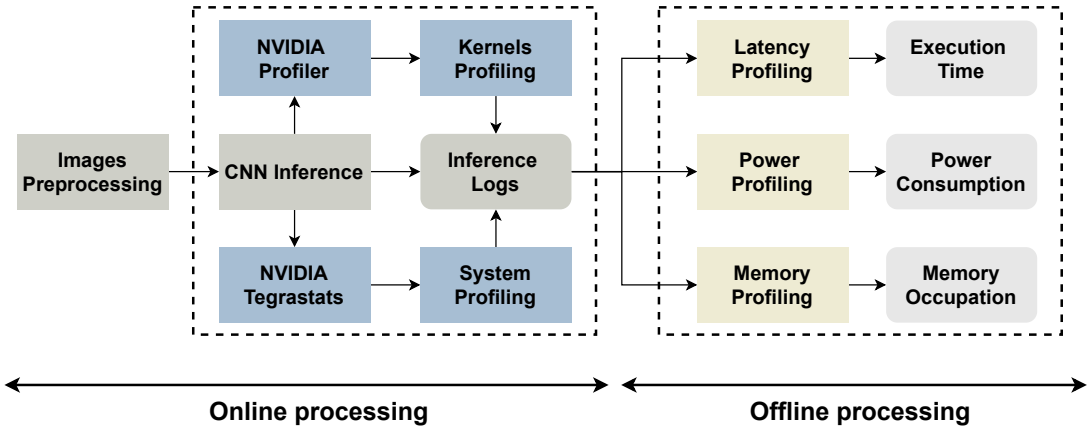


Figure 6.8 The workflow of the CNN inference profiling on the edge GPU.

randomly chosen images. We then consider the average measured execution times over the 100 experiments according to the recommendation given in [158]. We have also noticed that the measured execution times of the replicated experiments are normally distributed ; therefore, we consider the mean a central tendency.

③ **Power consumption profiling**: To measure the power consumption of the CNN inference, we use the onboard GPU sensors of NVIDIA Jetson platforms. These sensors can be read automatically and periodically with the tegrastats command utility [164]. When running CNN inference, we also run a background process to periodically examine the GPU power consumption, with a sampling period set to the minimum value (i.e., 1 ms). To ensure the reliability of the power consumption measurements, we disconnect all the peripheries connected to the GPU board during measurements except the port for SSH communication. We repeat the experiments 100 times for Jetson AGX Xavier as the inference times are generally short, and 50 times for Jetson Nano and TX2. We have noticed that measured power consumption over the repeated experiments form a skewed normal distribution as power consumption tends to be relatively high in the first runs because of the GPU warm-up. Hence, we consider the median of the peak power consumption of the replicated experiments as a central tendency.

④ **Memory Usage Profiling**: NVIDIA Jeston platforms are characterized by their unified memory. Thus, the GPU and the CPU share the same physical memory. In this case, GPU memory usage is not limited ; it can use all available system memory. This particularity of Nvidia Jetson platforms makes it possible to monitor GPU memory usage by monitoring the system memory. For this reason, we measure the peak memory usage of the CNN with the tegrastats utility [164] by monitoring the system memory usage in the same way as for power consumption. We define the CNN memory usage as the peak system memory usage during the CNN inference minus the initial memory usage before running the CNN inference.

6.6.3 Prediction Models Hyperparameters Tuning

For hyperparameters tuning, we run an exhaustive grid search on the hyperparameters' search space. During this process, we evaluate the goodness of the

explored hyperparameters via the K-fold cross-validation [184] to identify the optimal hyperparameter values and combinations. Afterward, the ML models with the optimal found hyperparameters are trained on their internal parameters using the full training dataset. Hyperparameters search space for each ML prediction algorithm are given in Table 6.4.

Table 6.4 Search space of the Hyperparameters per ML method.

Prediction model	Hyperparameters	Range	Default value
Poly	Degree	[2- 20]	2
	Lambda	[1e-8- 1.0]	1
SVR	Cost (C)	[0.5- 5000]	1
	Epsilon	[0.01- 5]	0.1
	Gamma	[0.01- 10]	scale
	Kernel	[linear- rbf- poly]	rbf
	Degree	[2- 20]	3
MLP	Hidden dim	[4- 72]	100
	Num. Layers	[1- 5]	1
	Activation	[identity- logistic- tanh- relu]	relu
	Optimizer	[sgd- adam]	adam
	Init. learning rate	[1e-4- 1e-1]	0.001
	Schedule. learning rate	[constant- adaptive]	constant
	Alpha	[1e-6- 1e-1]	0.0001
	Max. iterations	[100- 3000]	200
	Early Stopping	[False- True]	False
Random Forest	Num. estimators	[10- 500]	100
	Max. depth	[10- 300]	None
	Min. samples split.	[1- 10]	2
	Min. samples leaf	[1- 10]	1
	Max. features	[auto- sqrt- log2- None]	auto
	Bootstrap	[True- False]	True
XGBoost	Early Stopping	[True- False]	False
	Rounds	[100- 3000]	100
	Max. depth	[5- 20]	15
	Min. child weight	[5- 15]	5
	Sub.sample	[0.1- 1.0]	1
	Col. sample bytree	[0.1- 1.0]	1
	Gamma	[0.01- 10]	0.1
	Learning rate	[0.01- 0.9]	0.3

6.6.4 Prediction Models Design, Training, and Evaluation

The datasets from the benchmarking are denoted as $D_{jk} = \{cnni; y_{ijk}\}_{i=1}^{n}$, where k represents a performance metric and j for $edgpu_j$. After constructing the datasets D_{jk}, we proceed to train several sets of prediction models for each performance metric k and edge GPU $edgpu_j$. During this stage, every prediction model learns the correlation between the feature vectors cnn_i and y_{ijk}. The technique employed for designing the prediction models is depicted in Figure 6.10. This procedure has two inputs: ❶ *ML algorithm name*, which corresponds to one of the ML algorithms listed in section 6.5.3. ❷ *Collected data*, which is refers to the underlying dataset D_{jk}. Table 6.5 details the extracted CNN features from the CNN models reported in Table 6.3.

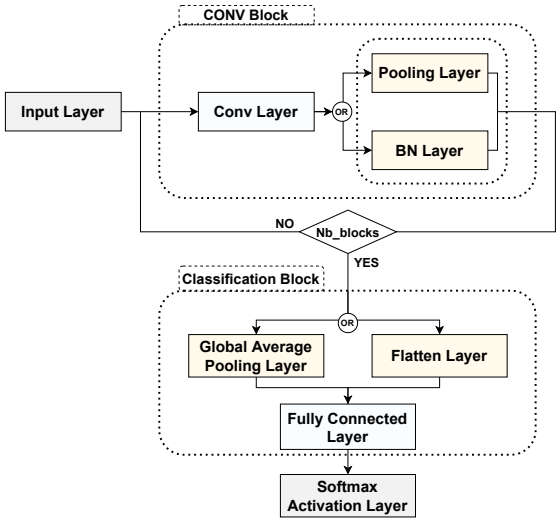


Figure 6.9 CNN baseline for NCA.

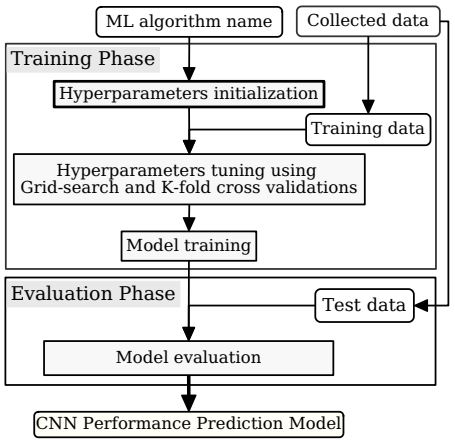


Figure 6.10 Prediction model design.

First, we tune prediction models' hyperparameters using the grid search and K-fold cross-validation techniques to select the best values. Second, once the optimal combination of hyperparameters is found, we train the prediction models on the 70% data for training. Finally, we test the final prediction models on the 30% data for evaluation. To assess the prediction accuracy and generalization, we have designed three sets of evaluation data by scaling three factors (i.e., CNN architectures, variants, and input size) as follows

— **New Image Sizes (NIS):** Here we evaluate the prediction models on state-of-the-art CNNs with new input image sizes. We vary the input size from 32×32 to 2400×2400 pixels, depending on the CNN architecture as detailed in Table 6.3. That is, for each CNN architecture, we test different input sizes. For instance, for GoogleNet [209], we vary the input size ($W \times H \times C$) from $224 \times 224 \times 3$ to $1024 \times 1024 \times 3$, whereas for DenseNet [94], we vary the input size, for each of its five (5) variants, from $32 \times 32 \times 3$ to $1024 \times 1024 \times 3$. By varying the input size of the CNN, we obtain different values of FLOPs, intermediate activations, and neurons. Tthis is the simplest exploration space as we only vary the input size of existing CNNs.

— **New CNN Variants (NCV):** We evaluate the prediction models when new CNN variants are considered in this second exploration space. Based on SOTA CNN baselines, we derive new models by varying scaling factors that control the CNN's width, depth, input size, and type of operators. These factors are, for instance, the depth and width multipliers for MobileNet [90] and EfficientNet [215], the number of layers and residual blocks for ResNet [84, 85], the number of output channels for ShuffleNet [261, 261]. If we consider ResNet [84] as an example, we trained our predictors on ResNet-18 to Resnet-101, and we predict the execution time, power consumption, and memory usage for ResNet-152 with new numbers of FLOPs, activations, neurons, parameters, and layers. Hence, this exploration space is more complicated than the first one.

— **New CNN Architectures (NCA):** Here we evaluate our prediction models on new synthetic CNN architectures. Figure 6.9 represents the CNN baseline architecture used to construct the NCA exploration space. To generate

synthetic CNNs from this baseline, we randomly vary the input size and convolutional block configurations by varying numbers and types of convolutional layers, pooling, and batch normalization layers. We also randomly vary the configurations for each layer, for instance, the number and size of kernels in convolutional layers, stride, and kernel size in pooling layers, and the number of units in fully-connected layers. This exploration space is more complicated than NIS and NCV.

Table 6.5 CNN features.

CNN feature	Range
# of hidden layers	[10 – 4072]
# of CONV layers	[4 – 1825]
# of BN layers	[0 – 265]
# of FC layers	[0 – 18]
# of filters per CONV layer	[3 – 2688]
# of units per FC layer	[0 – 5625x10^3]
Filter size	[(1x1) – (11x11)]
Input size	[32 – 2400]
# of CONV layers parameters	[0.27x10^6 – 87.1x10^6]
# of BN layers parameters	[0 – 0.8x10^6]
# of FC layers parameters	[0 – 119.8x10^6]
Total number of FLOPs	[3.1x10^6 – 5.2x10^{12}]
Sum of intermediate activations	[0.1x10^6 - 37.6x10^9]

Table 6.6 Hardware setup.

Hardware feature	Jetson Nano	Jetson TX2	Jetson AGX
CPU	4-core A57 1.43 GHz	6-core Denver 2 GHz	8-core Carmel 2.26 GHz
Memory size	4 GB 64-bit LPDDR4	8 GB 128-bit LPDDR4	16 GB 256-bit LPDDR4x
Memory BW	25.6 GB/s	58.4 GB/s	137 GB/s
GPU	128-core Maxwell 1.23 GHz	256-core Pascal 1.3 GHz	512-core Volta 1.37 GHz
Max Power	10W	15W	30W

6.6.5 Experimental Setup

We deploy more than 1500 CNN models on different edge GPUs to characterize the inference performances. We use three Nvidia GPU devices from the Jetson series dedicated to edge computing: Jetson Nano, Jetson TX2, and Jetson AGX Xavier. The hardware specifications for each device are reported in Table 6.6. We configure these platforms to the MAXN power mode. We use the same underlying software configuration on the three (03) edge GPUs. As depicted in Table 6.6, the three devices differ regarding computing/memory capacities. CNNs have been deployed on the edge GPUs using the Keras 2.3.1 API with TensorFlow 1.14 as backend [4].

6.6.6 Experimental Results

This section comprehensively evaluates and analyzes our proposed performance modeling methodology. Each prediction model undergoes an evaluation as follows:

— First, we evaluate the prediction error of each prediction model using the Mean Absolute Percentage Error (MAPE) [50].

— Second, we compare the prediction models according to the time needed for: training, tuning their corresponding hyperparameters, running the prediction model on a single input features vector (cnn_i).

— Third, we discuss the correlation coefficients between the measured and predicted performance metrics. To assess the extent to which prediction models adhere to the ranking of performance metric measurements, we employ both Kendall-Tau [6] and Pearson [12] rank correlation coefficients.

6.6.6.1 Execution Time Prediction

Figure 6.11 gives the obtained Mean Absolute Percentage Error (MAPE) and their corresponding confidence interval of 95%. Table 6.7 summarizes the analysis of the five (05) prediction models in terms of MAPE, training, tuning costs, and prediction latency. Table 6.8 reports the calculated rank correlation coefficients of all the obtained results for CNN execution time estimations.

Table 6.7 Execution time prediction models analysis.

Prediction Model	Test Data	MAPE			Training Time	Tuning Time	Prediction Latency		
		Nano	TX2	AGX			AGX	TX2	Nano
Poly	NIS	10.16%	10.29%	9.55%	**12.72 ms**	**6.32 mn**	**357.9 ns**	**528.1 ns**	**704.3 ns**
	NCV	11.55%	11.06%	10.20%					
	NCA	11.54%	13.19%	13.04%					
MLP	NIS	10.98%	12.47%	12.17%	1.13 s	11.09 hr	22.7 us	30.9 us	53.8 us
	NCV	12.37%	9.97%	13.80%					
	NCA	13.65%	15.18%	13.18%					
SVR	NIS	11.83%	14.97%	14.68%	159 ms	22.6 hr	41.8 us	52.1 us	78.5 us
	NCV	**9.29%**	**9.30%**	**7.92%**					
	NCA	16.01%	16.86%	15.86%					
RF	NIS	9.39%	10.97%	11.94%	3.5 s	4.6 hr	1.1 ms	1.5 ms	2.4 ms
	NCV	**9.14%**	**8.91%**	**10.41%**					
	NCA	13.97%	13.79%	13.76%					
XGBoost	NIS	**9.12%**	**9.58%**	**8.28%**	**538.5 ms**	**13.22 mn**	**2.1 us**	**3.5 us**	**6.2 us**
	NCV	**7.99%**	**9.09%**	**7.45%**					
	NCA	**13.80%**	**13.11%**	**11.74%**					

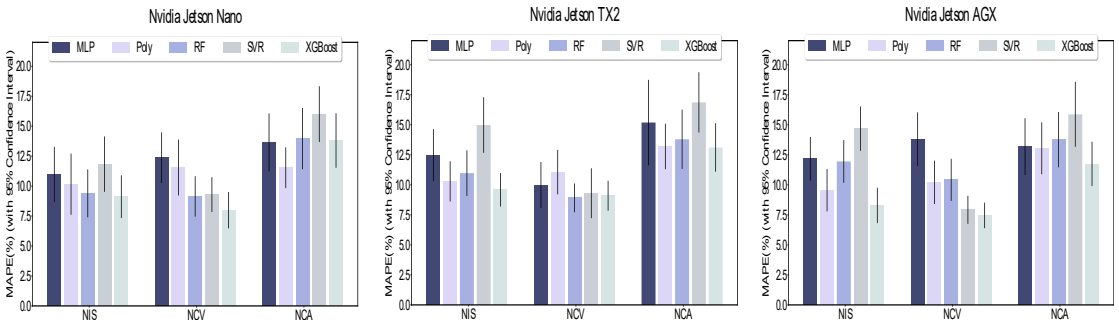


Figure 6.11 Mean Absolute Percentage Error (MAPE) for execution time prediction on the three edge GPUs: Nvidia Jetson Nano, TX2, and AGX, respectively, with the corresponding 95% Confidence Interval.

① **Prediction error and generalization power:** From Figure 6.11 and Table 6.7, we notice that the MAPE values vary between $\sim$ 7% and $\sim$ 16%. It is worth noting that the NIS and NCV consistently yield the lowest MAPE values, whereas the NCA consistently yields the highest MAPE values. This finding corroborates the hypothesis on the complexity of NCA compared to NIS and NCV, as previously detailed in Section 6.6. For the three exploration spaces (i.e., NIS, NCV, and NCA), XGBoost and Ridge Polynomial Regression outperform the other prediction models and offer the lowest MAPE values.

RF depicts comparable performance to XGBoost and Polynomial regression. MLP generally has good performances with a slight loss of generalization for NIS and NCA. This nonperformance can be due to its nature which tends to overfit data. SVR is less accurate than the models mentioned above for NIS and NCA.

The variation in the MAPE values in the three exploration spaces is relatively high, interpreted as overfitting. If we compare the obtained results of the three edge GPUs: Jetson Nano, TX2, and AGX, we notice that the prediction errors regarding the three (3) exploration spaces are very similar.
② **Training and tuning costs**: Most ML-based prediction models are sensitive to hyperparameters variations. For instance, XGBoost shows sensitivity to the Booster hyperparameters. The prediction error converges by increasing the number or depth of the decision trees in Random Forest. These two (2) factors increase the complexity of the training and the prediction latency in RF. For this reason, when a time budget is available for tuning and training, XGBoost is more efficient.

MLP is very sensitive to the variation of the hyperparameters. Large MLP networks are prone to overfitting compared to small ones. SVR is sensitive to the type of kernels and the cost (C) (see Table 6.4). From table 6.7, we notice that SVR takes the longest hyperparameter tuning time as training Polynomial and RBF kernels is time-consuming.

Table 6.8 Rank correlation coefficients analysis.

Prediction model	Test Data	Kendall coefficients			Pearson coefficients		
		Nano	TX2	AGX	Nano	TX2	AGX
Poly	NIS	0.949	0.935	0.93	0.998	0.998	0.997
	NCV	0.938	0.935	0.93	0.984	0.998	0.997
	NCA	0.961	0.935	0.93	0.995	0.998	0.997
MLP	NIS	0.937	0.936	0.932	0.998	0.994	0.996
	NCV	0.935	0.936	0.932	0.982	0.994	0.996
	NCA	0.955	0.936	0.932	0.992	0.994	0.996
SVR	NIS	0.932	0.924	0.93	0.995	0.989	0.994
	NCV	0.94	0.924	0.93	0.982	0.989	0.994
	NCA	0.949	0.924	0.93	0.993	0.989	0.994
RF	NIS	0.95	0.935	0.933	0.997	0.994	0.997
	NCV	0.945	0.935	0.933	0.979	0.994	0.997
	NCA	0.945	0.935	0.933	0.991	0.994	0.997
XGBoost	NIS	**0.95**	**0.948**	**0.942**	**0.994**	**0.995**	**0.997**
	NCV	**0.953**	**0.948**	**0.942**	**0.979**	**0.995**	**0.997**
	NCA	**0.945**	**0.948**	**0.942**	**0.982**	**0.995**	**0.997**

③ **Prediction rank-preserving**: We analyze the rank correlation between the measured and predicted inference execution times to determine whether the prediction models are rank-preserving. To assess models rank-preserving, we use Kendall-tau and Pearson correlation coefficients. These correlation coefficients evaluate the nature and the degree of similarity between the two (2) sets of data: the sets of measured and predicted inference execution times. Table 6.8 details the obtained rank correlation coefficients. As seen, all prediction models highly preserve the rank as their Kendall coefficients range from 0.94 to 0.97, and their Pearson coefficients range from 0.98 to 0.99. Thus, our prediction models can be used to rank the CNN architectures according to their estimated execution times.

6.6.6.2 Power Consumption Prediction

Figure 6.12 gives the obtained MAPE values with their corresponding confidence interval of 95%. Table 6.9 summarizes the analysis of the five (05) prediction models for CNN power consumption in terms of MAPE, training, tuning costs, and

prediction latency. Table 6.10 reports the calculated rank correlation coefficients of all the obtained results for CNN power consumption estimations.

Table 6.9 Power consumption prediction models analysis.

Prediction Model	Test Data	MAPE			Training Time	Tuning Time	Prediction Latency		
		Nano	TX2	AGX			AGX	TX2	Nano
Poly	NIS	6.18%	7.02%	5.85%	**11.29 ms**	**6.09 mn**	**401.1 ns**	**713.9 ns**	**900.6 ns**
	NCV	6.51%	6.97%	5.66%					
	NCA	7.10%	7.05%	5.71%					
MLP	NIS	5.66%	6.72%	5.72%	1.9 s	15.33 hr	23.2 us	27.4 us	39.0 us
	NCV	7.96%	7.41%	6.43%					
	NCA	7.22%	7.19%	6.03%					
SVR	NIS	**4.51%**	**5.14%**	**3.94%**	771.3 ms	25.1 hr	201.3 us	511.2 us	803.5 us
	NCV	5.39%	6.50%	5.33%					
	NCA	8.29%	8.57%	7.81%					
RF	NIS	4.96%	5.82%	5.88%	3.31 s	4.6 hr	908.5 us	1.2 ms	1.7 ms
	NCV	6.94%	5.84%	4.13%					
	NCA	5.18%	6.75%	5.81%					
XGBoost	NIS	**3.56%**	**5.24%**	**4.51%**	**488.5 ms**	**10.12 mn**	**4.7 us**	**6.0 us**	**8.9 us**
	NCV	**4.91%**	**5.20%**	**5.69%**					
	NCA	**4.62%**	**6.04%**	**5.31%**					

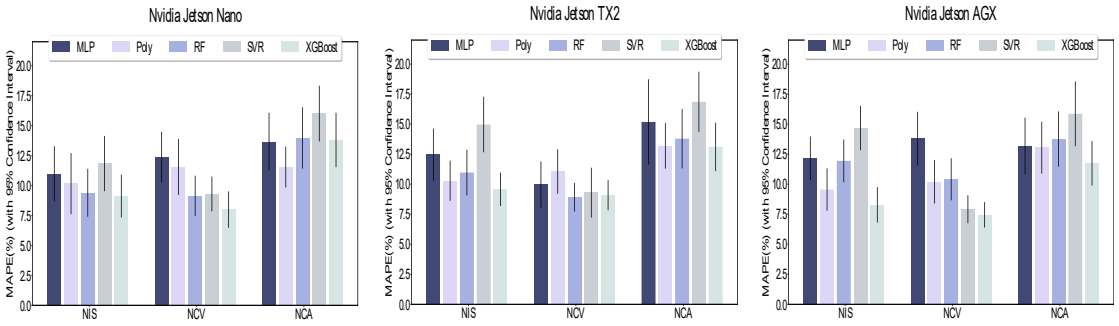


Figure 6.12 Mean Absolute Percentage Error (MAPE) for power consumption prediction on the three edge GPUs: Nvidia Jetson Nano, TX2, and AGX, respectively, with the corresponding 95% Confidence Interval.

① **Prediction error and generalization power:** Firstly, the lowest prediction errors are obtained for NIS and NCV, while the highest is for NCA. Secondly, both XGBoost and Ridge polynomial regression provide the lowest prediction error for power consumption estimation on the 03 edge GPUs. These two models also achieve a good generalization between the three exploration spaces. From Figure 6.12 and Table 6.9, we can also notice that the MAPE values are generally between $\sim 3\%$ and $\sim 8\%$, which indicates both reduced bias and variance errors. This figure also shows that the obtained MAPE for power consumption is smaller than the obtained MAPE for execution time, as edge GPUs have a limited power budget. RF and MLP depict similar performances in terms of prediction error and generalization. SVR provides low prediction error for NIS and NCV but a quiet loss of generalization for NCA.

② **Training and tuning costs**: From Table 6.9, we observe that the training and tuning costs of the prediction models for power consumption have similar tendencies as the execution time. Nonetheless, we have also noticed differences in the

obtained optimal hyperparameters of each prediction model compared to execution time modeling. We highlight in the following the main observed differences:

— MLP presents huge overfitting when increasing the network size in the case of execution time. However, small MLP networks are not enough for an accurate prediction. Instead, they result in considerable underfitting. Consequently, we consider large MLP networks for power consumption modeling.

— Although linear kernels for SVR perform well for execution time they provide poor performances for power consumption. On the contrary, we noticed that RBF kernels provide lower prediction errors. This is explained by the non-linear correlation between input features and power consumption, as in between the number of FLOPs and power consumption (See Figure 6.4).

Table 6.10 Rank correlation coefficients analysis.

Prediction model	Test data	Kendall coefficients			Pearson coefficients		
		Nano	TX2	AGX	Nano	TX2	AGX
Poly	NIS	0.769	0.926	0.94	0.952	0.987	0.995
	NCV	0.911	0.926	0.94	0.985	0.987	0.995
	NCA	0.657	0.926	0.94	0.935	0.987	0.995
MLP	NIS	0.758	0.942	0.931	0.977	0.994	0.994
	NCV	0.88	0.942	0.931	0.976	0.994	0.994
	NCA	0.74	0.942	0.931	0.966	0.994	0.994
SVR	NIS	0.765	0.952	0.957	0.974	0.996	0.996
	NCV	0.899	0.952	0.957	0.983	0.996	0.996
	NCA	0.542	0.952	0.957	0.896	0.996	0.996
RF	NIS	0.836	0.935	0.934	0.976	0.992	0.992
	NCV	0.9	0.935	0.934	0.98	0.992	0.992
	NCA	0.743	0.935	0.934	0.97	0.992	0.992
XGBoost	NIS	**0.905**	**0.945**	**0.947**	**0.979**	**0.995**	**0.996**
	NCV	**0.934**	**0.945**	**0.947**	**0.991**	**0.995**	**0.996**
	NCA	**0.737**	**0.945**	**0.947**	**0.975**	**0.995**	**0.996**

③ **Prediction rank-preserving**: Table 6.10 shows the obtained correlation analysis for power consumption prediction models. We can see that XGBoost, RF, and MLP provide the highest Kendall and Pearson correlation coefficients. These coefficients range from 0.7 to 0.9 for Kendall and 0.8 to 0.9 for Pearson. On the other hand, SVR and Ridge Polynomial regression results are less correlated, especially for the Jetson Nano. The reason behind the low Kendall-tau values for Jetson Nano comes from the limited range of power consumption values. Generally, we have observed that obtained power consumption values are less distributed and tend to have very similar values to the execution time case, where values have a higher degree of diversity. It is also worth noting that the overfitting problem of SVR appears clearly in the obtained correlation coefficient values for NCA. This observation is also proven by the obtained MAPE results reported in Table 6.9, indicating a considerable generalization loss for NCA.

6.6.6.3 Memory Usage Prediction

Figure 6.13 gives the obtained MAPE values and their corresponding confidence interval of 95%. Table 6.11 summarizes the analysis of the five (05) prediction models in terms of MAPE, training, tuning costs, and prediction latency. Table 6.12 reports the rank correlation coefficients of all the results for CNN memory usage estimations.

Table 6.11 Memory usage prediction models analysis.

Prediction Model	Test Data	MAPE			Training Time	Tuning Time	Prediction Latency		
		Nano	TX2	AGX			AGX	TX2	Nano
Poly	NIS	**9.73%**	**4.91%**	**4.38%**	**5.13 ms**	**6.09 mn**	**89.3 ns**	**140.5 ns**	**270.2 ns**
	NCV	**9.15%**	**5.01%**	**4.78%**					
	NCA	**6.43%**	**6.87%**	**5.47%**					
MLP	NIS	11.03	4.98%	5.85%	341.6 ms	13.8 hr	14.7 us	24.0 us	34.7 us
	NCV	14.83%	5.60%	4.92%					
	NCA	11.59%	9.04%	7.62%					
SVR	NIS	7.89%	6.65%	**3.62%**	101.2 ms	20.7 hr	96.6 us	121.3 us	249.6 us
	NCV	8.90%	**5.12%**	**3.64%**					
	NCA	7.82%	8.57%	6.78%					
RF	NIS	7.25%	6.22%	4.58%	1.62 s	3.1 hr	261.6 us	397.0 us	508.2 us
	NCV	8.15%	4.96%	6.83%					
	NCA	10.46%	9.18%	7.63%					
XGBoost	NIS	**9.03%**	**6.46%**	**4.10%**	**338.5 ms**	**7.2 mn**	**708.2 ns**	**933.8 ns**	**1.1 us**
	NCV	10.14%	**5.41%**	**5.72%**					
	NCA	7.60%	**8.73%**	**7.40%**					

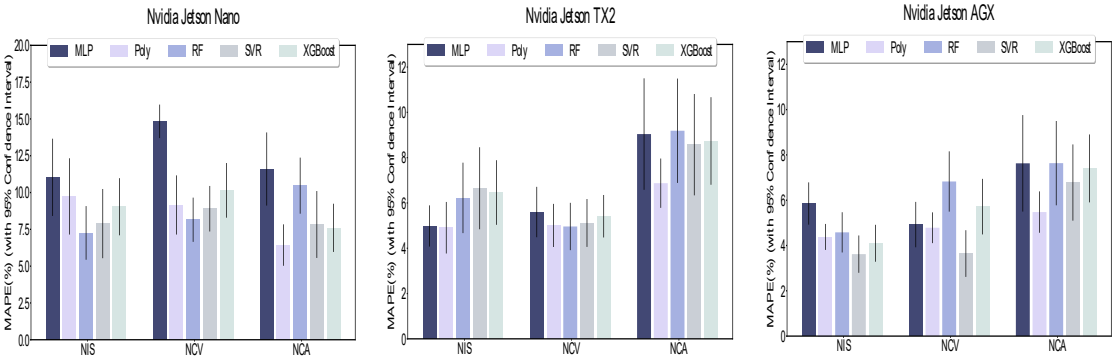


Figure 6.13 Mean Absolute Percentage Error (MAPE) for memory usage prediction on the three edge GPUs: Nvidia Jetson Nano, TX2, and AGX, respectively, with the corresponding 95% Confidence Interval.

① **Prediction error and generalization power:** In contrast with the two previous performance metrics, where XGBoost and Ridge Polynomial regression provides similar results, Ridge Polynomial regression outperforms XGBoost and the other models for memory usage. Indeed, training datasets for memory usage differ from those for execution time and power consumption. We have performed two reduction techniques on the training datasets for memory usage:

1. Considering only the smallest and the largest input size for each CNN depending on the reported values in Table 6.3. From our preliminary experiments, we noticed that memory usage does not vary significantly with input sizes.
2. Selecting only the prediction features related to memory operations.

These two data reduction techniques result in thoroughly different training datasets compared to the cases of execution time and power consumption. From 6.13 and Table 6.11, MLP, SVR, and RF provide similar performances in terms of MAPE values. As for the execution time and power consumption cases, reported prediction errors for the 03 edge GPUs are very close. This confirms the portability of our modeling methodology for new edge GPUs.

② **Training and tuning costs**: Training and tuning costs for memory usage models are lower than the execution time and power consumption models. These costs are

correlated with the size and dimensionality of the training dataset. In contrast to power consumption modeling, smaller MLP networks perform better than larger networks. Besides, we noticed that RBF and Poly kernels deliver the best performances for SVR compared to linear ones. Thus, SVR, MLP, and RF are the most challenging models to tune for the three performance metrics and the three edge GPUs.

Table 6.12 Rank correlation coefficients analysis.

Prediction model	Test data	Kendall coefficients			Pearson coefficients		
		Nano	TX2	AGX	Nano	TX2	AGX
Poly	NIS	**0.833**	**0.909**	**0.92**	**0.994**	**0.994**	**0.993**
	NCV	**0.871**	**0.909**	**0.92**	**0.995**	**0.994**	**0.993**
	NCA	**0.818**	**0.909**	**0.92**	**0.942**	**0.994**	**0.993**
MLP	NIS	0.858	0.912	0.905	0.994	0.992	0.988
	NCV	0.865	0.912	0.905	0.996	0.992	0.988
	NCA	0.784	0.912	0.905	0.912	0.992	0.988
SVR	NIS	0.824	0.875	0.923	0.991	0.987	0.993
	NCV	0.86	0.875	0.923	0.995	0.987	0.993
	NCA	0.79	0.875	0.923	0.908	0.987	0.993
RF	NIS	0.828	0.9	0.917	0.991	0.992	0.993
	NCV	0.848	0.9	0.917	0.991	0.992	0.993
	NCA	0.794	0.9	0.917	0.924	0.992	0.993
XGBoost	NIS	**0.82**	**0.885**	**0.907**	**0.991**	**0.991**	**0.993**
	NCV	**0.829**	**0.885**	**0.907**	**0.991**	**0.991**	**0.993**
	NCA	**0.783**	**0.885**	**0.907**	**0.926**	**0.991**	**0.993**

③ **Prediction rank-preserving** As seen in Table 6.12, the rank is highly respected between the measured and predicted CNN memory usage values by the prediction models. Kendall and Pearson correlation coefficients range from 0.8 to 0.9 and from 0.95 to 0.98, respectively, indicating a strong positive correlation between the two data sets (i.e., measured and predicted memory usage). We must also note that the prediction models follow the same tendency for Kendall and Pearson correlation coefficients in all cases (i.e., exploration spaces and edge GPUs).

6.7 Discussion and Key Insights

In this section, we discuss the main advantages and limitations of our work in terms of hardware and software adaptability. Using a set of benchmarks that we built, we were able to evaluate and estimate CNN performances on edge GPUs. In summary:

— **(i)** Our Machine Learning based modeling methodology to characterize CNNs performances for edge GPUs gives predictions for execution time, power consumption, and memory usage with up to ∼92% prediction accuracy.

— **(ii)** Our resulting prediction models have been evaluated on a large set of CNN architectures and three (03) commodity edge GPUs with different hardware resources and micro-architectures.

Furthermore, Our benchmarks are composed of small and large, simple and complex, conventional and recent SOTA CNNs and variants. In Tables 6.13, 6.14, and 6.15, we present a comparison of our work with relevant prior studies on CNN performance modeling. We use the MAPE to assess the prediction error. For execution time, the prediction error of our models is the lowest for GPUs compared

to related works. In addition, We have obtained the smallest MAPE for power consumption and memory usage compared to other related works.

Table 6.13 SOTA models Vs. our models for execution time prediction.

Ref.	CNNs	System	MAPE
[143]	NIN, VGG19M	TK1 CPU – Caffe	4.71%
		TK1 GPU – Caffe	23.70%
	NIN, VGG19M, SqueezeNet, MobileNet	TX1 CPU – Caffe	39.91%
		TX1 GPU – Caffe	31.51%
[28]	VGG16, AlexNet, NIN, Overfeat, CIFAR10-6conv	Titan X GPU – TensorFlow	7.96%
		GTX1070 GPU – TensorFlow	12.32%
	AlexNet, NIN	GTX1070 GPU – Caffe	16.17%
[218]	AlexNet, All-CNN-C, MobileNet, ResNet-18, SimpleNet, SqueezeNet, Tiny YOLO	RPi3 CPU – Caffe	5.02%
		RPi3 CPU – OpenCV	7.92%
		XU4 CPU – Caffe	3.25%
Our work	ResNets, MobileNets, EfficientNets, ShuffleNets, SENets, DenseNets, GoogleNet, Inception, SqueezeNets, MnasNets, DPN, Xception	Jetson Nano GPU – TensorFlow	**8.55%**
		Jetson TX2 GPU – TensorFlow	**9.33%**
		Jetson AGX GPU – TensorFlow	**7.86%**

Table 6.14 SOTA models Vs. our models for power consumption prediction.

Ref.	CNNs	System	MAPE
[143]	NIN, VGG19M, SqueezeNet, MobileNet	TX1 CPU – Caffe	39.08%
		TX1 GPU – Caffe	15.30%
[28]	VGG16, AlexNet, NIN, Overfeat, CIFAR10-6conv	Titan X GPU – TensorFlow	2.25%
		GTX1070 GPU – TensorFlow	8.40%
	AlexNet, NIN	GTX1070 GPU – Caffe	21.99%
[218]	AlexNet, All-CNN-C, MobileNet, ResNet-18, SimpleNet, SqueezeNet, Tiny YOLO	RPi3 CPU – Caffe	8.52%
		RPi3 CPU – OpenCV	7.24%
		XU4 CPU – Caffe	10.46%
Our work	ResNets, MobileNets, EfficientNets, ShuffleNets, SENets, DenseNets, GoogleNet, Inception, SqueezeNets, MnasNets, DPN, Xception	Jetson Nano GPU– TensorFlow	**4.23%**
		Jetson TX2 GPU – TensorFlow	**5.22%**
		Jetson AGX GPU – TensorFlow	**5.10%**

Table 6.15 SOTA models Vs. our models for memory usage prediction.

Ref.	CNNs	System	MAPE
[143]	NIN, VGG19M	TK1 CPU – Caffe	39.89%
		TK1 GPU – Caffe	34.33%
		TX1 CPU – Caffe	49.92%
		TX1 GPU – Caffe	40.94%
Our work	ResNets, MobileNets, EfficientNets, ShuffleNets, SENets, DenseNets, GoogleNet, Inception, SqueezeNets, MnasNets, DPN, Xception	Jetson Nano GPU – TensorFlow	**9.72%**
		Jetson TX2 GPU – TensorFlow	**5.93%**
		Jetson AGX GPU – TensorFlow	**4.91%**

We have demonstrated that our modeling methodology can be ported to any edge GPU to predict the performance of computer vision CNN. Furthermore, by using our proposed benchmarks and modeling methodology, the designer can quickly build prediction models by following our proposed modeling steps on new edge devices, new use cases, or new metrics. Nevertheless, the ability to predict the performance of a set of CNNs on an entirely new edge hardware GPU – without repeating the same pipeline of benchamking, tuning, and training new performance prediction models – is not convenient because of the following reasons:

— **(i)** In our work, we propose an ML-based modeling approach for predicting CNNs performances on edge GPUs. However, as these embedded hardware

devices have been recently proposed in the market, the number of their hardware configurations is limited. Generalizing the predictions models to a new and unknown edge hardware device without additional benchmarking, tuning, and training is not possible.

— **(ii)** Modeling the architectural hardware features of edge GPU devices is complicated due to their complex integrated nature, where both the GPU and CPU share the same memory and the differences in their micro-architectures.

— **(iii)** Building accurate analytical models for these edge GPUs requires modeling the holistic and hierarchical levels of the CNN execution stack. This stack comprises the CNN structure, the DL SDK, the compiler, and finally, the hardware micro-architecture. This approach is very challenging as some of the details concerning the HW and SW are kept confidential by the GPU manufacturers (e.g., the NVIDIA CUDA compiler).

We are also aware of the limitations of our prediction models for non-computer vision CNN. The current prediction models are specific to computer vision applications as they were trained on CNN benchmarks for image classification. However, the overall proposed modeling methodology, from data collection and feature extraction to ML-based modeling (See Figures 6.2 and 6.10), can be applied to other CNN-based applications (e.g., audio recognition, natural language processing).

6.8 Summary

In this chapter, we proposed a modeling methodology for CNN performance on edge GPUs. Our approach includes two main parts: First, we characterized the CNN architectures at a model-level granularity to extract the most impacting features. Second, we implemented and compared five of the most efficient Machine Learning algorithms to build our performance prediction models. The resulting prediction models can be used to predict execution time, memory usage, and power consumption of any new CNNs on the studied three (03) edge GPUs without retraining the prediction models. Our modeling methodology has been easily generalized over the three exploration spaces: NIS, NCV, and NCA, on the three different NVIDIA edge GPUs from the Jetson series, namely, Nano, TX2, and AGX Xavier. To evaluate our approach, we analyzed the performance of the prediction models from the following perspectives:

— **(1)** Prediction error and rank-preserving,

— **(2)** Prediction latency,

— **(3)** Training and tuning costs of the models.

, Our evaluation has seen the superiority of XGBoost, Random Forest, and Ridge Polynomial regression to estimate execution time with an average error of 10%. Ridge Polynomial regression and XGBoost give similar performances for power consumption with an average error of 6%. Finally, for memory usage, Ridge polynomial regression outperforms the rest of the prediction models with an average error of 8%. Experimental results demonstrated that XGBoost and Ridge polynomial regression can provide an acceptable trade-off between prediction performance and tuning/training/inference time. Thus, they can be further integrated into a multi-objective optimization framework to accelerate the process of Neural Architecture Search (NAS) on edge GPU devices.

Chapitre 7

SONATA: Self-adaptive Evolution for Multi-objective Hardware-aware Neural Architecture Search

7.1 Introduction

The design process of NNs can be articulated as a bi-level optimization problem where neural architectural parameters (i.e., layers and operators) and neuron weights are both searchable. However, tuning the NN architectural parameters, learning the neurons' weights, and choosing the adequate hardware configuration for deployment is labor intensive. Indeed, both design domains lack interpretability and explainability regarding the obtained performance [135]. Recently, efforts have been shifted towards automating the design process of NN with Hardware-awareness through the *HW-aware NAS* paradigm [33]. Still, the search space of NN is relatively large to explore because of the time-consuming trial-error iterative process that needs an expensive optimization budget.

Earlier *HW-aware NAS* approaches employ grid or random search to tackle the stagnation in local optima [38]. Nevertheless, these approaches are time-wasting as NN architectures are generated randomly and explored unthinkingly without a well-established basis. Then, evolutionary search approaches have been applied to lower the uncertainty by contributing fitness evaluation functions that help guide the search process to favor the exploitation of the best-performing NNs and exploration of randomly generated ones. Following this mechanism, NN neighborhoods are generated in a manner that jeopardizes both exploration and exploitation [142]. However, this is suboptimal as the evolution strategy needs to generate new populations using uniform evolution operators such as *mutation* and *crossover* (See Figure 7.1). For instance, the mutation is randomly applied to *add*, *alter*, or *remove* one or more neural components/operators to create a new *offspring* population from a set of best-performing NN architectures. Similarly, in a crossover, sets of neural components/operators are randomly selected and combined from different NN architectures to produce a new *offspring* population.

Both *mutation* and *crossover* in evolutionary search are founded upon the basis of the *natural selection* theorem, assuming that *the combination of genes from well-designed genomes always enhances the design and performances of the resulting child genome*. However, this assumption does not always hold regarding NP-hard problems, such as *HW-aware NAS*, as uniform evolution operators between well-performing NN architectures may result in poorly performing NNs. Consequently, this can steer towards exploring ill-designed NNs with worse performances and wasting an expensive search budget. Thus, carefully choosing where and when to apply the evolution operators is necessary to adjust the search process. Such exper-

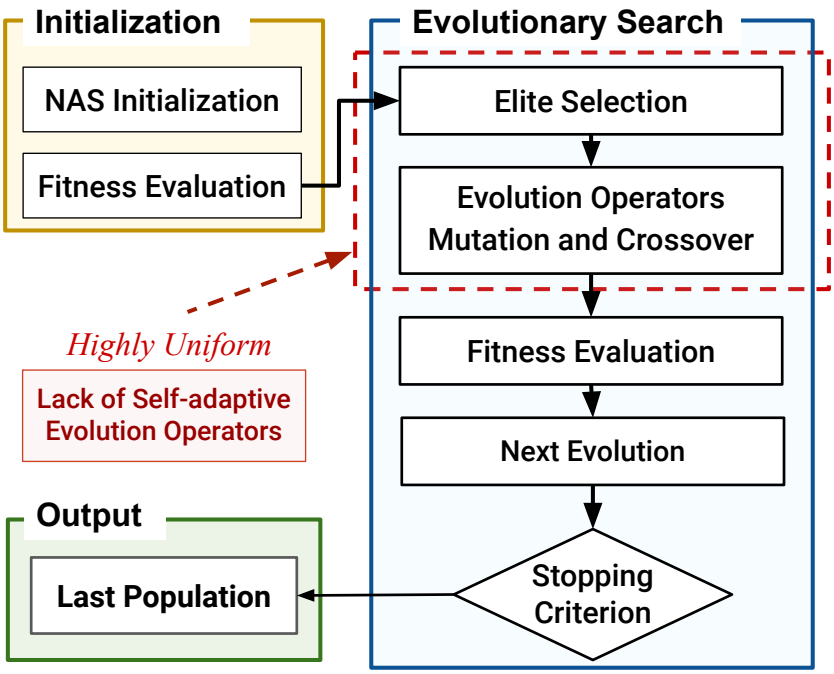


Figure 7.1 The flowchart of a typical Evolutionary NAS (ENAS).

tise can be acquired during the search by exploiting the evaluated populations to build knowledge on the search space that can help establish self-adaptive evolution operators to better guide the search algorithm towards the global optima.

Data-driven approaches are widely used to improve the design and performance of NNs following the *ML-for-ML* paradigm. For instance, data-driven surrogate modeling leverages evaluation data of NNs to train a model capable of estimating the performances of unseen ones [137]. However, data-driven approaches have been limited to NN performance estimation to accelerate the fitness evaluation in evolutionary HW-aware NAS [142, 77]. Viewing things from a fresh angle, data-driven ML-based models can also be used to learn and infer the importance of NN design parameters to guide the evolution of search operators in Evolutionary NAS (ENAS) approaches. Alternatively, *How can we better use the generated data on sampled NN architectures and their performance evaluation during the search process to build ML-based models capable of guiding the search algorithm through assessing the importance of NN design parameters ?*

Addressing these challenges, we propose `SONATA`, a self-adaptive evolutionary search for HW-aware NAS. Our method leverages ML-based evolution operators (i.e., mutation and crossover) that progressively learn the importance of NN design variables and intensify the search accordingly. Comprehensive evaluations across various NAS search spaces and edge devices demonstrate that our approach improves upon the baseline ENAS, such as NSGA-II [52], with an accuracy improvement up to ∼**0.25%** on the ImageNet-1k dataset and latency/energy gains up to ∼**2.42x**. Contributions and results of this chapter have been submitted in:

— [26] **Halima Bouzidi**, Hamza Ouarnoughi, Abdessamad Ait El Cadi, El-Ghazali Talbi, and Smail Niar, "Sonata: Self-adaptive Evolution for Multi-objective Hardware-aware Neural Architecture Search". Submitted to IEEE Transactions on Evolutionary Computation, 2024.

7.2 Related Works

7.2.1 Evolutionary Neural Architecture Search (ENAS)

Evolutionary algorithms are employed in more than 60% of NAS frameworks due to their efficiency and flexibility in handling large search spaces of complex encoding schemes [138]. As typical evolutionary search methods, ENAS requires extensive evaluation and trial-error iterations, which turns out to be time-consuming for HW-aware NAS problems [168]. However, leveraging performance predictors into the ENAS framework can reduce the search time [168]. The key components of ENAS involve the following:

— (*i*) *Genome encoding*, which defines how a single NN architecture is represented – ENAS generally operates on the adjacency matrix of the NN using a one-hot/categorical encoding [228] or by learning an embedding of the directed acyclic graph embedding using GCN or GNN [194].
— (*ii*) *Mutation,* which defines the changes to be made on a given NN representation to generate a new one through adding, altering, or removing parameters (e.g., operator node, inter-layer connection).
— (*iii*) *Crossover*, which determines how two NN representations are combined to derive a new one.
— (*iv*) *Elite selection strategy*, which renders a set of Pareto optimal solutions as elite genomes that will undergo *Mutation* and *crossover* steps [201].

ENAS generally employ Pareto ranking methods to sort genomes according to their dominance and diversity in the objective space using Tournament selection and crowding distance measurements [51].

7.2.2 Surrogate-assisted Multi-objective ENAS (SaMo-ENAS)

Given the high convexity observed in HW-aware NAS problems, simple analytical models can not be used to infer insights on the correlation between NN architectures and the obtained performances. Consequently, black-box data-driven surrogate models have been incorporated to accelerate the ENAS process [61]. These surrogate models can be used at different levels of the ENAS algorithm:

— (*i*) *Fitness evaluation*: Performance predictors can be used to estimate the accuracy, latency, or energy of the sampled NNs during the search process. The predictors can be either trained a priori (i.e., before the search) by sampling a set of NNs, evaluating them -through training on the target task and deployment on the HW device- to get the ground truth labels, then training ML-based predictors on the NNs representations and their performance measurements. In literature, predictors are typically built upon Random forest [171], XGBoost [168], GNN [226], or Gaussian process [71, 34, 45].
— (*ii*) *Initialization and Sampling*: By gathering pre-knowledge on the quality of the search to enhance the sampling of new NNs using Multifidelity estimators [222], Gaussian process [45], and clustering methods [217].
— (*i*) Neighborhood generation: In ENAS, through mutation and crossover, a neighborhood is generated to explore the next generation of NNs to balance the exploration-exploitation tradeoff. In that sense, [42] was the first to introduce a learning-based mutation [42] using a reinforcement learning

controller. Probabilistic models of mutation have been introduced in [238] to self-adapt the mutation strategy during the search in a block-wise level of the NN. In that same spirit, [179] proposes a mutation controller based on the NN model size.

However, most of the existing works do not explore data-driven methods to learn the importance of NN design variables. Incorporating knowledge that are collected at run-time to build a fully self-adaptive search strategy has not been explored. To the best of our knowledge, our work is the first to address this issue by proposing a purely data-driven approach that explored the search history in ENAS to train surrogate models that learn and predict the importance of design parameters at each evolutionary generation. The learned importance scores will be used to guide the evolution operators in a way that assists guiding the search to more optimal NN designs.

7.3 Novel Scientific Contributions

In the realm of our observations and motivations summarized in the above section, we introduce the following novel contributions:

- We present `SONATA`, a self-adaptive multi-objective optimization framework for HW-aware-NAS that progressively learns the importance of NN design parameters and guides the search accordingly.
- We implement self-adaptive evolution search operators guided by ML-based models to continuously learn the importance of NN design parameters from the search data history to explore effective neighborhoods of NNs by mutating and crossover the most important and critical design parameters.
- We compare `SONATA` against baseline evolutionary optimization algorithms for HW-aware-NAS frameworks.
- We validate our approach on many HW-aware NAS problem instances by varying the search spaces and hardware devices on the large ImageNet-1k dataset for image classification.
- Through an extensive set of experiments, we demonstrate that our approach can find optimal NNs under a low optimization budget. Evaluation results have shown the merit of `SONATA` with an accuracy improvement up to $\sim$**0.25%** on the ImageNet-1k dataset [114] and latency/energy gains up to $\sim$**2.42x** on edge GPUs, compared to SOTA ENAS, NSGA-II, under the same optimization budget (i.e., number of evolutionary iterations).

7.4 Design Parameters Importance Estimation for NAS

Quantifying the importance of NN design parameters is crucial to pave the way for more explainability of the causality between NN design perturbations and performance variations. Within this scope, design parameters can be ranked according to their importance a priori to help the designer know the most critical parameters and what to keep, tune, or remove [153]. Alternatively, the importance of the NN parameter can also be leveraged during or after the NAS to get insights into how NN architectures evolve to satisfy the optimization objectives. However, the drawn conclusions highly depend on the search space coverage and the search algorithm's effectiveness. The impact of design parameters on performance can

be evaluated by systematically varying the values of individual design parameters while keeping others fixed and observing the resulting performance changes to assess how NNs are sensitive to different design parameters. Sensitivity analysis aims to assess each parameter's significance by analyzing the variability in performance changes that can be attributed to each design parameter. It can be conducted via functional ANOVA (fANOVA) [95, 225], Local Parameter Importance (LPI) [17, 16], or by fitting surrogate models such as Random Forest [265]. Nevertheless, these techniques are only used for one objective (i.e., accuracy) and never adapted for a HW-aware NAS multi-objective context. Thus, one must ask the following question: *"How existing parameter importance analysis methods can be adapted in a multi-objective context to understand the interplay between NN design parameters and their impact on the overall performance regarding the accuracy and hardware efficiency ?"*

7.5 Problem Statement

7.5.1 The Main Problem: HW-aware NAS

Let $\mathcal{M}$ be a NN architecture that comprises n sequentially arranged computing blocks $\mathcal{B}^n$, each encompasses d_n computing layer $\mathcal{L}^{d_n}$ (See Figure 7.4). Typically, the type, order, and inter-dependency of $\mathcal{B}^n$ define the macro-architecture of the NN, whereas the specifications of $\mathcal{L}^{d_n}$ define micro-architecture of the NN. To simplify the design, existing search spaces embedded into supernets [31, 219, 75] assume a fixed macro-architecture while evolving the NN design around the variations of the $\mathcal{B}^n$ micro-architecture. The micro-architecture of the block defines the internal layers components (e.g., MBConv [186], Attention [57]) and their respective design parameters (e.g., depth, kernel size, attention heads). We refer to this micro-architecture search space by $\mathbb{M}$.

$$\mathcal{M}(\cdot) = \mathcal{B}^n \circ \mathcal{B}^{n-1} \circ \mathcal{B}^{n-2} \circ \cdots \circ \mathcal{B}^2 \circ \mathcal{B}^1 s.t.\ \ \mathcal{M} \in \mathbb{M} \tag{7.1}$$

$$\text{Where} \quad \mathcal{B}^j = \mathcal{L}^{d_j} \circ \mathcal{L}^{d_j-1} \circ \cdots \circ \mathcal{L}^2 \circ \mathcal{L}^1 \tag{7.2}$$

A typical Hardware-aware NAS problem is formulated as a multi-objective optimization problem in which the aim is to automate the exploration of the search space $\mathbb{M}$, to retain NN architectures $\mathcal{M}^*$ that offer minimal prediction error (Err), short execution latency (Lat), and low energy consumption ($Ergy$):

$$\mathcal{M}^* = \underset{\mathcal{M} \in \mathbb{M}}{\arg\min}[Err(\mathcal{M}), Lat(\mathcal{M}, \mathcal{H}), Ergy(\mathcal{M}, \mathcal{H})] \tag{7.3}$$

Where Err, Lat, $Ergy$ designate the optimization objectives, and $\mathcal{H}$ is the Hardware configuration (e.g., edge devices) for deployment. Within this global optimization problem, we substitute and formulate another sub-problem for learning and estimating NN design parameters' importance given a sampled set of NN architectures $\mathcal{M}$ and their objective evaluations. This sub-problem is detailed in the following.

7.5.2 The Sub-Problem: Design Parameter Importance Learning

Let ($\mathbb{X}^T$, $\mathbb{Y}^T$) be a history set of sampled NN architectures ($\mathbb{X}^T$) and their evaluations on the underlying optimization objectives ($\mathbb{Y}^T$), respectively, during T generations of the ENAS. Let $\mathcal{E}$ be the encoding vector of $\mathcal{M}$ where:

$$\mathcal{E} = [\pi_1, \pi_2, \cdots, \pi_m], \quad s.t. \;\; \pi \in \{\mathbb{R}, \mathbb{K}, \mathbb{E}, \mathbb{W}, \mathbb{D}\} \tag{7.4}$$

Here π defines a design parameter of the $\mathcal{M}$ blocks micro-architecture that can designate an input resolution ($\mathbb{R}$), kernel size ($\mathbb{K}$), channel expand ration (i.e., channel width) ($\mathbb{E}$), block width ($\mathbb{W}$), or block depth ($\mathbb{D}$). A design parameter is included in the encoding vector *if on only if* it's as design variable of the search space $\mathbb{M}$. Given the multi-objective context of the HW-aware NAS, we aspire to learn the correlation between the settings of the NN design parameters with regard to the obtained Pareto optimality and diversity scores. In other words, we aim to discover the most influential design parameters on the Pareto front optimality and diversity. This learning process can leverage the history of the ENAS data ($\mathbb{X}^T$, $\mathbb{Y}^T$) to train a surrogate model capable of estimating NN *design parameters importance*. This model can be updated after each G generation – For instance, after each two (02) generations, we re-train the model with new sampled NN architectures. Thus, G designates the update rate. The surrogate model for NN design importance learning is noted by $\theta_K^{\mathbb{M},\mathcal{H}}$ and is detailed as follows:

$$\theta_G^{\mathbb{M},\mathcal{H}} : \mathcal{E} \rightarrow \mathcal{S} \;,\; \mathcal{E}_i = encoding(x_i) \mid x_i \in \mathbb{X}^T \subset \mathbb{M} \tag{7.5}$$

$$\text{where} \;\; \mathcal{S}_i = Optimality(y_i)^{\beta_1} + Diversity_(y_i)^{\beta_2} \tag{7.6}$$

$$s.t. \; y_i \in \mathbb{Y}^T \mid y_i = [Err(x_i), Lat(x_i, \mathcal{H}), Ergy(x_i, \mathcal{H})] \tag{7.7}$$

Here $\theta_G^{\mathbb{M},\mathcal{H}}$ defines a mapping function between the encoding vectors $\mathcal{E}_i$ of NN and a weighted sum of their *Optimality* and *Diversity* scores $\mathcal{S}_i$. For the sake of generality, β_1 and β_1 can be used as control knobs to favor optimality over diversity or vice-versa. In our case, as we're equally interested in both optimality and diversity, we set the them as $\beta_1 = \beta_2 = 0.5$.

We quantify the *Optimality* using Pareto ranking scores as shown in equation (7.8). The *Pareto_rank* score is obtained from calculating the minimal Euclidean distance between each objective vector $y_i \in \mathbb{Y}^T$ and each optimal (i.e., non-dominated) objective vector $z_i^* \in \mathcal{PF}_{Ref}(\mathbb{Y}^T)$ from a reference Pareto front $\mathcal{PF}_{Ref}$. We compute the reference Pareto front using the non-dominated sorting algorithm [52] on the objective vectors from T evaluated population ($\mathbb{X}^T$, $\mathbb{Y}^T$). We note that K in equation (7.8) refers to the number of optimization objectives. In our use-case, $K = 3$ and we consider 3 objectives: Error in the classification (Err), Latency (Lat), and Energy consumption ($Ergy$).

$$Pareto\_rank(y_i) = \min_{z_i \in \mathcal{PF}_{Ref}} \sqrt{\sum_{k=1}^{K} (y_i^k - z_i^k)^2} \tag{7.8}$$

For interpretation, NN architectures with the lowest Pareto rank scores are closer to the optimal reference Pareto front $\mathcal{PF}_{Ref}$, indicating better convergence. Thus, the lowest the Pareto rank score is, the more optimal the NN architectures explored

by the ENAS are.

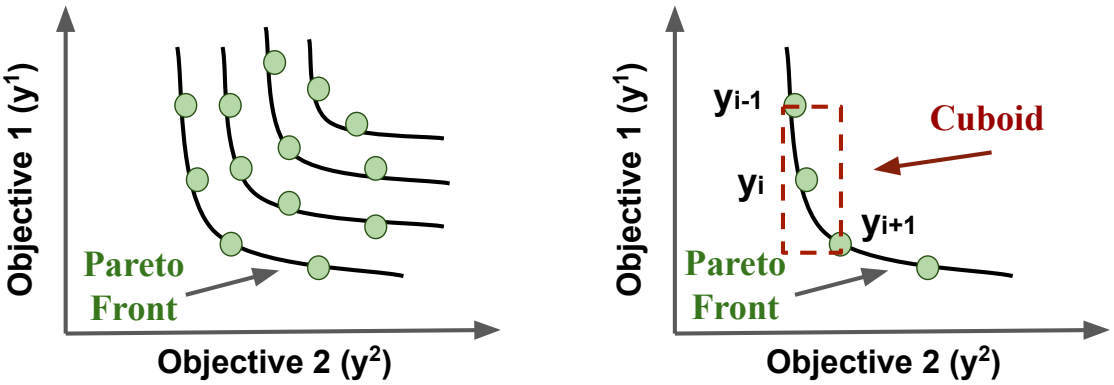


Figure 7.2 Calculation of the crowding distance. To simplify the interpretability, we give an example of two optimization objectives, y^1 and y^2. However, the same technique can be applied to the case of more than two optimization objectives.

We measure the *Diversity* by employing the crowding distance metric introduced in NSGA-II [51] to characterize the disparity of NN architectures in the objective space (i.e., based on the disparity of their objectives vectors). The mathematical formulation of the *crow_distance* is detailed in equation (7.9):

$$crow\_distance(y_i) = \sum_{k=1}^{K} \frac{y^k_{(i+1)} - y^k_{(i-1)}}{y^k_{max} - y^k_{min}} \tag{7.9}$$

Where K is the number of objectives, $y_{(i+1),k}$ and $y_{(i-1),k}$ are the values of the k-th objective for the objective vector immediately succeeding and preceding the i-th objective vector, respectively.

y^k_{min} and y^k_{max} are the minimum and maximum values of the k-th objective. The crowding distance of the i-*th* objective vector y_i expresses the average side-length of the cuboid as illustrated in Figure 7.2.

On comparing two NN architectures with different crowding distances, the NN architecture with the large crowding distance is considered present in a less crowded region, thereby being more different. Thus, the higher the crowding distances are, the more diverse the NN architectures explored by the ENAS are.

How NN design parameters importance can be inferred from the mapping function θ between the encoding vectors the NN and the weighted sum of their optimality and diversity scores ?

We aim to estimate the importance of NN design variables π by *analyzing* the learned mapping function θ. In other words, we aspire to measure the impact of the NN design variables π on the performance scores by observing which design variables are critical to better fit θ on the search history data $(\mathbb{X}^T, \mathbb{Y}^T)$.

Thus, we assume that the design variables π with the high variance in the predictions provided by θ are the most critical for the *Optimality* and *Diversity* scores of NNs.

The analysis step of the mapping function $\theta^{\mathbb{M},\mathcal{H}}_G$ can be performed using methods such as variance analysis [95, 225] or by quantifying the information gain of prediction features coverage in tree-based ML methods [39, 130]. For instance,

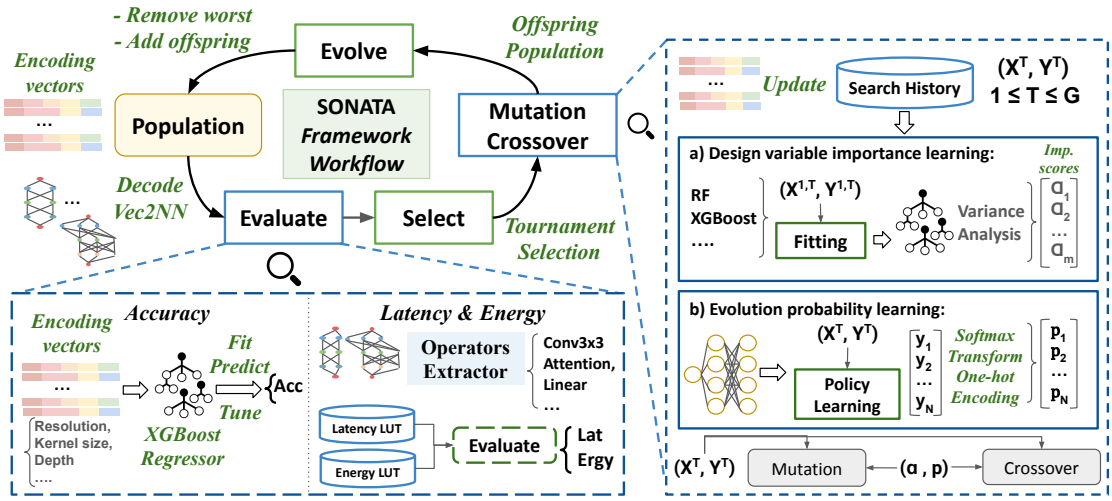


Figure 7.3 SONATA: self-adaptive and data-driven evolutionary search process.

if the mapping function is a tree-based ML model (e.g., Random Forest [130]), the importance of design variables can be measured by the number of times each design variable has been used to split and construct nodes in the decision tree (See Figure 7.5). Alternatively, in variance analysis with fANOVA [95, 225], the importance of design variables can be expressed as the degree of perturbations observed in the outputs of the mapping function when varying one specific design variable while fixing the others at specific values (e.g., at their minimal values).

Once the learned mapping function is analyzed, the estimated importance scores of the design parameters will be used to rank them from the least to the most important. Logically, exploring the most important NN design parameters increases the potentiality of retrieving NN architecture candidates with better spread and variance in the *Optimality* and *Diversity* scores. Accordingly, in the context of ENAS, mutation and crossover should be intensified and focused on the most critical design parameters. In a nutshell, our ultimate goal is to decide – in a data-driven manner– which NN design parameters are worth to *explore* and *exploit* in the ENAS using evolution operators (i.e., mutation and crossover).

7.6 Proposed Approach

We propose, SONATA, a novel self-adaptive evolutionary algorithm for Hardware-aware Neural Architecture Search. SONATA leverages the data generated during the search to learn, adjust, and use ML-based models to direct the search towards Pareto optimal NN designs while sustaining minimal search overhead. The key idea of our framework is to fully reuse generated data and evaluations from the search process to understand how design variables impact the Pareto optimality and diversity scores. In a data-driven fashion, SONATA aims to build knowledge on how the search process should evolve given the multi-objective context of HW-aware NAS. Figure 7.3 gives an overview of our framework. We build the entire search process upon the existing NSGA-II algorithm for multi-objective optimization by reusing fundamental components such as population initialization and tournament selection. Our contributions entail leveraging ML-based data-driven methods to enhance two critical components:

— (*i*) *Mutation and Crossover*: by using ML-based methods to estimate the most impacting design variables that would result in diverse and optimal NNs. Moreover, we use an ML-based controller that learns -in an unsupervised fashion- the probabilities of mutation and crossover for each individual in the selected subset $\mathbb{X}'^{T}$.
— (*ii*) *Evaluation*: by employing cheap performance estimation strategies such as ML-based surrogate models to evaluate the accuracy of the sampled NN and lookup tables to compute the latency and energy consumption on the target hardware device. These proxy performance estimators help accelerate the fitness evaluation process, which is the bottleneck in HW-aware NAS.

The overhead of training and updating the ML-based models incorporated in the evaluation and mutation/crossover is minimal and does not compromise the overall search time. Furthermore, our approach contributes self-adaptive search operators upon the existing NSGA-II and can be compatible with any search space, hardware devices, or target task and dataset.

7.6.1 Search Space Encoding and Initialization

In ENAS, every NN architecture is represented as a *genome* corresponding to an *encoding vector* that embeds the neural design specification $\mathcal{E} = [\pi_1, \pi_2, \cdots, \pi_m]$. One gene of the genome corresponds to one of the design variables π of the encoding vector. We employ a direct discrete encoding as depicted in Figure 7.4 to represent a single NN. The same genome representation is used as feature vectors to train the surrogate models at the mutation/crossover and for fitness evaluation. We note that the dimension m of the encoding vector depends on the search space defined in $\mathbb{M}$. In this paper, we study the case of *micro-search space* in which the number of neural blocks, noted n in equation (7.1), for $\mathcal{M}$ is fixed, whereas the block components and operator in each layer L^{d_j}, in equation (7.2) of the n B^j blocks are searchable.

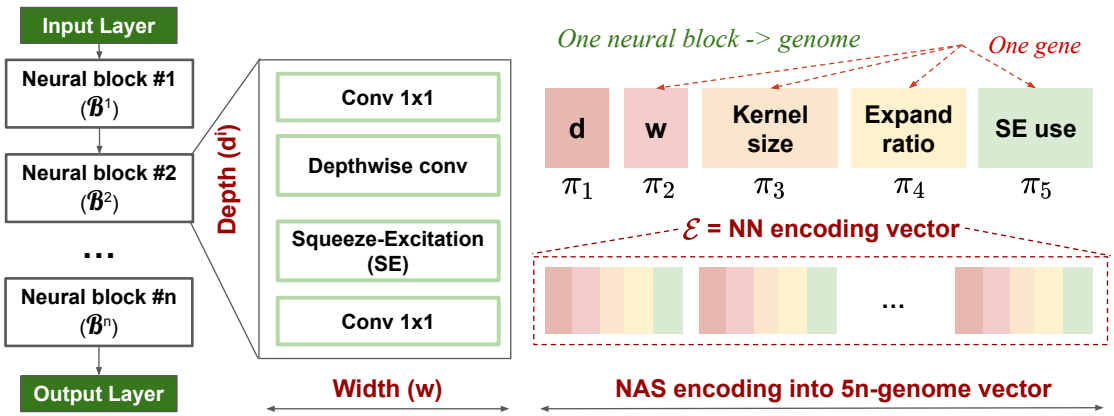


Figure 7.4 SONATA Neural network (NN) encoding scheme.

The initialization method in ENAS is essential to ensure diversity within the first population. To achieve the said purpose, we employ the *Latin Hypercube Sampling (LHS)*, a statistical technique for efficiently sampling multi-dimensional spaces. It involves dividing each parameter's range into equally probable intervals and randomly selecting a single value from each interval for π. These values are then combined to form a sampled genome $\mathcal{E}$.

7.6.2 Self-adaptive Mutation and Crossover

Following the evolutionary paradigm, once a population is evaluated, a selection step follows up to render a subset of promising genomes that will evolve to produce the next offspring population. This evolution step is ensured through mutation and crossover. Typically, Conventional mutation and crossover operators are applied to random positions of the genomes. However, this randomness often leads to a significant lack of directional focus, which becomes particularly costly when dealing with expensive evaluation functions, as observed in HW-aware NAS problems. Acknowledging this challenge, we design learning-based mutation and crossover operators that target only the most pivotal decision variables. The rationale behind this approach is that by focusing perturbations on the most influential design variables (i.e., genes) within the genomes, we aim to induce high variability in the objective outcomes, thereby maximizing the rates of exploration and exploitation in ENAS.

To achieve the said purpose, we leverage the search data history to identify the design variables that are most crucial to Pareto optimality and diversity. This process is illustrated in Figure 7.3:

— In **step (a)**, ML-based models are trained to learn and map importance scores to each design variable as formulated in 7.5.2.

— In **step (b)**, we employ a learning-based policy that determines each genome's mutation and crossover probabilities.

Thus, the first step addresses the question of '**How ?**' in terms of pinpointing which design variables to focus on, while the second phase tackles the '**Where ?**' by determining which genomes are promising for applying mutation and crossover.

① Design Parameter Importance Learning

We use the history of search data ($\mathbb{X}^T$, $\mathbb{Y}^T$) to train, tune, and update a gradient-based boosting tree [39] surrogate model to fit the mapping function detailed in equation (7.5). Each design parameter in the encoding vector $\mathcal{E}$ in equation (7.4) is considered as a potential feature for the node and leaf splitting during the learning and generation of the decision trees. We employ XGBoost to learn generate the tree-based model $\theta_K^{\mathbb{M},\mathcal{H}}$. The learning process follows a gradient-descent optimization where the loss function to minimize is given as following:

$$\Theta(\mathcal{S},\hat{\mathcal{S}}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\mathcal{S}_i - \hat{\mathcal{S}}_i))^2} + \sum_{k=1}^{K}\mathcal{L}_2(f_k) \;\text{ where }\; \Omega(f_k) = \gamma T + \frac{1}{2}\lambda\|\mathbf{w}\|^2 \quad (7.10)$$

Where n is the number of ground-truth labels and K is the maximum number of decision trees, Ω is the L_2 regularization term used to control over-fitting by penalizing the complexity of the model. Specifically, Ω is defined for each tree f_k. For each tree f_k, T is the number of leaves, w is the vector of scores in the leaf nodes, γ and λ are regularization parameters. We note that a node or a leaf is associated with a unique feature, i.e., decision variable $\pi \in \mathcal{E}$. Once the learning process completes its procedure, a vector of importance scores $\alpha = [\alpha_1, \alpha_1, \ldots, \alpha_m]$ is generated. Each α_i designates an importance score of a design variable $\pi \in \mathcal{E}$. The importance score is computed based on the number of times a leaf or node, in decision trees, was split based on π (See Figure 7.5).

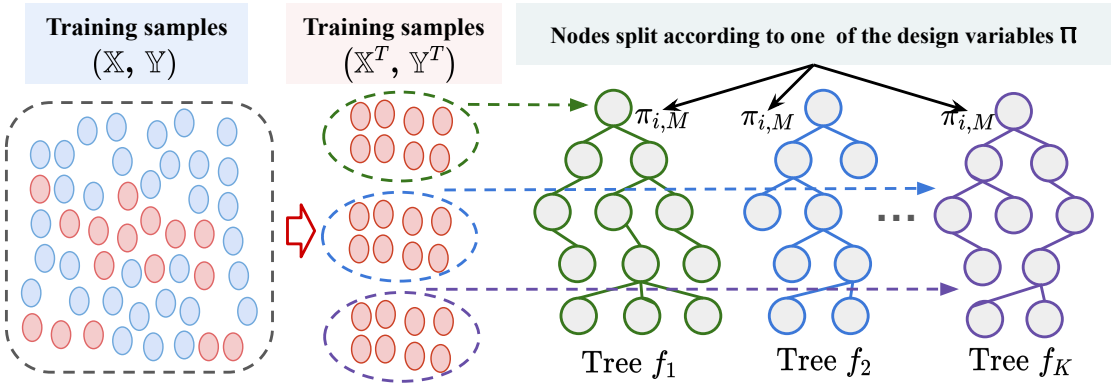


Figure 7.5 Nodes split mechanism in tree-based model (XGBoost).

② Evolution Probability Learning

Learning the importance of the NN design parameter helps pinpoint which genes are worth exploring during the ENAS process. However, another source of uncertainty comes from the probability at which evolution operators should be applied. Dynamic probability sampling for mutation and crossover has been shown promising for ENAS [238].

In the same spirit, we propose training a network *Policy learning* (See Figure 7.3) to learn the probabilities of mutation/crossover per genome. We formulate probability learning as a sequential decision-making process, where an agent, neural networks denoted as ψ, can make a sequence of decisions about selecting appropriate evolution probabilities $p = [p_1, p_2, \ldots, p_N]$ according to the gain in Pareto scores in $\mathcal{S}$. We design the network agent ψ with three (03) full-connected (FC) layer that takes as inputs the encoding vectors (π^*) of selected genomes for mutation/crossover after the tournament selection.

The 03 FC layers perform feature extraction to select appropriate evolution probabilities for each input π^*. To enhance the agent network, we use a reward function, $\mathcal{R}_\psi$, in Reinforcement Learning manner [271], as follows:

$$\mathcal{R}_\psi = \frac{1}{N}\sum_{i=1}^{N}\Big(-\big(\mathcal{S}_i - \max(\mathcal{S}_{(1,\ldots,N)})\big) \times \log(\text{soft\_prob}_i)\Big) \tag{7.11}$$

where $\mathcal{S}$ is the Pareto scores, sof_prob are the softmax probabilities predicted by the agent network ψ for the given inputs of encoding vectors. We employ the policy gradient algorithm proposed by [9] to optimize our agent network ψ.

7.6.3 Surrogate-assisted Fitness Evaluation

To accelerate the fitness evaluation, we leverage surrogate [1] models based on XGBoost [39] ML-based methods for regression. As discussed in Chapter 6, XGBoost has shown impressive prediction accuracy, generalization power, and rank-preserving capabilities. Prior works in surrogate modeling, such as NAS-Bench-301

1. Surrogate models and Prediction models can be used interchangeably to designate performance estimation models. The terminology of '*Surrogate models*' is widely used in the context of optimization and HW-aware NAS.

[198], share similar findings. In our framework SONATA, we employ XGBoost to model the accuracy of NNs. Prior to the search process, we randomly sampled and evaluated 500 NNs on the validation dataset. Then, we use the sampled data to train prediction models for accuracy. The architectural specifications of the sampled NNs (e.g., input resolution, depth, width, kernel size) are used as feature vectors for the surrogate model training process.

For hardware-related metrics, we use lookup tables (LUT) by inspecting and analyzing the commonly used operators (e.g., convolution, attention, fully connected) across the studied search spaces in this work. We then deploy and profile the operators on the target hardware devices to get latency and energy consumption measurements. The reason for employing lookup is that recent and effective search spaces in [31, 33, 219, 75] share common and similar neural operators and configurations. Thus, retrieving these common operations and measuring their hardware overheads once and for all is more convenient to reuse them across search spaces. However, we recommend using the model-level prediction models detailed in Chapter 6 for unstructured search spaces that exhibit much less similar and common neural operators.

7.7 Experiments and Evaluation

In this section, we conduct experiments to showcase the merit of SONATA to render optimal NN architectures within a low optimization budget. We evaluate the generality of our framework across NAS search spaces and target hardware devices. Specifically, we experiment on SOTA CNN and Transformer search spaces for image classification on the large ImageNet-1k dataset [114]. on three different edge GPUs from NVIDIA. Following, we provide more details on the experiment setup and evaluation results.

7.7.1 Experimental Setup

① NAS Search Spaces

We study the case of four (04) SOTA search spaces, originally designed for edge computing systems. Specifically we built our NAS search spaces upon the baselines introduced in AlphaNet [219], Once-for-all (OFA) [30], ProxylessNAS [33], and NASViT [75]. The three first ones are based on CNN architectures. Notably, OFA and ProxylessNAS are built upon MobileNet-V3 [89], AlphaNet upon FBNet [229], while NASViT is a CNN-Transformer hybrid architecture combining MB-Conv blocks from [90, 187] and Transformer blocks from [57]. Additional details on search spaces are given in Tables 7.2, 7.3, and 7.1.

Table 7.1 Details on OFA [30] and ProxylessNAS [33] search spaces

Block name	Channel width	Depth	Kernel size	Expansion ratio	SE	Stride
Conv	16	-	3	-	-	2
MBConv-1	16	1	3	1	N	1
MBConv-2	24	{2, 3, 4}	{3, 5, 7}	{3, 4, 6}	N	1
MBConv-3	40	{2, 3, 4}	{3, 5, 7}	{3, 4, 6}	N	2
MBConv-4	80	{2, 3, 4}	{3, 5, 7}	{3, 4, 6}	Y	2
MBConv-5	112	{2, 3, 4}	{3, 5, 7}	{3, 4, 6}	N	2
MBConv-6	160	{2, 3, 4}	{3, 5, 7}	{3, 4, 6}	Y	1
MBConv-7	960	{2, 3, 4}	{3, 5, 7}	{3, 4, 6}	Y	2
MBPool	1280	-	1	6	-	-
Input resolution	{128,.., 224}					

Table 7.2 Details on the AlphaNet [219] search space

Block name	Channel width	Depth	Kernel size	Expansion ratio	SE	Stride
Conv	{16, 24}	-	3	-	-	2
MBConv-1	{16, 24}	{1,2}	{3, 5}	1	N	1
MBConv-2	{24, 32}	{3, 4, 5}	{3, 5}	{4, 5, 6}	N	2
MBConv-3	{32, 40}	{3, 4, 5, 6}	{3, 5}	{4, 5, 6}	Y	2
MBConv-4	{64, 72}	{3, 4, 5, 6}	{3, 5}	{4, 5, 6}	N	2
MBConv-5	{112,128}	{3, 4, 5, 6, 7, 8}	{3, 5}	{4, 5, 6}	Y	2
MBConv-6	{192, 200, 208, 216}	{3, 4, 5, 6, 7, 8}	{3, 5}	6	Y	1
MBConv-7	{216, 224}	{1, 2}	{3, 5}	6	Y	2
MBPool	{1792, 1984}	-	1	6	-	-
Input resolution	{192, 224, 256, 288}					

Table 7.3 Details on the NASViT [75] search space

Block name	Channel width	Depth	Kernel size	Expansion ratio	SE	Stride	Attention windows
Conv	{16, 24}	-	3	-	-	2	-
MBConv-1	{16, 24}	{1,2}	{3, 5}	1	N	1	-
MBConv-2	{24, 32}	{3, 4, 5}	{3, 5}	{4, 5, 6}	N	2	-
MBConv-3	{32, 40}	{3, 4, 5, 6}	{3, 5}	{4, 5, 6}	Y	2	-
Transformer-4	{64, 72}	{3, 4, 5, 6}	-	{1, 2}	-	2	1
Transformer-5	{112,120,128}	{3, 4, 5, 6, 7, 8}	-	{1, 2}	-	2	1
Transformer-6	{160, 168, 176, 184}	{3, 4, 5, 6, 7, 8}	-	{1, 2}	-	1	1
Transformer-7	{208, 216, 224}	{3, 4, 5, 6}	-	{1, 2}	-	2	1
MBPool	{1792, 1984}	-	1	6	-	-	-
Input resolution	{192, 224, 256, 288}						

② Hardware Devices Settings

We validate our method on three (03) edge devices from NVIDIA:

1. Jetson AGX Xavier equipped with an NVIDIA Carmel Arm-64bit CPU and a Volta GPU of 512 GPU cores.
2. Jetson TX2 composed of an NVIDIA Denver 64Bit and ARM-A57 CPU

cores and a Pascal GPU with 256 GPU cores.

3. Jetson Nano that comprises an ARM-A57 CPU and a Maxwell GPU of 128 GPU cores.

The NNs have been implemented and evaluated using TensorRT 8.4 as high-performance SDK from NVIDIA [3]. We map the NN entirely on the GPU as our the target computing unit under FP32 data precision and `MAXN DVFS` setting.

③ Evolutionary Search Settings

We build `SONATA` on top of the existing NSGA-II algorithm [52]. We keep the main algorithmic components. The only changes are the fitness evaluation and evolution operators as discussed in Sections 7.6.2 and 7.6.3. We fix the population size to 300 and run the optimization framework for $G = 10$ generations (i.e., evolution cycle). We update the surrogate models after every two (02) generations. The network agent ψ selects the evolution probabilities within a predefined range of $[0.3, 1.0]$. We select the top five (05).

We use the native NSGA-II – without our modifications – as a baseline search algorithm for comparison in the following sections. To ensure a fair comparison, we use the same population initialization method (i.e., LSH), fitness evaluation strategy, elite selection, and optimization budget (i.e., population size and number of generations). We also tuned the mutation and crossover probabilities and found a combination of 0.5-0.5 as an optimal setting for this baseline.

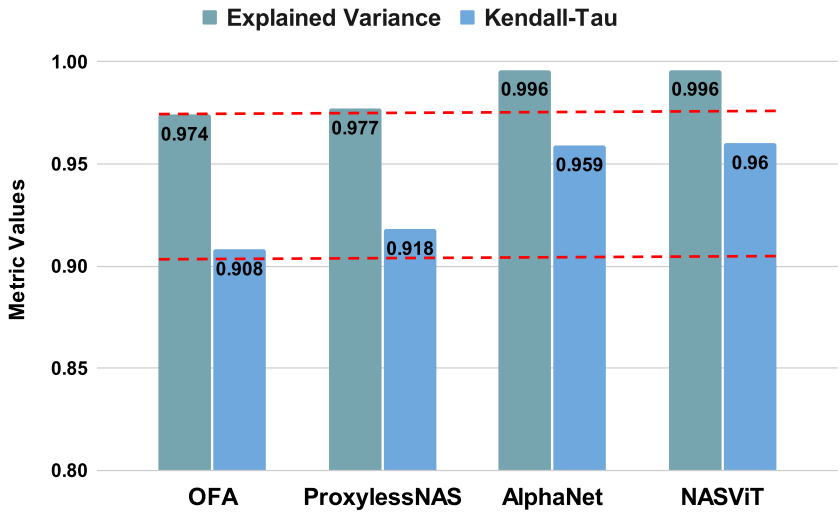


Figure 7.6 Performance of the surrogate models used to estimate the accuracy of NNs in `SONATA`. We report the explained variance and Kendall-Tau rank correlation coefficients – The higher values indicate optimal performances.

7.7.2 Surrogate Models Analysis

In this section, we discuss the performance of the surrogate models employed in our framework, notably:

— (i) The surrogate model used for NN accuracy estimation in the fitness evaluation step of `SONATA`.

— (ii) the surrogate model used for design variable importance learning and estimation in the Mutation/Crossover step of `SONATA`.

Firstly, we trained the XGBoost surrogate model to estimate the accuracy of the sampled NN. As shown in Figure 7.6, XGBoost gives relatively accurate estimations in all NAS search spaces. The explained variances, also known as R2, and Kendall-Tau rank correlation coefficient are above 0.80. This result indicates a strong prediction accuracy and rank preservation. Our preliminary results also have shown a prediction accuracy higher than 95% for search spaces, further stipulating the generalization and rank-preserving power of XGBoost [39]. We also note that the shared macro-architecture structure in SOTA NAS search spaces highly contributes to the performance of the surrogate models used for accuracy [198]. For unstructured search spaces with uneven NN macro-architecture, fitting a surrogate model to predict the accuracy can be challenging. Thus, employing a string predictor like XGBoost is highly recommended.

Secondly, in Figure 7.7 we report the performances of the surrogate models, Θ mathematically defined by equation (7.5). Θ is employed to learn and estimate the importance of NN design variables to guide the mutation/crossover in **Step (a)** (See Figure 7.3). We note that these surrogate models leverage information on NN accuracy and HW efficiency (i.e., latency and energy) to assign optimality and diversity scores to NNs as explained in Section 7.6.2. These scores depend on the NAS search space and the target HW device. The pairs (NN, HW) are indicated in the x-axis of the graph depicted in Figure 7.7.

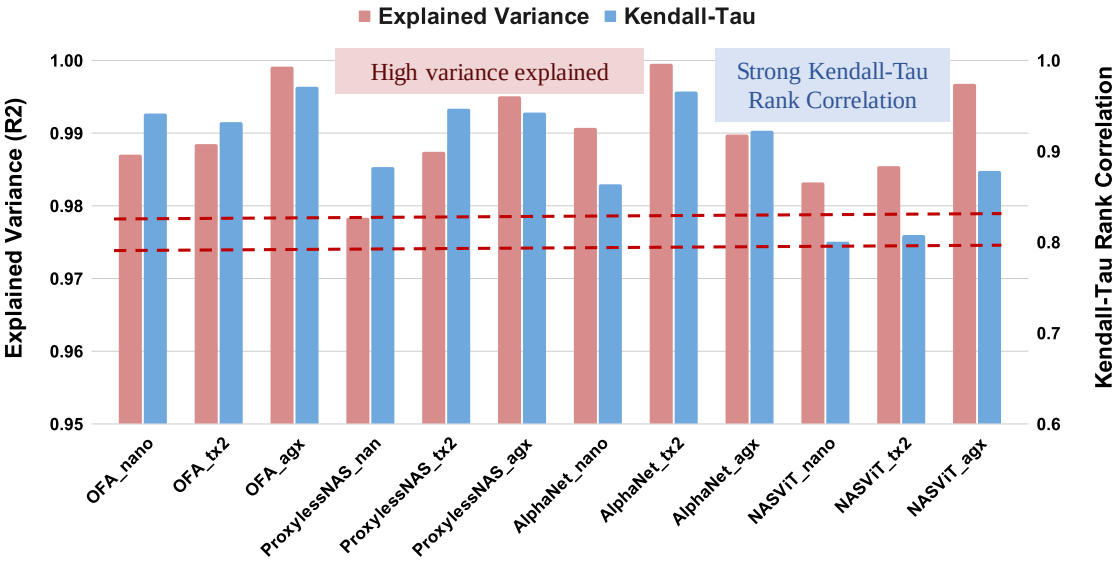


Figure 7.7 Performance of the surrogate models used to learn the importance of NN design parameters importance in `SONATA`. We report the explained variance and Kendall-Tau rank correlation coefficients – The higher values indicate optimal performances. s

As shown, the high values of the explained variances and Kendall-Tau rank correlation coefficients indicate strong performances and high fitness of Θ models. The Kendall-Tau correlation coefficients are slightly lower than those in Figure7.6. The reason behind this observation is the complexity of the correlation in the scores used to train Θ. This complexity arises from the contradictory nature of the involved objectives, notably NN accuracy and HW efficiency. Furthermore, we noticed that the diversity term in (7.6) added another layer of complexity. Nevertheless, we noticed that the results of `SONATA` can be less diverse without

including the diversity term in (7.6) – thus, we opted to keep it along the optimality term to obtain an optimal balance of both.

7.7.3 SONATA Optimization Efficiency

In multi-objective optimization problems, both convergence and diversity are of high priority and importance. To showcase the merit of our framework SONATA compared to baseline ENAS algorithms, such as NSGA-II, we employ different performance evaluation metrics widely used to assess optimality and diversity:

— (*i*) Hypervolume that measures the portion of the objective space dominated by the Pareto front obtained by the search algorithm.

— (*ii*) Inverted Generational Distance (IGD) that measures the average distance from each point in the optimal Pareto front to its nearest point in the Pareto front obtained by the search algorithm. As the optimal Pareto front is unknown for the NP-hard problems, we approximate it by merging the Pareto fronts of our SONATA and the baseline NSGA-II.

— (*ii*) Dominance ratio computes the ratio of non-dominated solutions from the obtained Pareto front by the search algorithm contributing to the optimal Pareto front – approximated as previously mentioned

Table 7.4 Comparison between the optimization efficiency (i.e., convergence and diversity) of the baseline static ENAS NSGA-II [52] and our self-adaptive SONATA.

HW Device	NAS Search Spaces	Static ENAS NSGA-II [52]			Our Adaptive SONATA		
		Hypervolume	IGD	Dominance Ratio	Hypervolume	IGD	Dominance Ratio
Jetson Nano	OFA [31]	1725509	1.484	0.327	**1733316**	**0.1305**	**0.826**
	ProxylessNAS [33]	1464862	1.072	0.071	**1469478**	**0.0233**	**1.0**
	AlphaNet [219]	**2336429**	1.592	**0.687**	2330159	**0.8175**	0.305
	NASViT [75]	1547510	3.040	0.447	**1548843**	**0.4109**	**0.853**
Jetson TX2	OFA [31]	1756847	2.299	0.399	**1757666**	**0.2964**	**0.866**
	ProxylessNAS [33]	1485138	2.291	0.170	**1495840**	**0.1373**	**0.936**
	AlphaNet [219]	**2479025**	0.978	**0.806**	2476035	**0.3215**	0.372
	NASViT [75]	1809648	6.435	0.417	**1817793**	**0.5333**	**0.809**
Jetson AGX	OFA [31]	1842773	0.282	0.415	**1845282**	**0.0965**	**0.980**
	ProxylessNAS [33]	1569035	0.850	0.043	**1572546**	**0.0027**	**1.0**
	AlphaNet [219]	**2558571**	0.658	**0.826**	2561133	**0.2776**	0.579
	NASViT [75]	2268507	3.854	0.524	**2270813**	**0.8185**	**0.709**

We summarize the results of this evaluation in Table 7.4. We note that high hypervolume and dominance ratio values indicate an optimal balance between convergence and diversity, whereas low IGD values stipulate better convergence. As reported in Table 7.4, our SONATA generally outperforms the baseline search algorithm NSGA-II in all optimization metrics. This is attributed to the ability of SONATA to focus on the most rewarding NN design variables while also expanding the scope of exploration by dynamically varying the mutation/crossover probabilities via the RL-based agent network ψ.

More interestingly, the baseline NSGA-II only outperforms SONATA in the AlphaNet [219] search space. However, by observing the exploration results in the

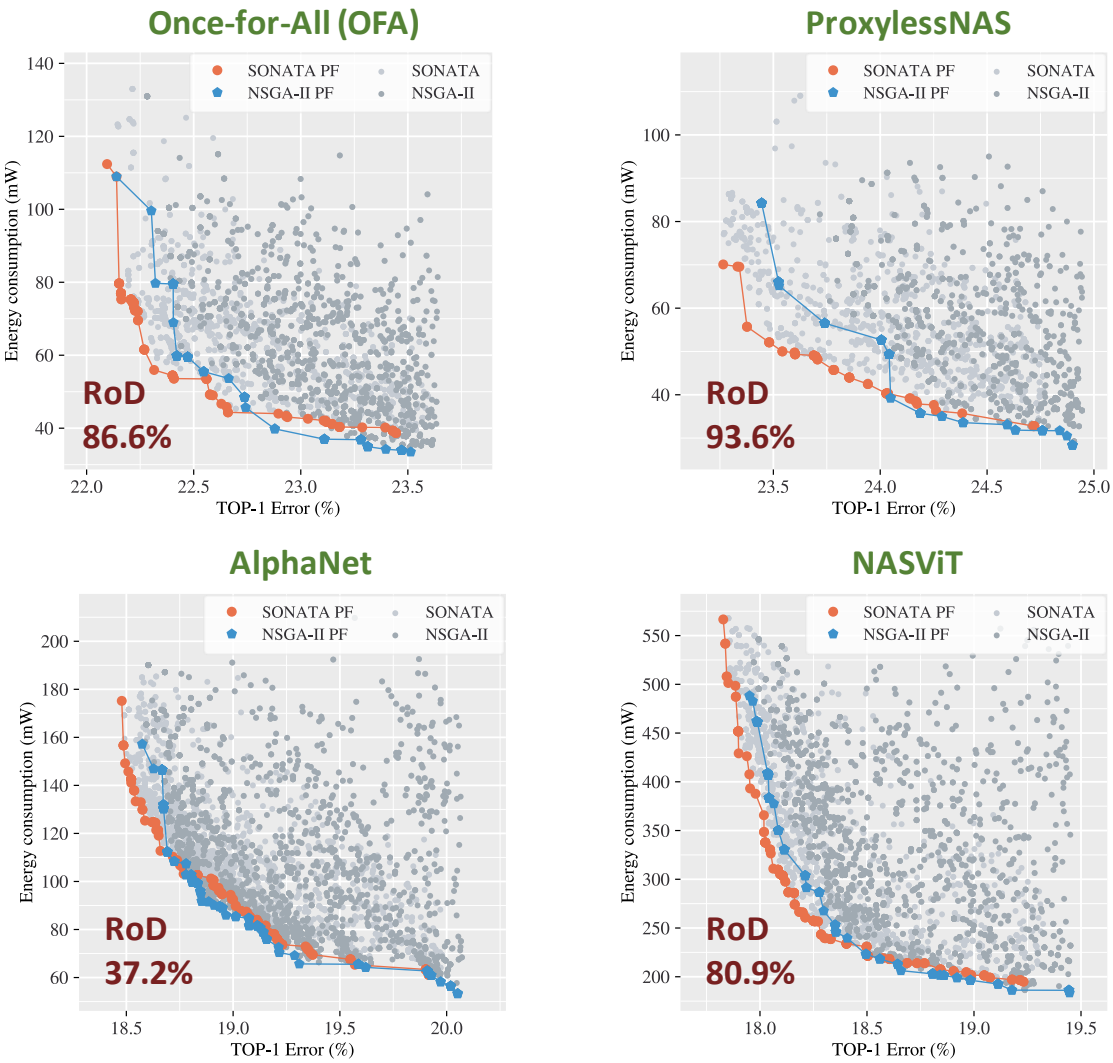


Figure 7.8 Comparing the optimization results of SONATA Vs. NSGA-II. The results are reported on all of the studied the search spaces on the NVIDIA Jetson TX2. The TOP-1 error are reported for the ImageNet-1k dataset.

plots of Figure 7.8, we can see that SONATA provides more accurate NNs with the same HW execution cost as NSGA-II. Thus, we highlight the importance of visual analysis when assessing the performance of optimization algorithms for HW-aware NAS. Quantitative and qualitative analysis must be jointly used to determine the effectiveness of search algorithms in HW-aware NAS. For instance, in the AlphaNet case, if NN accuracy is of the highest priority among other objectives, the designer/user may be more interested in the results of SONATA, as they provide more accurate NNs despite failing to outperform the baseline NSGA-II in the quantitative analysis of Table 7.4.

From the exploration results in Figure 7.8, we can also see how SONATA was able to explore diverse accuracy-efficiency trade-offs. Notably, by intensifying the search in the region of interest – the extreme left depicts the highest accuracy and lowest energy consumption. We can observe that SONATA optimization reaches NN designs that provide better accuracy and low energy consumption than those reached by the baseline NSGA-II. For instance, in the ProxylessNAS case, the Pareto front of SONATA provides an accuracy improvement by ∼**0.25%** and energy

gains of ∼**2.42x**. These observations further demonstrate the merit of adapting the evolution operators to intensify the search around the most important and promising NN design parameters.

7.8 Summary

In this chapter, we have explored the prospect of leveraging ML-based methods to enhance the efficiency of evolutionary NAS frameworks. Specifically, we built our motivation upon the assumption that not all NN design variables can substantially improve performance and hardware efficiency. We thus argue that ENAS search algorithms can focus on exploring the most rewarding design variables to accelerate the convergence and discovery of optimal NN design with better performance-efficiency tradeoffs.

Following this assumption, we proposed `SONATA`, a self-adaptive evolutionary framework for multi-objective HW-aware NAS frameworks. Our framework aims to reduce the randomness of conventional evolutionary operators (i.e., mutation and crossover) and replace them with learning-based operators. As such, we use the search data history and tree-based ML methods, such as XGBoost, to progressively learn the importance of NN design variables seeking optimality and diversity in the multi-objective context of HW-aware NAS. The learned importance variable scores are used to select a subset of the most important NN design variables on which mutation and crossover should be applied. Furthermore, we employ an RL-based agent that assigns dynamic mutation and crossover probabilities to evolve only the most promising NN architectures. We also employ ML-based and LUT surrogate models to accelerate the fitness evaluation step.

We evaluate `SONATA` on a plethora of HW-aware NAS problem instances that we formulate using:

— SOTA NAS search spaces from OFA [31], ProxylessNAS [33], AlphaNet [219] and NASViT [75]. These search spaces combine CNN and Transformers neural operators and are used for image classification on the ImageNet-1k [114].

— Edge hardware devices from the NVIDIA Jetson series, namely, Nano, TX2, and AGX Xavier, under the maximum clock frequency setting `MAXN`.

Evaluation results have shown the merit of `SONATA` with an accuracy improvement up to ∼**0.25%** and latency/energy gains up to ∼**2.42x**.

We have taken the first step towards a self-adaptive ENAS by leveraging ML-based methods to enhance evolutionary operators and fitness evaluation. However, data-driven ML-based methods can be extended to enhance other NAS components, notably:

— Search space by adding, altering, or removing NN design parameters based their importance prior [17] or during the search (This work).

— Search strategy via leveraging the dynamic reconfigurability of the search algorithm hyperparameters [119].

— Fitness evaluation by training surrogate predictors to estimate NN performances in an offline (as discussed in Chapter 6) or online [142] settings.

However, we believe that our work can be further extended to generalize the use of ML-based methods to enhance the different components of HW-aware NAS, in a multi-objective optimization context. As future works, we propose to extend our framework by enhancing the knowledge of the importance of NN design variables

with GNN or Transformers as introduced in [119]. Learning the distribution of NN design variables that improve performance and efficiency in a multi-objective context is also a promising direction for ENAS. Incorporating preference-based evolutionary operators to favor specific objectives and intensify the search around certain values without setting constraints or pruning the search spaces is an important step for the evolutionary HW-aware NAS framework.

Our framework can also be used offline to tune the search space and remove the least important design variables. It can also be used in an online setting, as we proposed, or by hierarchizing the exploration process for multi-level search spaces. It will be possible to co-optimize the NN and the HW by simultaneously setting their corresponding design parameters.

Chapitre 8

Conclusions, Outlooks, and Future Directions

8.1 Summary of the Thesis

The extraordinary success of Neural Networks (NN) can be attributed to their proficiency in processing data and extracting valuable patterns and features. This success is further bolstered by the availability of datasets and the variety of neural designs and operators. These advancements have expanded the NN's capability to represent intricate information and generalize across various tasks, including computer vision and natural language processing. Following this wave of success, NN has been widely adopted in Internet of Things (IoT) systems, fostering intelligent applications at the edge, closer to end-user devices and data sources. However, the resource limitations of edge devices in IoT systems — in terms of computational capability, memory, and energy budgets — necessitate the optimization of NNs for an effective deployment on such hardware platforms.

As mentioned in chapter 1, NN and hardware architectures continue to evolve at varying paces, and finding a definitive and deterministic answer to *How to optimize NN execution on every edge hardware device ?* remains an ongoing challenge. We stepped towards advancing research in this area by contributing from two perspectives. Firstly, by expanding the Hardware-aware Neural Architecture Search (HW-aware NAS) concept, we explored various optimization landscapes at both software and hardware levels. Secondly, we proposed novel methods to enhance performance evaluation and search strategy within HW-aware NAS by leveraging the '*Machine Learning for Machine Learning*' (ML4ML) paradigm. The overarching goal of this thesis is to address the following research questions:

— How to enlarge the typical NAS search space by integrating hardware properties, such as the *Dynamic Voltage and Clock Frequency Scaling* (`DVFS`) configurations, to realize more energy-efficient NNs ?
— How to further enhance the search space from the former question through leveraging Dynamic Neural networks (DyNNs) -via input-adaptive early-exit- and hardware properties -via `DVFS`- to enhance the energy efficiency and enable NN adaptability to the runtime environment ?
— How can we better incorporate the prospect of distributed computing and HW-aware NAS to optimize the execution of emerging Graph Neural Networks (GNNs) on heterogeneous MPSoCs for edge computing ?
— How can the dynamic usage of NN components and hardware computing units in heterogeneous MPSoCs contribute to a novel collaborative energy-efficient computing paradigm for DyNN inference ?
— How to adopt the '*ML4ML*' concept to enhance the performance evalua-

tion in HW-aware NAS by elaborating robust prediction models capable of rapidly estimating the latency, energy, and memory usage of NN ?

— How to extend the '*ML4ML*' concept to improve the search strategy in HW-aware NAS by progressively learning and evaluating the importance of NN design variables and guiding the search accordingly ?

In Chapter 2, we investigated the importance of exploring `NN` and `DVFS` configurations for edge GPUs. Our preliminary observation of the non-linear correlation between clock frequencies and latency/power consumption of the NN inference led us to enlarge the typical search space of NAS by considering (`DVFS`) configurations. We introduced DVFS-NAS, a co-optimization framework built upon the multi-objective evolutionary search algorithm NSGA-II, to explore the underlying joint search space effectively. Experimental results on the NVIDIA Jetson AGX Xavier have seen up to ∼1.53x energy efficiency compared to state-of-the-art methods while sustaining similar or better accuracy and latency. We also validated our framework on the high-performance SDK, TensorRT, in which results showed an execution speedup of up to ∼ 1.81x and a power saving of 61%. The output of our DVFS-NAS can be used to switch NN and/or DVFS configurations at runtime to accommodate various requirements of accuracy, latency, or power budget.

In Chapter 3, we extended our DVFS-NAS by investigating the dynamic scaling of neural components within a fixed NN. Switching NN models at runtime can be costly, so we explored the prospect of dynamically scaling NN components in an input-adaptive manner through an early exit strategy. Starting from our observation of the complexity of jointly exploring the design space of such NN and leveraging DVFS properties, we proposed HADAS, a novel and versatile HW-aware NAS for DyNNs with DVFS features. We implemented a hierarchical design space exploration for NN backbones, early-exit strategies, and DVFS configurations to find optimal DyNNs that balance performance and energy efficiency. We have shown that HADAS can achieve up to ∼1.57x energy efficiency while retaining better to similar accuracy levels than state-of-the-art NNs.

In Chapter 4, we explored the world of GNN as an important and complex class of NN that has lately seen a proliferation of its adoption on edge computing systems. We adapted HADAS's unique and hierarchical optimization strategy to serve the co-optimization of GNNs and their distributed workload mapping on the heterogeneous CUs of MPSoCs. We proposed MaGNAS, a mapping-aware co-optimization framework that characterizes the GNN architectural design space and distributed mapping options of workloads to make the best out of both SW and HW worlds. MaGNAS aims to enable the discovery of GNN designs optimized for distributed deployment on MPSoCs. MaGNAS employs a two-tier evolutionary search framework to identify optimal *architecture* and *mapping* pairings that provide the best performance trade-offs. Evaluation results have shown a ∼1.57× latency speedup, ∼3.38× energy efficiency for several image classification datasets executed on the Xavier MPSoC vs. the GPU-only deployment while sustaining an average 0.11% accuracy reduction from the baseline GNN models [81].

In Chapter 5, we investigate the perspective of leveraging both the efficiency of DyNNs, as seen in Chapter 3, and the effectiveness of distributed computing on MPSoCs, as demonstrated in Chapter 4. Accordingly, we introduced a new computing paradigm that best uses the two previous optimization techniques.

We proposed *Map-and-Conquer*, a novel energy-efficient execution paradigm for DyNN on MPSoCs. Our approach aims to identify an optimal way to split a DyNN across its 'width' dimension, enabling the parallel execution of multiple derived dynamic inference stages on different computing units. We validated our framework using Transformers and Convolutional Neural Networks (CNNs) on the CIFAR-100 dataset and employed NVIDIA's Jetson AGX Xavier MPSoC. Our findings have shown up to ∼2.1x more energy efficiency than GPU-only setup and up to ∼1.7x execution speedup than DLA-only setup.

In Chapter 6, we explored the concept of '*ML4ML*', elaborating how ML-based methods can streamline the HW-aware NAS. The major bottleneck in HW-aware NAS is the NN performance evaluation, involving an end-to-end pipeline from model loading to deployment and extensive performance measurements. We proposed performance prediction models to speed up this process. We presented a detailed performance analysis and characterization of NN inference workloads on edge GPU devices. We examined the correlation between NN features and various performance metrics. Consequently, we elaborated an end-to-end framework that leveraged different ML techniques to learn and model the underlying correlations. We validated our approach using three (03) edge GPUs from NVIDIA. Our proposed prediction models have shown an average prediction error of ∼11%, ∼6%, and ∼8% for latency, power, and memory usage, respectively.

In Chapter 7, we extended the prospect of *ML4ML* to enhance the efficiency of evolutionary HW-aware NAS search algorithms. These algorithms heavily rely on randomness without established reasoning on the importance of search space design parameters on the optimization objectives (i.e., accuracy and hardware efficiency metrics). We addressed this gap by proposing `SONATA`, a self-adaptive evolution for multi-objective HW-aware NAS frameworks. We designed self-adaptive evolution operators, notably mutation and crossover, guided by the importance of NN design variables. We leverage tree-based ML models to progressively learn the importance of design parameters from the history of the HW-aware NAS search. An extensive evaluation of multiple NAS search spaces and edge devices has shown that our approach improves upon the baseline with an accuracy improvement up to ∼0.25% and latency/energy gains up to ∼2.42x.

8.2 Outlook and Future Directions

> "One never notices what has been done; one can only see what remains to be done."
>
> — **Marie Curie**

This thesis brought several key insights and contributions to existing research on NN performance optimizations for Edge AI. The manifold aspects involving NN design, inference strategy (static vs. dynamic), and hardware configurations (DVFS and MPSoCs) make the optimization problem even harder. Furthermore, the multi-objective nature of the search algorithm – accuracy, execution latency, and energy consumption – puts the designer against a decisional dilemma whether

in terms of *What optimizations to investigate ?* or *What configurations to choose after the optimization process ?*.

Having explored part of these challenges, we have concluded that an *hierarchical and cooperative design spaces exploration* across multiple optimization landscapes and levels is mandatory to achieve optimal performance-efficiency tradeoff. With our scientific contributions, we aspired to advance the NN and HW optimization research one step further on the long path toward meeting the highest limits of execution speedup, energy efficiency, and performance. To continue this journey, we discuss in the following possible future research directions to push the boundaries of existing optimization methods:

8.2.1 Enrich the Search Space of NAS

Most existing NAS search spaces are human-biased and built upon fixed baselines, such as in [220, 219, 56]. Despite advancements in optimizing and speeding up NAS search strategy and performance evaluation, there remains a critical issue: The inherent limitations of optimizing within these predefined search spaces. The complexity intensifies when integrating additional dimensions, such as hardware characteristics or neural scaling methods, like early exits. A pivotal challenge for the research community is to develop a generalized NAS search space [134, 264]. This space should comprehensively encompass all possible NN architectures, enabling the application of other optimization techniques without the risk of suboptimality due to a restrictive dependency on the predefined search space.

8.2.2 Investigate Novel Hardware Technologies

As NNs continue to evolve, incorporating new neural architectures like Transformers and GNNs, they present varied computational and memory demands. Heterogeneous hardware architectures, such as MPSoCs, are increasingly outperforming traditional monolithic designs in this dynamic landscape. The reason is their superior adaptability to the diverse requirements of these emerging NN operations. In contrast, being optimized for specific applications, monolithic hardware struggles to keep up due to its inherent inflexibility towards novel applications and neural operators. A promising solution to this challenge is leveraging HW-aware NAS techniques on adaptable and more flexible hardware technologies, notably the chiplet based heterogeneous integration [154, 126]. As such, this approach enables the joint exploration of a broader spectrum of NN and HW options, facilitating the creation of highly efficient, tailor-made NN-HW pairings, thereby yielding significant gains in both efficiency and performance and contributing to the seamless integration of NN on edge systems.

8.2.3 Incorporate Advanced Dynamic Inference Strategy

Input-adaptive DyNNs represent a burgeoning field of study with significant potential, particularly in their adaptability for real-time applications like autonomous vehicles and robotics. This adaptability is crucial for promptly responding to new inputs. However, designing these networks involves exploring a vast design space, which includes finding the optimal configurations for static backbone structures, dynamic neural components, and deployment strategies. Consequently, a

well-defined search space encapsulating these design aspects is essential for elaborating the next generation of highly optimized DyNNs for edge computing systems [82]. An unexplored area in this research field is the potential of integrating hybrid dynamic inference strategies –such as early exit, computation skipping, and dynamic routing – each offering unique advantages. Additionally, the optimization of hardware-DyNN synergy remains an understudied area. Investigating this in conjunction with heterogeneous hardware architectures could significantly enhance the capabilities of current DyNN frameworks. We believe that extending our HW-aware dynamic NAS methods, such as in HADAS [21] and Map-and-Conquer [22], would help in accelerating investigating more design options of DyNNs on monolithic and heterogeneous hardware devices.

8.2.4 Towards Self-explainable HW-aware NAS

Existing multi-objective optimization frameworks for HW-aware NAS rely on randomness to explore a broad range of NN candidates. They lack reasoning, interpretability, and explainability before, during, and after the search. While this aspect logically occurs due to their black-box optimization nature, it can also be the reason behind their inability to converge rapidly towards the global optima [55]. The search in HW-aware NAS often needs many trial-error iterations due to exploring ill-designed NNs that add no gain to the optimization objectives. To alleviate this problem, HW-aware NAS methods must incorporate more explainability to guide the search process efficiently without wasting expensive optimization time and budget on exploring useless NN designs [233]. ML, RL, and Bayesian techniques allow learning the importance of design variables and varying them accordingly as proposed in [87, 193]. However, most existing methods are proposed in the mono-objective context, overlooking the multi-objective nature of HW-aware NAS. Thus, we believe that more work is needed to better understand the implications of design variables within the HW-aware NAS and how to employ such knowledge to accelerate the discovery of Pareto optimal NNs.

8.2.5 Generalize the HW-aware NAS to Multimodality AI

Multimodality AI is emerging as an advanced NN approach that enables the simultaneous processing of different types of input data, surpassing the capability of their unimodal counterparts. Recently, the outstanding capabilities of chatbots (e.g., GPT-4) are highly attributed to the multimodal learning paradigm. However, designing such networks is highly complex [249]. It involves searching optimal backbone networks for each modality and searching for an optimal fusion network to learn a joint embedding of information. The design landscape becomes more complex when integrating the hardware dimension as it constrains the number of modalities and fusion network capacity. Furthermore, HW-aware NAS frameworks for this type of NN are lacking in the literature. We have stepped towards bridging this gap by proposing our Harmonic-NAS [72, 96]. Still, the path toward studying, characterizing, and optimizing the performance of multimodal NN is long and challenging. A comprehensive design framework that accounts for hardware optimizations is needed for multimodal NNs.

Bibliographie

[1] Dimension ai. `https://app.dimensions.ai/discover/publication`. Accessed : 2023-10-01.

[2] Our world in data. `https://ourworldindata.org/grapher/artificial-intelligence-parameter-count`. Accessed : 2023-10-01.

[3] NVIDIA tensorrt. `https://developer.nvidia.com/tensorrt/`. Accessed : 2021-05-01.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow : A system for Large-Scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283. USENIX Association, Nov. 2016.

[5] M. S. Abdelfattah, Ł. Dudziak, T. Chau, R. Lee, H. Kim, and N. D. Lane. Best of both worlds : Automl codesign of a cnn and its hardware accelerator. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[6] H. Abdi. The kendall rank correlation coefficient. *Encyclopedia of Measurement and Statistics. Sage, Thousand Oaks, CA*, pages 508–510, 2007.

[7] M. Amarís, R. Y. de Camargo, M. Dyab, A. Goldman, and D. Trystram. A comparison of gpu execution time prediction using machine learning and analytical modeling. In *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, pages 326–333, 2016.

[8] Y. Arafa, A.-H. Badawy, G. Chennupati, A. Barai, N. Santhi, and S. Eidenbenz. *Fast, Accurate, and Scalable Memory Modeling of GPGPUs Using Reuse Profiles*. Association for Computing Machinery, New York, NY, USA, 2020.

[9] A. Ashok, N. Rhinehart, F. Beainy, and K. M. Kitani. N2n learning : Network to network compression via policy gradient reinforcement learning. In *International Conference on Learning Representations*, 2018.

[10] 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.

[11] M. Awad and R. Khanna. Support vector regression. In *Efficient learning machines*, pages 67–80. Springer, 2015.

[12] J. Benesty, J. Chen, Y. Huang, and I. Cohen. *Pearson Correlation Coefficient*, pages 1–4. Springer Berlin Heidelberg, 2009.

[13] H. Benmeziane, H. Bouzidi, H. Ouarnoughi, O. Ozturk, and S. Niar. Treasure what you have : Exploiting similarity in deep neural networks for efficient video processing. *arXiv preprint arXiv :2305.06492*, 2023.

[14] H. Benmeziane, K. E. Maghraoui, H. Ouarnoughi, S. Niar, M. Wistuba, and N. Wang. A comprehensive survey on hardware-aware neural architecture search, 2021.

[15] S. Bianco, R. Cadene, L. Celona, and P. Napoletano. Benchmark analysis of representative deep neural network architectures. *IEEE access*, 6 :64270–64277, 2018.

[16] A. Biedenkapp, M. Lindauer, K. Eggensperger, F. Hutter, C. Fawcett, and H. Hoos. Efficient parameter importance analysis via ablation with surrogates. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[17] A. Biedenkapp, J. Marben, M. Lindauer, and F. Hutter. Cave : Configuration assessment, visualization and evaluation. In *Learning and Intelligent Optimization : 12th International Conference, LION 12, Kalamata, Greece, June 10–15, 2018, Revised Selected Papers 12*, pages 115–130. Springer, 2019.

[18] T. Blickle. Tournament selection. *Evolutionary computation*, 1 :181–186, 2000.

[19] T. Bolukbasi, J. Wang, O. Dekel, and V. Saligrama. Adaptive neural networks for efficient inference. In *International Conference on Machine Learning*, pages 527–536. PMLR, 2017.

[20] H. Bouzidi, S. Niar, H. Ouarnoughi, and E.-G. Talbi. Sonata : Self-adaptive evolutionary framework for hardware-aware neural architecture search. *arXiv preprint arXiv :2402.13204*, 2024.

[21] H. Bouzidi, M. Odema, H. Ouarnoughi, M. A. Al Faruque, and S. Niar. Hadas : Hardware-aware dynamic neural architecture search for edge performance scaling. In *2023 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1–6, 2023.

[22] H. Bouzidi, M. Odema, H. Ouarnoughi, S. Niar, and M. A. Al Faruque. Map-and-conquer : Energy-efficient mapping of dynamic neural nets onto heterogeneous mpsocs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2023.

[23] H. Bouzidi, H. Ouarnoughi, S. Niar, and A. A. E. Cadi. Performance prediction for convolutional neural networks on edge gpus. In *Proceedings of the 18th ACM International Conference on Computing Frontiers*, CF '21, page 54–62, New York, NY, USA, 2021. Association for Computing Machinery.

[24] H. Bouzidi, H. Ouarnoughi, S. Niar, and A. A. E. Cadi. Performance modeling of computer vision-based cnn on edge gpus. *ACM Transactions on Embedded Computing Systems (TECS)*, 21(5) :1–33, 2022.

[25] H. Bouzidi, H. Ouarnoughi, S. Niar, E.-G. Talbi, and A. A. El Cadi. Co-optimization of dnn and hardware configurations on edge gpus. In *2022 25th Euromicro Conference on Digital System Design (DSD)*, pages 398–405. IEEE, 2022.

[26] H. Bouzidi, H. Ouarnoughi, E.-G. Talbi, A. A. El Cadi, and S. Niar. Evolutionary-based optimization of hardware configurations for dnn on edge gpus. In *META'21, The 8th International Conference on Metaheuristics and Nature Inspired Computing*, 2021.

[27] H. Bouzidi, H. Ouarnoughi, E.-G. Talbi, A. A. El Cadi, and S. Niar. Evolutionary-based co-optimization of dnn and hardware configurations on edge gpu. In *International Conference on Optimization and Learning*, pages 3–12. Springer, 2022.

[28] E. Cai, D.-C. Juan, D. Stamoulis, and D. Marculescu. *NeuralPower* : Predict and deploy energy-efficient convolutional neural networks. In M.-L. Zhang and Y.-K. Noh, editors, *Proceedings of the Ninth Asian Conference on Machine Learning*, volume 77 of *Proceedings of Machine Learning Research*, pages 622–637. PMLR, 15–17 Nov 2017.

[29] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang. Efficient architecture search by network transformation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[30] H. Cai et al. Once-for-all : Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations (ICLR)*, 2019.

[31] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han. Once-for-all : Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*.

[32] H. Cai, C. Gan, L. Zhu, and S. Han. Tinytl : Reduce memory, not parameters for efficient on-device learning. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 11285–11297. Curran Associates, Inc., 2020.

[33] H. Cai, L. Zhu, and S. Han. Proxylessnas : Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations*, 2018.

[34] M. B. Calisto and S. K. Lai-Yuen. Emonas-net : Efficient multiobjective neural architecture search using surrogate-assisted evolutionary algorithm for 3d medical image segmentation. *Artificial Intelligence in Medicine*, 119 :102154, 2021.

[35] C.-H. Chan, L. Cheng, W. Deng, P. Feng, L. Geng, M. Huang, H. Jia, L. Jie, K.-M. Lei, X. Liu, et al. Trending ic design directions in 2022. *Journal of Semiconductors*, 43(7) :071401, 2022.

[36] 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.

[37] 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.

[38] L.-C. Chen, M. Collins, Y. Zhu, G. Papandreou, B. Zoph, F. Schroff, H. Adam, and J. Shlens. Searching for efficient multi-scale architectures for dense image prediction. *Advances in neural information processing systems*, 31, 2018.

[39] T. Chen and C. Guestrin. Xgboost : A scalable tree boosting system. KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[40] W. Chen, Y. Wang, S. Yang, C. Liu, and L. Zhang. You only search once : A fast automation framework for single-stage DNN/accelerator co-design. In *2020 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1283–1286. IEEE, 2020.

[41] Y. Chen, J. Li, H. Xiao, X. Jin, S. Yan, and J. Feng. Dual path networks. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[42] Y. Chen, G. Meng, Q. Zhang, S. Xiang, C. Huang, L. Mu, and X. Wang. Renas : Reinforced evolutionary neural architecture search. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 4787–4796, 2019.

[43] Y.-H. Chen, J. Emer, and V. Sze. Eyeriss : A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH computer architecture news*, 44(3) :367–379, 2016.

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

[45] H. Cho, J. Shin, and W. Rhee. B2ea : An evolutionary algorithm assisted by two bayesian optimization modules for neural architecture search. *arXiv preprint arXiv :2202.03005*, 2022.

[46] K. Choi, D. Hong, H. Yoon, J. Yu, Y. Kim, and J. Lee. Dance : Differentiable accelerator/network co-exploration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 337–342. IEEE, 2021.

[47] F. Chollet. Xception : Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[48] X. Chu, B. Zhang, and R. Xu. Fairnas : Rethinking evaluation fairness of weight sharing neural architecture search. In *Proceedings of the IEEE/CVF International Conference on computer vision*, pages 12239–12248, 2021.

[49] 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.

[50] A. de Myttenaere, B. Golden, B. Le Grand, and F. Rossi. Mean absolute percentage error for regression models. *Neurocomputing*, 192 :38–48, 2016. Advances in artificial neural networks, machine learning and computational intelligence.

[51] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization : Nsga-ii. In *Parallel Problem Solving from Nature PPSN VI : 6th International Conference Paris, France, September 18–20, 2000 Proceedings 6*, pages 849–858. Springer, 2000.

[52] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm : Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2) :182–197, 2002.

[53] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet : A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[54] P. Dong, M. Sun, A. Lu, Y. Xie, K. Liu, Z. Kong, X. Meng, Z. Li, X. Lin, Z. Fang, et al. Heatvit : Hardware-efficient adaptive token pruning for vision transformers. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 442–455. IEEE, 2023.

[55] X. Dong, D. J. Kedziora, K. Musial, and B. Gabrys. Automated deep learning : Neural architecture search is not the end. *arXiv preprint arXiv :2112.09245*, 2021.

[56] X. Dong and Y. Yang. Nas-bench-201 : Extending the scope of reproducible neural architecture search. In *International Conference on Learning Representations*, 2019.

[57] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al. An image is worth 16x16 words : Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2020.

[58] S. Duan, D. Wang, J. Ren, F. Lyu, Y. Zhang, H. Wu, and X. Shen. Distributed artificial intelligence empowered by end-edge-cloud computing : A survey. *IEEE Communications Surveys and Tutorials*, 2022.

[59] T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search : A survey. *The Journal of Machine Learning Research*, 20(1) :1997–2017, 2019.

[60] J. Eric, T. Floris, A. Theo, B. Halima, C. Ramon, H. Omar, L.-S. Cyril, M. Christophe, N. Smail, T.-M. Serge, and P. Thierion. An evaluation bench for the exploration of machine learning deployment solutions on embedded platforms. *The European Congress on Real Time Embedded Systems (ERTS)*, 2024.

[61] L. Fan and H. Wang. Surrogate-assisted evolutionary neural architecture search with network embedding. *Complex and Intelligent Systems*, 9(3) :3313–3331, 2023.

[62] 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.

[63] N. Fasfous et al. Anaconga : analytical hw-cnn co-design using nested genetic algorithms. In *2022 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 238–243. IEEE, 2022.

[64] J. Frankle and M. Carbin. The lottery ticket hypothesis : Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2018.

[65] X. Fu, Q. Ren, H. Wu, F. Xiang, Q. Luo, J. Yue, Y. Chen, and F. Zhang. A cim-based high-utilization architecture with dynamic pruning and two-way ping-pong macro for vision transformer. *IEEE Transactions on Circuits and Systems I : Regular Papers*, 2023.

[66] B. Gaide et al. Xilinx adaptive compute acceleration platform : Versaltm architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 84–93, 2019.

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

[68] 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.

[69] Y. Gao, B. Zhang, X. Qi, and H. K.-H. So. Dpacs : Hardware accelerated dynamic neural network pruning through algorithm-architecture co-design. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 237–251, 2023.

[70] 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.

[71] E. C. Garrido-Merchán and D. Hernández-Lobato. Dealing with categorical and integer-valued variables in bayesian optimization with gaussian processes. *Neurocomputing*, 380 :20–35, 2020.

[72] M. I. E. Ghebriout, H. Bouzidi, S. Niar, and H. Ouarnoughi. Harmonic-nas : Hardware-aware multimodal neural architecture search on resource-constrained devices. *arXiv preprint arXiv :2309.06612*, 2023.

[73] A. Gholami, K. Kwon, B. Wu, Z. Tai, X. Yue, P. Jin, S. Zhao, and K. Keutzer. Squeezenext : Hardware-aware neural network design. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 1638–1647, 2018.

[74] S. S. Gill, M. Xu, C. Ottaviani, P. Patros, R. Bahsoon, A. Shaghaghi, M. Golec, V. Stankovski, H. Wu, A. Abraham, et al. Ai for next generation computing : Emerging trends and future directions. *Internet of Things*, 19 :100514, 2022.

[75] C. Gong, D. Wang, M. Li, X. Chen, Z. Yan, Y. Tian, V. Chandra, et al. Nasvit : Neural architecture search for efficient vision transformers with gradient conflict aware supernet training. In *International Conference on Learning Representations*, 2021.

[76] J. Gou, B. Yu, S. J. Maybank, and D. Tao. Knowledge distillation : A survey. *International Journal of Computer Vision*, 129 :1789–1819, 2021.

[77] B. Greenwood and T. McDonnell. Surrogate-assisted neuroevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1048–1056, 2022.

[78] X. Guo, A. D. Pimentel, and T. Stefanov. Automated exploration and implementation of distributed cnn inference at the edge. *IEEE Internet of Things Journal*, 10(7) :5843–5858, 2023.

[79] 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.

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

[81] 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.

[82] 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.

[83] C. Hao, X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W.-m. Hwu, and D. Chen. Fpga/DNN co-design : An efficient design methodology for 1ot intelligence on the edge. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2019.

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

[85] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016.

[86] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han. Amc : Automl for model compression and acceleration on mobile devices. In *Proceedings of the European conference on computer vision (ECCV)*, pages 784–800, 2018.

[87] R. Hosseini and P. Xie. Saliency-aware neural architecture search. *Advances in Neural Information Processing Systems*, 35 :14743–14757, 2022.

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

[89] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, et al. Searching for mobilenetv3. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1314–1324, 2019.

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

[91] J. Hu, L. Shen, and G. Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7132–7141, 2018.

[92] S. Hu, S. Xie, H. Zheng, C. Liu, J. Shi, X. Liu, and D. Lin. Dsnas : Direct neural architecture search without parameter retraining. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12084–12092, 2020.

[93] G. Huang, D. Chen, T. Li, F. Wu, L. Van Der Maaten, and K. Q. Weinberger. Multi-scale dense convolutional networks for efficient prediction. *arXiv preprint arXiv :1703.09844*, 2(2), 2017.

[94] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[95] F. Hutter, H. Hoos, and K. Leyton-Brown. An efficient approach for assessing hyperparameter importance. In *International conference on machine learning*, pages 754–762. PMLR, 2014.

[96] M. G. M. Imed, P. N. Smail, P. O. Hamza, P. B. K. ESI, and M. B. Halima. Recherche d'architectures neuronales multimodales avec la prise en compte du matériel edge.

[97] J. Janai, F. Güney, A. Behl, A. Geiger, et al. Computer vision for autonomous vehicles : Problems, datasets and state of the art. *Foundations and Trends® in Computer Graphics and Vision*, 12(1–3) :1–308, 2020.

[98] 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.

[99] W. Jiang, L. Yang, S. Dasgupta, J. Hu, and Y. Shi. Standing on the shoulders of giants : Hardware and neural architecture co-search with hot start. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11) :4154–4165, 2020.

[100] W. Jiang, L. Yang, E. H.-M. Sha, Q. Zhuge, S. Gu, S. Dasgupta, Y. Shi, and J. Hu. Hardware/software co-exploration of neural architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12) :4805–4815, 2020.

[101] J. Jo, S. Jeong, and P. Kang. Benchmarking gpu-accelerated edge devices. In *2020 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 117–120, 2020.

[102] D. Justus, J. Brennan, S. Bonner, and A. S. McGough. Predicting the computational cost of deep learning models. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 3873–3882, 2018.

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

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

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

[106] M. Kiani and A. Rajabzadeh. Sdam : a combined stack distance-analytical modeling approach to estimate memory performance in gpus. *The Journal of Supercomputing*, 77(5) :5120–5147, 2021.

[107] 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.

[108] K. Kiningham et al. Grip : A graph neural network accelerator architecture. *IEEE Transactions on Computers*, 2022.

[109] S. Kornblith, J. Shlens, and Q. V. Le. Do better imagenet models transfer better ? In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2661–2671, 2019.

[110] O. Krestinskaya, A. P. James, and L. O. Chua. Neuromemristive circuits for edge computing : A review. *IEEE transactions on neural networks and learning systems*, 31(1) :4–23, 2019.

[111] A. Krishnakumar, U. Ogras, R. Marculescu, M. Kishinevsky, and T. Mudge. Domain-specific architectures : Research problems and promising approaches. *ACM Transactions on Embedded Computing Systems*, 22(2) :1–26, 2023.

[112] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[113] A. Krizhevsky, V. Nair, and G. Hinton. Cifar-10 (canadian institute for advanced research). *URL http ://www. cs. toronto. edu/kriz/cifar. html*, 5(4) :1, 2010.

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

[115] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[116] 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.

[117] H. Kwon et al. 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.

[118] 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.

[119] R. Lange, T. Schaul, Y. Chen, C. Lu, T. Zahavy, V. Dalibard, and S. Flennerhag. Discovering attention-based genetic algorithms via meta-black-box optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 929–937, 2023.

[120] S. Laskaridis, A. Kouris, and N. D. Lane. Adaptive inference through early-exit networks : Design, challenges and directions. In *Proceedings of the 5th International Workshop on Embedded and Mobile Deep Learning*, pages 1–6, 2021.

[121] 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.

[122] Y. Lee, J. Kim, J. Willette, and S. J. Hwang. Mpvit : Multi-path vision transformer for dense prediction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7287–7296, 2022.

[123] Y.-L. Lee, P.-K. Tsung, and M. Wu. Techology trend of edge ai. In *2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pages 1–2, 2018.

[124] C. Li, A. Dakkak, J. Xiong, W. Wei, L. Xu, and W.-m. Hwu. Xsp : Across-stack profiling and analysis of machine learning models on gpus. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 326–327, 2020.

[125] 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.

[126] T. Li, J. Hou, J. Yan, R. Liu, H. Yang, and Z. Sun. Chiplet heterogeneous integration technology—status and challenges. *Electronics*, 9(4) :670, 2020.

[127] X. Li, C. Lou, Y. Chen, Z. Zhu, Y. Shen, Y. Ma, and A. Zou. Predictive exit : Prediction of fine-grained early exits for computation-and energy-efficient inference. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 8657–8665, 2023.

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

[129] Y. Liang, L. Lu, Y. Jin, J. Xie, R. Huang, J. Zhang, and W. Lin. An efficient hardware design for accelerating sparse CNNs with NAS-based models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[130] A. Liaw, M. Wiener, et al. Classification and regression by randomforest. *R news*, 2(3) :18–22, 2002.

[131] 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.

[132] Y. Lin, D. Hafdi, K. Wang, Z. Liu, and S. Han. Neural-hardware architecture search. *NeurIPS WS*, 2019.

[133] Y. Lin, M. Yang, and S. Han. Naas : Neural accelerator architecture search. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1051–1056. IEEE, 2021.

[134] M. Lindauer and F. Hutter. Best practices for scientific research on neural architecture search. *The Journal of Machine Learning Research*, 21(1) :9820–9837, 2020.

[135] C.-H. Liu, Y.-S. Han, Y.-Y. Sung, Y. Lee, H.-Y. Chiang, and K.-C. Wu. Fox-nas : Fast, on-device and explainable neural architecture search. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 789–797, 2021.

[136] P. Liu, B. Wu, H. Ma, and M. Seok. Memnet : memory-efficiency guided neural architecture search with augment-trim learning. *arXiv preprint arXiv :1907.09569*, 2019.

[137] S. Liu, H. Zhang, and Y. Jin. A survey on computationally efficient neural architecture search. *Journal of Automation and Intelligence*, 1(1) :100002, 2022.

[138] Y. Liu, Y. Sun, B. Xue, M. Zhang, G. G. Yen, and K. C. Tan. A survey on evolutionary neural architecture search. *IEEE transactions on neural networks and learning systems*, 2021.

[139] Z. Liu, J. Xu, X. Peng, and R. Xiong. Frequency-domain dynamic pruning for convolutional neural networks. *Advances in neural information processing systems*, 31, 2018.

[140] W. Lou, L. Xun, A. Sabet, J. Bi, J. Hare, and G. V. Merrett. Dynamic-ofa : Runtime dnn architecture switching for performance scaling on heterogeneous embedded platforms. In *Conference on Computer Vision and Pattern Recognition*, pages 3110–3118, 2021.

[141] Q. Lu, W. Jiang, X. Xu, Y. Shi, and J. Hu. On neural architecture search for resource-constrained hardware platforms. In *International Conference on Computer-Aided Design*, 2019.

[142] Z. Lu, K. Deb, E. Goodman, W. Banzhaf, and V. N. Boddeti. Nsganetv2 : Evolutionary multi-objective surrogate-assisted neural architecture search. In *Computer Vision–ECCV 2020 : 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part I 16*, pages 35–51. Springer, 2020.

[143] Z. Lu, S. Rallapalli, K. Chan, and T. La Porta. Modeling the resource requirements of convolutional neural networks on mobile devices. In *Proceedings of the 25th ACM International Conference on Multimedia*, MM '17, page 1663–1671, New York, NY, USA, 2017. Association for Computing Machinery.

[144] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun. Shufflenet v2 : Practical guidelines for efficient cnn architecture design. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 116–131, 2018.

[145] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo. Performance modeling for cnn inference accelerators on fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(4) :843–856, 2020.

[146] 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.

[147] 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.

[148] S. D. Manasi and S. S. Sapatnekar. Deepopt : Optimized scheduling of cnn workloads for asic-based systolic deep learning accelerators. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, ASPDAC '21, page 235–241, New York, NY, USA, 2021. Association for Computing Machinery.

[149] J. Mao et al. Modnn : Local distributed mobile computing system for deep neural network. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2017.

[150] L. Mei, K. Goetschalckx, A. Symons, and M. Verhelst. Defines : Enabling fast exploration of the depth-first scheduling space for dnn accelerators through analytical modeling. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 570–583. IEEE, 2023.

[151] P. Molchanov, A. Mallya, S. Tyree, I. Frosio, and J. Kautz. Importance estimation for neural network pruning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11264–11272, 2019.

[152] M. A. H. Monil et al. Mephesto : Modeling energy-performance in heterogeneous socs and their trade-offs. In *PACT*, pages 413–425, 2020.

[153] C. Moussa, Y. J. Patel, V. Dunjko, T. Bäck, and J. N. van Rijn. Hyperparameter importance and optimization of quantum neural networks across small datasets. *Machine Learning*, pages 1–26, 2023.

[154] S. Mukhopadhyay, Y. Long, B. Mudassar, C. Nair, B. H. DeProspo, H. M. Torun, M. Kathaperumal, V. Smet, D. Kim, S. Yalamanchili, et al. Heterogeneous integration for artificial intelligence : Challenges and opportunities. *IBM Journal of Research and Development*, 63(6) :4–1, 2019.

[155] R. T. Mullapudi, W. R. Mark, N. Shazeer, and K. Fatahalian. Hydranets : Specialized dynamic architectures for efficient inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8080–8089, 2018.

[156] F. Murtagh. Multilayer perceptrons for classification and regression. *Neurocomputing*, 2(5) :183–197, 1991.

[157] S. M. Nabavinejad, S. Reda, and M. Ebrahimi. Coordinated batching and DVFS for DNN inference on gpu accelerators. *IEEE Transactions on Parallel and Distributed Systems*, 2022.

[158] P. E. Nogueira, R. Matias, and E. Vicente. An experimental study on execution time variation in computer experiments. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, page 1529–1534, New York, NY, USA, 2014. Association for Computing Machinery.

[159] Nvidia. Nvidia profiler (nvprof), Jun 2007.

[160] Nvidia. Harness AI at the edge with the jetson TX2 developer kit, Jun 2018.

[161] Nvidia. Intel movidius myriad 2, 2018.

[162] Nvidia. Jetson AGX xavier developer kit, Jun 2018.

[163] Nvidia. Jetson Nano developer kit, Jun 2018.

[164] Nvidia. Tegrastats utility, Jun 2019.

[165] 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.

[166] 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.

[167] E. Ostertagová. Modelling using polynomial regression. *Procedia Engineering*, 48 :500–506, 2012. Modelling of Mechanical and Mechatronics Systems.

[168] C. Pan and X. Yao. Neural architecture search based on evolutionary algorithms with fitness approximation. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2021.

[169] P. Panda, A. Sengupta, and K. Roy. Conditional deep learning for energy-efficient and enhanced pattern recognition. In *2016 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 475–480. IEEE, 2016.

[170] D. Paul, J. Singh, and J. Mathew. Hardware-software co-design approach for deep learning inference. In *2019 7th International Conference on Smart Computing and Communications (ICSCC)*, pages 1–5. IEEE, 2019.

[171] Y. Peng, A. Song, V. Ciesielski, H. M. Fayek, and X. Chang. Pre-nas : predictor-assisted evolutionary neural architecture search. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1066–1074, 2022.

[172] F. Pérez-Cruz. Kullback-leibler divergence estimation of continuous distributions. In *2008 IEEE international symposium on information theory*, pages 1666–1670. IEEE, 2008.

[173] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean. Efficient neural architecture search via parameters sharing. In *International conference on machine learning*, pages 4095–4104. PMLR, 2018.

[174] 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.

[175] M. Pinos, V. Mrazek, and L. Sekanina. Evolutionary neural architecture search supporting approximate multipliers. In *European Conference on Genetic Programming (Part of EvoStar)*, pages 82–97. Springer, 2021.

[176] 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.

[177] H. Qi, E. R. Sparks, and A. Talwalkar. Paleo : A performance model for deep neural networks. 2016.

[178] H. Qin, R. Gong, X. Liu, X. Bai, J. Song, and N. Sebe. Binary neural networks : A survey. *Pattern Recognition*, 105 :107281, 2020.

[179] Z. Qiu, W. Bi, D. Xu, H. Guo, H. Ge, Y. Liang, H. P. Lee, and C. Wu. Efficient self-learning evolutionary neural architecture search. *Applied Soft Computing*, 146 :110671, 2023.

[180] Y. Rao et al. Dynamicvit : Efficient vision transformers with dynamic token sparsification. *NeurIPS*, 34, 2021.

[181] S. Ravanbakhsh, J. Schneider, and B. Poczos. Equivariance through parameter-sharing. In *International conference on machine learning*, pages 2892–2901. PMLR, 2017.

[182] W.-Q. Ren, Y.-B. Qu, C. Dong, Y.-Q. Jing, H. Sun, Q.-H. Wu, and S. Guo. A survey on collaborative dnn inference for edge intelligence. *Machine Intelligence Research*, 20(3) :370–395, 2023.

[183] C. F. Rodrigues, G. Riley, and M. Luján. Fine-grained energy profiling for deep convolutional neural networks on the jetson tx1. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 114–115, 2017.

[184] J. D. Rodriguez, A. Perez, and J. A. Lozano. Sensitivity analysis of k-fold cross validation in prediction error estimation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(3) :569–575, 2010.

[185] W. Samek, G. Montavon, S. Lapuschkin, C. J. Anders, and K.-R. Müller. Explaining deep neural networks and beyond : A review of methods and applications. *Proceedings of the IEEE*, 109(3) :247–278, 2021.

[186] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2 : Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[187] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2 : Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[188] S. Scardapane, M. Scarpiniti, E. Baccarelli, and A. Uncini. Why should we add early exits to neural networks ? *Cognitive Computation*, 12(5) :954–966, 2020.

[189] L. Sekanina. Neural architecture search and hardware accelerator co-search : A survey. *IEEE Access*, 9 :151337–151362, 2021.

[190] E. Shamsa et al. Goal-driven autonomy for efficient on-chip resource management : Transforming objectives to goals. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2019.

[191] 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.

[192] J. Shen, Y. Wang, P. Xu, Y. Fu, Z. Wang, and Y. Lin. Fractional skipping : Towards finer-grained dynamic cnn inference. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 5700–5708, 2020.

[193] Y. Shen, Y. Li, J. Zheng, W. Zhang, P. Yao, J. Li, S. Yang, J. Liu, and B. Cui. Proxybo : Accelerating neural architecture search via bayesian optimization with zero-cost proxies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 9792–9801, 2023.

[194] H. Shi, R. Pi, H. Xu, Z. Li, J. Kwok, and T. Zhang. Bridging the gap between sample-based and one-shot neural architecture search with bonas. *Advances in Neural Information Processing Systems*, 33 :1808–1819, 2020.

[195] 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.

[196] S. Shi, Q. Wang, and X. Chu. Performance modeling and evaluation of distributed deep learning frameworks on gpus. In *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, pages 949–957, 2018.

[197] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing : Vision and challenges. *IEEE internet of things journal*, 3(5) :637–646, 2016.

[198] J. N. Siems, L. Zimmer, A. Zela, J. Lukasik, M. Keuper, and F. Hutter. Nas-bench-301 and the case for surrogate benchmarks for neural architecture search. 2020.

[199] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In Y. Bengio et al., editors, *ICLR*, 2015.

[200] R. Singh and S. S. Gill. Edge ai : a survey. *Internet of Things and Cyber-Physical Systems*, 2023.

[201] N. Sinha and K.-W. Chen. Novelty driven evolutionary neural architecture search. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 671–674, 2022.

[202] K. Siu, D. M. Stuart, M. Mahmoud, and A. Moshovos. Memory requirements for convolutional neural network hardware accelerators. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 111–121, 2018.

[203] Y. Song et al. Sara : Self-aware resource allocation for heterogeneous mpsocs. In *DAC*, 2018.

[204] D. Stamoulis, E. Cai, D.-C. Juan, and D. Marculescu. Hyperpower : Power- and memory-constrained hyper-parameter optimization for neural networks. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 19–24, 2018.

[205] 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.

[206] Q. Sun, T. Chen, J. Miao, and B. Yu. Power-driven dnn dataflow optimization on fpga. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–7, 2019.

[207] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer. Efficient processing of deep neural networks : A tutorial and survey. *Proceedings of the IEEE*, 105(12) :2295–2329, 2017.

[208] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI conference on artificial intelligence*, pages 4278–4284, 2017.

[209] 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.

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

[211] E. Talpes et al. Compute solution for tesla's full self-driving computer. *IEEE Micro*, 40(2) :25–35, 2020.

[212] T. Tambe and Al. Edgebert : Sentence-level energy optimizations for latency-aware multi-task nlp inference. In *Micro-54*, pages 830–844, 2021.

[213] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le. Mnasnet : Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[214] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le. Mnasnet : Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[215] M. Tan and Q. Le. Efficientnet : Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019.

[216] 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.

[217] K. R. Traoré, A. Camero, and X. X. Zhu. A data-driven approach to neural architecture search initialization. *Annals of Mathematics and Artificial Intelligence*, pages 1–28, 2023.

[218] Velasco-Montero and other. Previous : A methodology for prediction of visual inference performance on iot devices. *IEEE Internet of Things Journal*, 7(10) :9227–9240, 2020.

[219] D. Wang, C. Gong, M. Li, Q. Liu, and V. Chandra. Alphanet : Improved training of supernets with alpha-divergence. In *International Conference on Machine Learning*. PMLR, 2021.

[220] D. Wang, M. Li, C. Gong, and V. Chandra. Attentivenas : Improving neural architecture search via attentive sampling. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021.

[221] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han. Haq : Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 8612–8620, 2019.

[222] L. Wang, S. Xie, T. Li, R. Fonseca, and Y. Tian. Sample-efficient neural architecture search by learning actions for monte carlo tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(9) :5503–5515, 2021.

[223] M. Wang, C. Meng, G. Long, C. Wu, J. Yang, W. Lin, and Y. Jia. Characterizing deep learning training workloads on alibaba-pai. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 189–202, 2019.

[224] 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.

[225] S. Watanabe, A. Bansal, and F. Hutter. Ped-anova : Efficiently quantifying hyperparameter importance in arbitrary subspaces. *arXiv preprint arXiv :2304.10255*, 2023.

[226] C. Wei, C. Niu, Y. Tang, Y. Wang, H. Hu, and J. Liang. Npenas : Neural predictor guided evolution for neural architecture search. *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

[227] G. Wetzstein, A. Ozcan, S. Gigan, S. Fan, D. Englund, M. Soljačić, C. Denz, D. A. Miller, and D. Psaltis. Inference in artificial intelligence with deep optics and photonics. *Nature*, 588(7836) :39–47, 2020.

[228] C. White, W. Neiswanger, S. Nolen, and Y. Savani. A study on encodings for neural architecture search. *Advances in neural information processing systems*, 33 :20309–20319, 2020.

[229] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer. Fbnet : Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10734–10742, 2019.

[230] B. Wu, F. Iandola, P. H. Jin, and K. Keutzer. Squeezedet : Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 129–137, 2017.

[231] K. Wu, Y. Guo, and C. Zhang. Compressing deep neural networks with sparse matrix factorization. *IEEE transactions on neural networks and learning systems*, 31(10) :3828–3838, 2019.

[232] 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.

[233] H. Xiao, Z. Wang, Z. Zhu, J. Zhou, and J. Lu. Shapley-nas : discovering operation contribution for neural architecture search. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11892–11901, 2022.

[234] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500, 2017.

[235] 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.

[236] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks ? In *International Conference on Learning Representations*, 2019.

[237] Y. Xu et al. Pccs : Processor-centric contention-aware slowdown model for heterogeneous system-on-chips. In *MICRO*, 2021.

[238] Y. Xue, Y. Wang, J. Liang, and A. Slowik. A self-adaptive mutation neural architecture search algorithm based on blocks. *IEEE Computational Intelligence Magazine*, 16(3) :67–78, 2021.

[239] L. Xun et al. Optimising resource management for embedded machine learning. In *2020 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1556–1561. IEEE, 2020.

[240] 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.

[241] 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.

[242] 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.

[243] J. Yang, X. Shen, J. Xing, X. Tian, H. Li, B. Deng, J. Huang, and X.-s. Hua. Quantization networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7308–7316, 2019.

[244] L. Yang, Z. Yan, M. Li, H. Kwon, L. Lai, T. Krishna, V. Chandra, W. Jiang, and Y. Shi. Co-exploration of neural architectures and heterogeneous ASIC accelerator designs targeting multiple tasks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[245] T.-J. Yang, Y.-H. Chen, and V. Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6071–6079, 2017.

[246] Y. Yang, D. Liu, H. Fang, Y.-X. Huang, Y. Sun, and Z.-Y. Zhang. Once for all skip : efficient adaptive deep neural networks. In *2022 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 568–571. IEEE, 2022.

[247] A. Yao and D. Sun. Knowledge transfer via dense cross-layer mutual-distillation. In *Computer Vision–ECCV 2020 : 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XV 16*, pages 294–311. Springer, 2020.

[248] H. Yao, D.-l. Zhu, B. Jiang, and P. Yu. Negative log likelihood ratio loss for deep neural network classification. In *Proceedings of the Future Technologies Conference (FTC) 2019 : Volume 1*, pages 276–282. Springer, 2020.

[249] Y. Yin, S. Huang, and X. Zhang. Bm-nas : Bilevel multimodal neural architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 8901–8909, 2022.

[250] 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.

[251] J. You, Z. Ying, and J. Leskovec. Design space for graph neural networks. *Advances in Neural Information Processing Systems*, 33 :17009–17021, 2020.

[252] 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.

[253] J. Yu and T. S. Huang. Universally slimmable networks and improved training techniques. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 1803–1811, 2019.

[254] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke. Scalpel : Customizing dnn pruning to the underlying hardware parallelism. *ACM SIGARCH Computer Architecture News*, 45(2) :548–560, 2017.

[255] J. Yu, L. Yang, N. Xu, J. Yang, and T. Huang. Slimmable neural networks. In *International Conference on Learning Representations*, 2018.

[256] 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.

[257] Z. Yu et al. Mia-former : Efficient and robust vision transformers via multi-grained input-adaptation. In *AAAI*, volume 36, 2022.

[258] Z. Yuan, X. Liu, B. Wu, and G. Sun. Enas4d : Efficient multi-stage cnn architecture search for dynamic inference. *arXiv preprint arXiv :2009.09182*, 2020.

[259] 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.

[260] B. Zhang et al. Low-latency mini-batch gnn inference on cpu-fpga heterogeneous platform. In *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 11–21, 2022.

[261] X. Zhang, X. Zhou, M. Lin, and J. Sun. Shufflenet : An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6848–6856, 2018.

[262] 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.

[263] Y. Zhang, Y. Fu, W. Jiang, C. Li, H. You, M. Li, V. Chandra, and Y. Lin. Dna : Differentiable network-accelerator co-search. *arXiv preprint arXiv :2010.14778*, 2020.

[264] X. Zheng, R. Ji, Y. Chen, Q. Wang, B. Zhang, J. Chen, Q. Ye, F. Huang, and Y. Tian. Migo-nas : Towards fast and generalizable neural architecture search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(9) :2936–2952, 2021.

[265] X. Zheng, R. Ji, Q. Wang, Q. Ye, Z. Li, Y. Tian, and Q. Tian. Rethinking performance estimation in neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11356–11365, 2020.

[266] F. Zhengxin, Y. Yi, Z. Jingyu, L. Yue, M. Yuechen, L. Qinghua, X. Xiwei, W. Jeff, W. Chen, Z. Shuai, et al. Mlops spanning whole machine learning life cycle : A survey. *arXiv preprint arXiv :2304.07296*, 2023.

[267] 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.

[268] K. Zhou et al. Auto-gnn : Neural architecture search of graph neural networks. *Frontiers in big Data*, 2022.

[269] Y. Zhou, X. Dong, B. Akin, M. Tan, D. Peng, T. Meng, A. Yazdanbakhsh, D. Huang, R. Narayanaswami, and J. Laudon. Rethinking co-design of neural architectures and hardware accelerators. *arXiv preprint arXiv :2102.08619*, 2021.

[270] B. Zoph and Q. Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations*, 2016.

[271] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710. IEEE, 2018.

[272] J. Zou, T. Rui, Y. Zhou, C. Yang, and S. Zhang. Convolutional neural network simplification via feature map pruning. *Computers and Electrical Engineering*, 70 :950–958, 2018.