System-level design for efficient execution of CNNs at the edge

Minakova, S.

System-level Design For Efficient Execution of CNNs at the Edge

Svetlana Minakova

This work has received funding from the European Unions Horizon 2020 Research and Innovation project under grant agreement No. 780788.

System-level Design For Efficient Execution of CNNs at the Edge. Svetlana Minakova. - Dissertation Universiteit Leiden.

This dissertation was typeset using LaTeX.
Cover design: from images generated using DALL·E mini Deep Learning algorithm [20]. The images are combined and post-processed by Anna Minakova.

System-level Design For Efficient Execution of CNNs at the Edge

Proefschrift

ter verkrijging van
de graad van doctor aan de Universiteit Leiden,
op gezag van rector magnificus prof.dr.ir. H. Bijl,
volgens besluit van het college voor promoties
te verdedigen op donderdag 24 november 2022
klokke 11.15 uur

door

Svetlana Minakova
geboren te Ryazan, Rusland
in 1993

Promotores:
Dr. T.P. Stefanov
Prof.dr. H.A.G. Wijshoff

Promotiecommissie:
Prof.dr. S. Ha (Seoul National University)
Prof.dr. J. Castrillon (Technical University of Dresden)
Prof.dr. H.E. Bal (Vrije Universiteit Amsterdam)
Prof.dr. A. Plaat
Prof.dr. N. Mentens
Prof.dr. M.S.K. Lew

To my family and friends

Contents

List of Figures

List of Tables

List of Abbreviations

BFS	Breadth-First Search
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CSDF	Cyclo-Static Data Flow
DL	Deep Learning
DSE	Design Space Exploration
FC	Fully Connected
FLOP	floating-point operation
FPGA	Field-programmable Gate Array
GA	Genetic Algorithm
GPU	Graphics Processing Unit
HAR	Human Activity Recognition
IoT	Internet-of-Things
KD	Knowledge Distillation
MB	MegaBytes
MoC	Model of Computation
MPSoC	Multi-Processor System-on-Chip
NAS	Neural Architecture Search

ONNX	Open Neural Network Exchange Format
SBRS	Scenario-Based Run-time Switching
SDF	Synchronous Data Flow
SOTA	State Of The Art
SSR	Scenario Switch Request
TPU	Tensor Processing Unit
UAV	Unmanned Aerial Vehicle

Chapter 1

Introduction

IN recent years, the field of Deep Learning (DL) [33] has received great attention. This new and rapidly developing field has achieved state-of-the-art results in solving problems in areas such as image processing, computer vision, speech recognition, machine translation, medical information processing, robotics and control, bio-informatics, natural language processing, and many others [4]. One of the most popular DL algorithms are Convolutional Neural Networks (CNNs) [56]. Nowadays, CNNs are the front-runners of image processing and computer vision tasks such as image segmentation, classification, and object detection in both academia and industry [4]. The success of CNNs is due to their ability to automatically, effectively, and adaptively extract and process high- and low-level abstractions from multi-dimensional (2D and 3D) data such as images or video. This capability is mostly associated with the CNNs architecture, inspired by the biological processes in the visual cortex of an animal [55].

A CNN consists of a set of interconnected elements, called *neurons*. The connected neurons exchange data: each neuron accepts input data, provided by other neurons or external sources, and generates data for other neurons. To generate its output data, a neuron applies a mathematical *operator* such as convolution, dot product, pooling, and others [59] to its input data. To perform the operator, the neuron uses a set of *parameters*, also referred as *weights*. The values of the weights are obtained via *training*: a computationally intensive procedure, through which a CNN processes large volumes of data and learns how to perform its respective task. The neurons performing the same operator form hierarchically organized groups called *layers*. Typically, a CNN has one input layer, one output layer, and one or more hidden layers. The input layer receives the CNN input data (e.g. an image) and passes it to

the first hidden layer. The hidden layers transform the input data using the respective operators, and pass the data from the input layer to the output layer. Finally, the output layer produces the CNN output (e.g. an image classification result). A simplified example of a CNN, performing image classification, is shown in Figure 1.1. The CNN has one input layer, one output layer, and six hidden layers. The number of layers in the CNN and the number of neurons per layer are often referred as the CNN *depth* and *width*, respectively.

The state-of-the-art CNNs are characterized with large width (hundreds of neurons per layer) and depth (hundreds to thousands of layers). They have millions of parameters and perform billions of computations, requiring large amount of hardware platform resources for their deployment and execution. Therefore, CNNs are usually deployed on high-performance platforms: GPU clusters and data centres. For applications, deployed on *Edge platforms* (mobile phones, tablets, cameras, etc.), CNNs are typically provided as cloud services. Execution of a CNN as a cloud service is shown in Figure 1.2 (on the left). To use a CNN provided as a cloud service, an application deployed at the Edge communicates with a server over the Internet. First, the application sends a request to the cloud server. The request contains data collected at the Edge, e.g., images from a CCTV camera. Then, a CNN deployed on a high-performance platform in the cloud processes the data (e.g., performs classification of the images) and sends the result back to the Edge platform.

It is important to note that the communication between the cloud server and the Edge platform takes place over the Internet. Because of this, execution of CNNs as cloud services suffers from low responsiveness and has privacy issues. This is unacceptable for many CNN-based applications. For example,

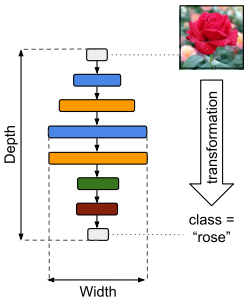


Figure 1.1: *CNN*

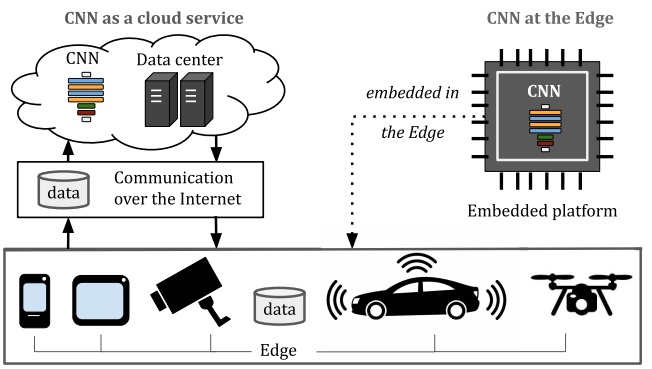


Figure 1.2: *Execution of CNNs as cloud services and at the Edge*

CNN-based navigation in self-driving cars [24] cannot tolerate variable and large response delays occurring due to the communication between the car and a server. These delays can lead to incorrect navigation of the car and, subsequently, endanger the life of passengers. Another example is applications in medicine [62] that use CNNs to analyse private data of patients. These applications cannot send their data over the Internet because this could lead to private data leakages and violation of patients' privacy rights. To address these concerns, many modern CNN-based applications shift the execution of CNNs to the Edge. Execution of a CNN at the Edge is shown in Figure 1.2 (on the right). When executed at the Edge, CNNs are deployed close to the source of data (e.g. camera or sensors) and to the rest of the CNN-based application (e.g. camera software, which collects the application data). They do not require communication over the Internet and ensure high application responsiveness and security. In this thesis, we focus on deployment and execution of CNNs at the Edge.

1.1 Accuracy and platform-aware characteristics of a CNN

Execution of a CNN is characterized by accuracy and platform-aware characteristics. The *accuracy* of a CNN (typically measured in %) is the fraction of correct predictions generated by the CNN from the total number of predictions generated by the CNN. It is the main quality metric of a CNN which quantifies the CNN's ability to perform its respective task, e.g., to classify images correctly. The higher the CNN accuracy is, the better the CNN is at performing its respective task.

The *platform-aware characteristics* characterize the execution of a CNN on a target platform. The most common platform-aware characteristics of a CNN are:

- *throughput* (typically measured in frames per second, fps), i.e., the amount of data samples (e.g. images) processed per second;
- *latency* (typically measured in seconds, s), i.e., the time taken by a CNN to process one input sample (e.g. one image);
- *energy cost* (typically measured in Joules), i.e., the total amount of energy, required by a CNN to process one input sample;
- *memory cost* (typically measured in MegaBytes, MB), i.e., the total amount of memory, required to deploy and execute a CNN.

1.2 Requirements posed on a CNN executed at the Edge

While execution of CNNs at the Edge is desirable and beneficial, it is also very challenging due to numerous requirements posed on the CNNs by the CNN-based applications and target edge platforms. These requirements concern the characteristics of a CNN introduced in Section 1.1. With respect to these characteristics, the most common requirements, posed on CNNs executed at the Edge, are:

- *high accuracy*: the CNNs should be able to perform their tasks very well;
- *high throughput*: applications, such as CNN-based navigation in self-driving cars [24], require CNNs to process their input data streams fast, i.e., to have high throughput;
- *low latency*: many applications, such as navigation in drones [53], require CNNs to have low latency, i.e., as small as possible delay between accepting an input and providing the respective output;
- *low memory cost*: typical edge platforms, used for CNN execution, have limited amount of memory available. Thus, to be deployed and executed on these platforms, CNNs should have low memory footprint;
- *low energy cost*: the energy budget of edge platforms, especially battery-powered ones such as drones [58], is very limited. Thus, CNNs executed on such platforms should have low energy consumption.

1.3 Current trends in the design of CNNs executed at the Edge

State-of-the-art methodologies for designing CNNs executed at the Edge typically follow the design flow shown in Figure 1.3. The heart of the design flow is the *CNN optimization engine* which performs automated search for optimal CNN architecture and weights. To perform the search, the CNN optimization engine uses techniques such as platform-aware Neural Architecture Search (NAS) [9,25,34,38,46,92,105] and CNN compression [41,99,106]. As inputs, the CNN optimization engine accepts: 1) A set of requirements posed on the CNN. The typical requirements posed on a CNN executed at the Edge are introduced in Section 1.2; 2) A *search space* which determines how the architecture of a

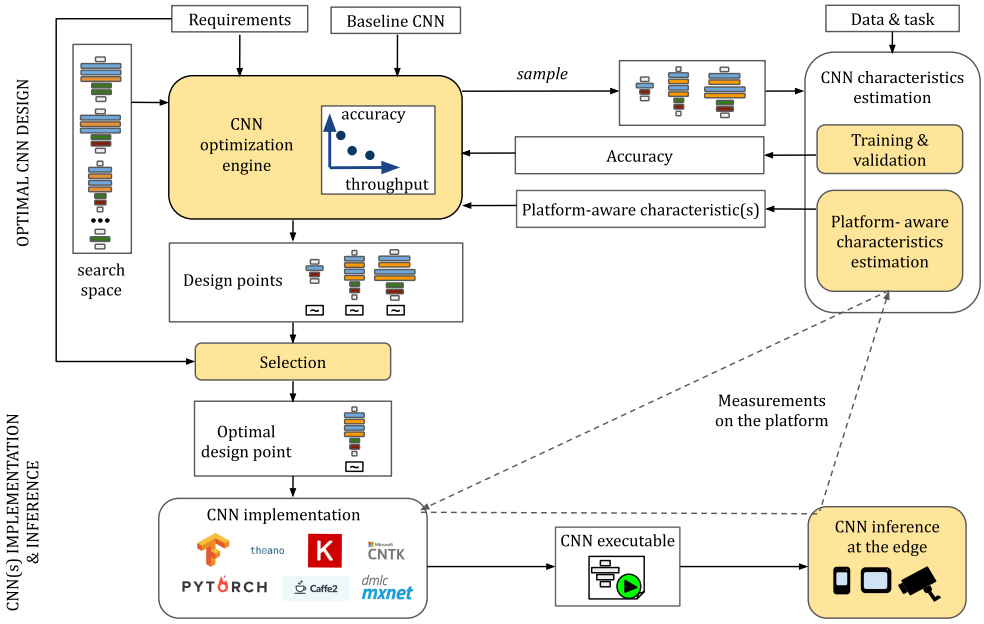


Figure 1.3: *Current trends in the design and inference of CNN-based applications executed at the Edge*

CNN can be constructed, i.e., which operators can be used by the CNN layers, which connections exist among the neurons of the layers, how many neurons and layers can a CNN have, etc. Also, it determines which CNN architectures are valid. Often specified as a set of rules, the search space defines a very large or even unbound set of valid CNNs that are able to solve the desired task; 3) (Optionally) A *baseline CNN*: a well-known, typically hand-crafted CNN, proven to solve the required task with high accuracy. The baseline CNN determines how the search is initialized, i.e., which CNNs are considered at the first step of the search. If no baseline CNN is specified, the CNN optimization engine initializes the search with CNNs randomly selected from the search space.

After the search is initialized with the first sample set of CNNs, the CNN optimization engine starts to explore the search space. The sample CNNs are passed to the *CNN characteristics estimation* component, which estimates the accuracy and platform-aware characteristics of the CNNs and returns the estimations back to the CNN optimization engine. The estimation of the CNN accuracy typically involves *training* and *validation* of the CNN. During the training, the CNN processes large volumes of data and learns how to perform

its task. During the validation, the CNN is applied to new data, unseen during the CNN training, and the CNN accuracy is computed [78]. The estimation of the platform-aware characteristics of a CNN involves either direct measurements on the target edge platform [105], or analytical formulas [34], or a combination of measurements on the platform together with analytical formulas [105]. It is worth noting that most of the approaches used for estimation of the platform-aware characteristics employ the combined estimation. Therefore, these approaches enable for more efficient (in terms of time and labour) estimation compared to only measurements on the platform, and more precise estimation compared to only analytical formulas [54, 60, 103].

Based on the received estimations, the input requirements, and the employed search/optimization strategy, the CNN optimization engine tries to improve the characteristics of the sample CNNs by altering the architecture and (possibly) the amount of CNNs. Typical alterations of a CNN architecture include changing the size (width and depth) of the CNN, adding and removing connections between the CNN neurons, reducing the precision of the CNN data and weights, and others [9, 25, 41, 99, 106]. The updated sample CNNs are then forwarded again for characteristics estimation. Thus, the exploration of the search space is a cycle, where every iteration involves sampling of CNNs and estimation of the CNNs' characteristics. The cycle continues until either a certain number of iterations is performed or a special condition is met (e.g., characteristics of the CNNs no longer improve). The result of the exploration is a set of CNNs, characterized with different architecture, weights, accuracy, and platform-aware characteristics. Hereinafter, we refer to these CNNs as to *design points*. The design points are passed to the *selection* component which chooses a single *optimal design point* from these CNNs.

The optimal design point is *implemented*, i.e., represented as an executable application and deployed on the target edge platform. The implementation and deployment of a CNN on an edge platform is a complex task requiring in-depth knowledge in the fields of Deep Learning (DL) and Embedded Systems Design. Fortunately, this task can be greatly simplified through the use of DL frameworks such as Keras [19], Pytorch [75], Tensorflow [1], TensorRT [72] and others [74]. These frameworks provide a highly abstract user-friendly API for implementation and deployment of CNNs at the Edge together with a library of highly optimized operators performed by the CNN layers. The deployed CNN is ready for its *inference* phase, at which the CNN performs its respective task on the real-world data collected at the Edge.

1.4 Limitations of the state-of-the-art design flow for CNNs executed at the Edge

In this section, we highlight two limitations that exist in the design flow shown in Figure 1.3. Also, we show the negative impact of these limitations on the design of CNNs executed at the Edge.

1.4.1 Limitation 1

The first limitation concerns the search for design points performed by the CNN optimization engine. As mentioned in Section 1.3, the CNN optimization engine explores CNNs with different architectures and weights and tries to find CNNs that are optimal in terms of the characteristics introduced in Section 1.1. To estimate the characteristics of the CNNs, the CNN optimization engine relies on the CNN characteristics estimation component. At this point, the CNN characteristics estimation component and the CNN optimization engine adopt **Limitation 1: a CNN is assumed to be executed sequentially, i.e., layer-by-layer**. This sequential manner of CNN execution is widely accepted by the DL frameworks [1, 19, 72, 74, 75] and is often used to execute CNNs. Nonetheless, layer-by-layer execution is not guaranteed to be optimal with respect to every edge platform and every set of requirements posed on a CNN. Recent works [22, 44, 45, 48, 50, 101, 110] show that there are alternative (non-sequential) manners to execute a CNN at the Edge. Moreover, these works show that a CNN may have better characteristics when executed in an alternative manner than when executed layer-by-layer. However, alternative manners of CNN execution are not explored by the CNN optimization engine. Thus, **due to Limitation 1, the existing methodologies for designing CNNs, executed at the Edge, may miss optimal design points**. We illustrate this in Figure 1.4 where we show three example CNNs, characterized with accuracy and throughput, and associated with two manners of CNN

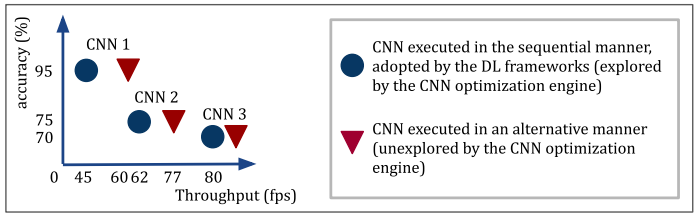


Figure 1.4: *CNNs associated with alternative manners of execution*

execution: 1) the sequential manner, accepted by the DL frameworks and assumed by the CNN optimization engine (shown as round points); 2) an alternative (non-sequential) manner, optimal with respect to the target edge platform and requirement of high throughput, posed on the CNNs (shown as triangle points). The accuracy of the CNNs does not depend on the manner the CNNs are executed, and therefore the accuracy is the same for a round point and a triangle point, associated with the same CNN. For example, the accuracy of $CNN\ 1$ is 95% irrespective of the manner $CNN\ 1$ is executed. The throughput of the CNNs is higher (i.e., better) when a CNN is executed in the non-sequential manner, optimal with respect to the target edge platform and requirements posed on the CNN - see the triangle points in Figure 1.4. Thus, these CNNs have better characteristics (same accuracy and better throughput) when they are executed in the non-sequential manner (triangle points), than when they are executed in the sequential manner (round points). However, due to **Limitation 1** mentioned above, the triangle points are missed by the CNN optimization engine.

1.4.2 Limitation 2

The second limitation concerns the selection of the final CNN from the set of design points, performed by the selection component shown in Figure 1.3. **Limitation 2** is formulated as follows: **currently, from the set of design points provided by the CNN optimization engine, only one design point (CNN) is selected to perform the required task in a CNN-based application**. With respect to the posed requirements, the selected CNN is characterized with certain accuracy and platform-aware characteristics that remain unchanged during the CNN-based application run-time. As a consequence, **the needs of CNN-based applications, affected by changes in the application environment during run-time, cannot be optimally served**. To illustrate this we give an example of a CNN-based road traffic monitoring application [53], which needs vary at the application run-time. The example application is executed on a drone. While flying, the drone takes images of the road and performs CNN-based recognition on these images. If there is a car accident or a traffic jam, the drone reports to the human operator. Depending on the situation on the roads and the level of the drone battery, the example application poses different requirements on the CNN. If the traffic is heavy, the application requires the CNN to have high throughput and high accuracy to process its input data, which typically means high energy consumption. During a traffic jam, when the high throughput is not required, or in case the battery of the drone is running low, the application would function optimally if the CNN

has reduced energy consumption at the cost of decreased throughput. If the example CNN-based application uses only one CNN to perform road traffic monitoring, it can either use a CNN with high throughput, high accuracy, and high energy cost, needed for a heavy-traffic application scenario, or use a CNN with reduced energy cost, as well as reduced accuracy and throughput. If the application uses a CNN with high throughput, high accuracy, and high energy cost, it optimally serves the application needs when the traffic is heavy, but does not optimally serve the application needs during a traffic jam or when the drone battery is low. Analogously, if the application uses a CNN with reduced energy cost, as well as reduced accuracy and throughput, it optimally serves the application needs during a traffic jam or when the drone battery is low, but not when the traffic is heavy. Thus, if the application uses only one CNN, the needs of the application cannot be optimally served during run-time in a changing application environment.

1.5 Research contributions

In this thesis, we try to relax the two limitations, outlined in Section 1.4, concerning the state-of-the-art CNN design flow shown in Figure 1.3. By relaxing the limitations, we try to reduce the negative impact of the limitations on the design of CNNs executed at the Edge. To this end, we extend the state-of-the-art CNN design flow shown in Figure 1.3 and explained in Section 1.3. The extended design flow is shown in Figure 1.5. The new components are shown in dark green. The extended design flow is one of our important research contributions. To realize the extended design flow, we propose other important research contributions (RC), summarized in Section 1.5.1, Section 1.5.2, Section 1.5.3, and Section 1.5.4, and denoted in Figure 1.5 as **RC 1**, **RC 2**, **RC 3**, and **RC 4**, respectively.

To relax **Limitation 1**, we extend the design flow with the *system-level optimization engine*. The system-level optimization engine accepts as an input the design points (CNNs), produced by the CNN optimization engine and assumed to be executed sequentially (layer-by-layer). The system-level optimization engine searches for alternative (non-sequential) manners to execute the input CNNs, thereby trying to find optimal design points missed by the CNN optimization engine. Along with the input CNNs, the system-level optimization engine accepts requirements posed on the CNNs and an edge platform model. The edge platform model which will be explained in details in Section 3.4 provides simplified, yet accurate description of a target edge platform to aid the search. As an output, the system-level optimization engine

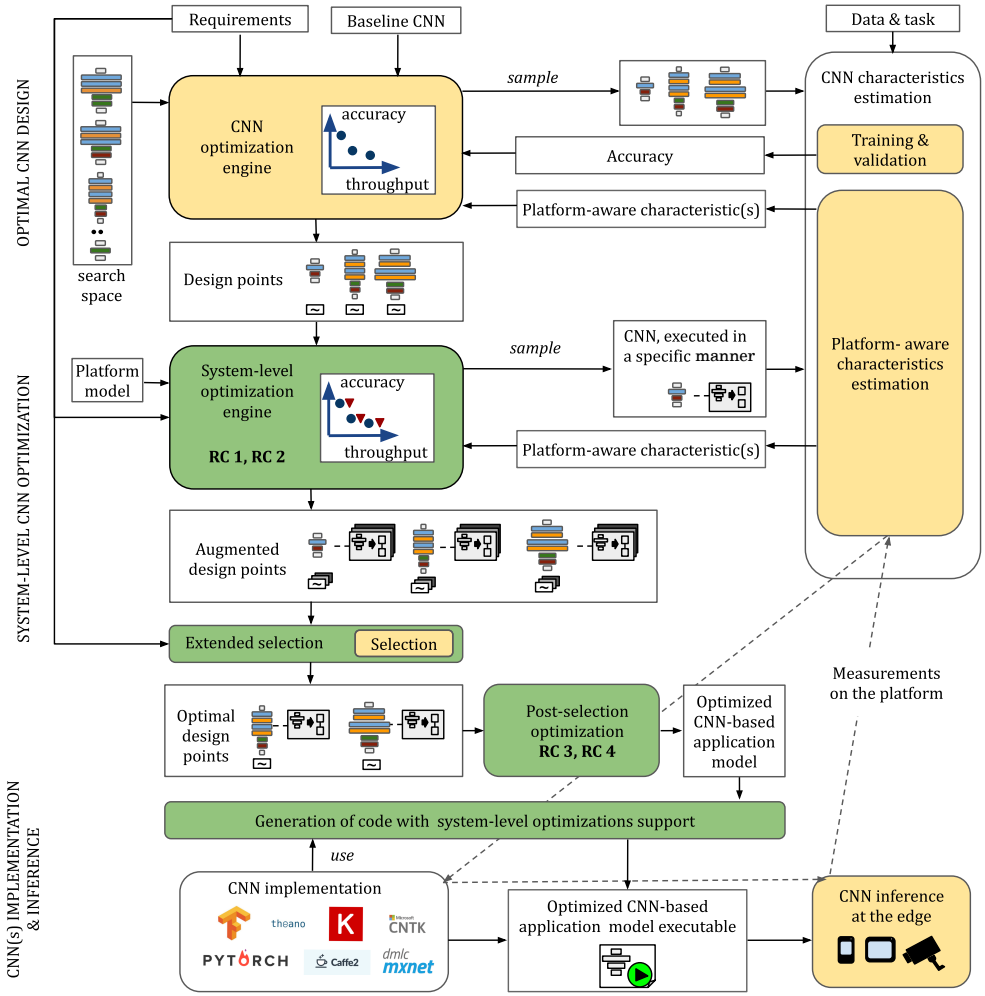


Figure 1.5: *Extended CNN design flow*

produces a set of *augmented design points* which contains the input CNNs associated with multiple alternative manners of execution. An example set of augmented design points is shown in Figure 1.4 and explained in Section 1.4. As shown in Figure 1.5, we place the system-level optimization engine after the CNN optimization engine. We note that the system-level optimization engine can also be placed within the CNN optimization engine. However, such positioning leads to a problem: it requires modifications of existing platform-aware NAS and CNN compression techniques and tools, used by the CNN optimization engine. Thus, it violates the principle of software architecture

modularity [80] and greatly complicates the reuse of existing platform-aware NAS and CNN compression techniques and tools. To avoid this problem, we place the system-level optimization engine after the CNN optimization engine. To realize the system-level optimization engine, in this thesis, we propose and utilize two novel methodologies that explore alternative manners of CNN execution: the methodology for high-throughput CNN inference, summarized in Section 1.5.1 and the methodology for low-memory CNN inference, summarized in Section 1.5.2. In Figure 1.5, the methodologies are denoted as research contributions **RC 1** and **RC 2**, respectively. It is worth noting that, while the two proposed methodologies are important for finding optimal design points, the capabilities of the system-level optimization engine are not limited to these methodologies. To enrich the performed system-level optimizations, the system-level optimization engine may integrate other complimentary methodologies such as methodologies proposed in [101] and [93].

To relax **Limitation 2**, we extend the design flow with the *extended selection* component and the *post-selection optimization* component. The extended selection component enables for selection of multiple pareto-optimal [18] design points (CNNs) along with the selection of the single optimal design point, offered by the (original) selection component. The *post-selection optimization* component determines how to optimally use multiple design points to best serve the needs of a CNN-based application. The post-selection optimization component introduces run-time adaptivity into a CNN-based application affected by changes in the application environment at run-time, and performs joint CNNs memory optimization of multiple design points (CNNs) used by a CNN-based application. As an output, the post-selection optimization component produces the final CNN-based application model which embeds the functionality of the CNNs used by the application as well as the system-level optimizations introduced into the application. To realize the post-selection optimization component, we propose and utilize two novel methodologies: the methodology for run-time adaptive inference of CNN-based applications, summarized in Section 1.5.3, and the methodology for joint memory optimization of multiple CNNs, summarized in Section 1.5.4. In Figure 1.5, the methodologies are denoted as research contributions **RC 3** and **RC 4**, respectively.

Finally, we extend the design flow with a component that performs *generation of code with system-level optimizations support*. The code generation component accepts as an input the optimized CNN-based application model, produced by the post-selection optimization component, and implements this model. We introduce the code generation component because the optimized

CNN-based application model cannot be implemented using only the DL frameworks that generate CNN-based application code in the state-of-the-art design flow shown in Figure 1.3. More precisely, the existing DL frameworks do not support the system-level optimizations (e.g., alternative manners of CNN execution and run-time adaptivity) embedded into the optimized CNN-based application model as explained above. Therefore, we extend the design flow with the code generation component which uses: 1) the existing DL frameworks to implement the CNNs functionality; 2) custom system-level design tools to implement the system-level optimizations. As an output, the code generation component provides an executable file with implementation of the input CNN-based application model. Although the code generation component is not presented as a separate research contribution in this thesis, it is used for implementation of the CNN-based applications and evaluation of the methodologies summarized in Section 1.5.1, Section 1.5.2, Section 1.5.3, and Section 1.5.4.

1.5.1 RC1: Methodology for high-throughput CNN inference

In this section, we summarize our novel methodology for high-throughput CNN inference at the Edge. The proposed methodology is based on our publication [67] and is explained in details in Chapter 3. The methodology exploits two types of parallelism, data-level parallelism and task-level parallelism, available in a CNN, to efficiently distribute (map) the computations within the CNN to the computational resources of an edge platform. The CNN distribution (mapping) is considered efficient if it ensures high CNN throughput. To find an efficient CNN mapping, our proposed methodology uses a Genetic Algorithm (GA) [85]. Exploitation of task-level (pipeline) parallelism together with data-level parallelism is the main novel feature of our proposed methodology. This feature distinguishes our methodology from the existing DL frameworks, introduced in Section 1.3, that utilize only task-level (pipeline) parallelism or only data-level parallelism, available in a CNN, to ensure high CNN throughput. Thanks to the combined use of task- and data-level parallelism, our proposed methodology takes full advantage of all computational resources that are available on the edge platform, and ensures very high CNN throughput. To evaluate our proposed methodology, we perform experiments where we apply our methodology to real-world CNNs from the Open Neural Network Exchange Format (ONNX) models zoo [7] and execute the CNNs on the NVIDIA Jetson TX2 edge platform [71]. We compare the throughput demonstrated by the CNNs mapped on the Jetson TX2 platform using: 1) our proposed methodology; 2) the best-known and state-of-the-art TensorRT DL

framework [72] for the Jetson TX2 edge platform. The experimental results shown that 1.36% to 42% higher throughput is achieved, when the CNNs are mapped using our methodology. We note that our proposed methodology considers edge platforms with computational resources composed of CPUs and GPUs because such platforms are most often used to execute applications, requiring high CNN throughput [32, 109]. However, extending our proposed methodology to other types of edge platforms (e.g., FPGA-based platforms [32]) is straightforward due to the modularity and generality of our methodology.

1.5.2 RC2: Methodology for low-memory CNN inference

In this section, we summarize our novel methodology for low-memory CNN inference at the Edge. The proposed methodology is based on our publication [65] and is explained in details in Chapter 4. To ensure low memory cost of the CNN inference, the methodology splits the data, processed by layers of a CNN, in parts and efficiently reuses the edge platform memory, allocated to store the data parts. Processing data by parts is the key novel feature of our proposed methodology. It enables our methodology to reduce the CNN-based application memory footprint without affecting the main CNN quality metric, i.e., the CNN accuracy. This compares favourably with the most common CNN memory reduction methodologies such as pruning and quantization [41, 99, 106] that reduce the CNN inference memory footprint at the cost of decreased CNN accuracy. However, data processing by parts may cause CNN execution time overheads (e.g., CNN layers may require time to switch among the data parts), leading to CNN throughput decrease. Thus, the proposed methodology reduces the amount of memory occupied by a CNN at the cost of reduced CNN throughput. The experimental results show that taking real-world CNNs from the ONNX models zoo [7] and applying our memory reduction methodology to these CNNs, the CNNs memory cost is reduced by 2.8% to 38% when compared to the memory reduction achieved by the state-of-the-art TensorRT DL framework [72].

1.5.3 RC3: Methodology for run-time adaptive inference of CNN-based applications

In this section, we summarize our novel methodology for run-time adaptive inference of CNN-based applications. The proposed methodology is based on our publication [64] and is explained in details in Chapter 5. The methodology enables to adapt a CNN-based application to changes in the application

environment during run-time. It is based on the concept of *scenarios* [15], widely used in embedded systems design. According to this concept, an application can have different internal operation modes, called *scenarios*, each with its own typical characteristics or/and functionality. During run-time, the application can switch among the scenarios, thereby adapting its characteristics or functionality to changes in the application environment. In our scenario-based run-time switching (SBRS) methodology, a scenario is a CNN designed to conform to a specific set of requirements in terms of accuracy and platform-aware characteristics. An application can have multiple scenarios that conform to different application needs. During run-time, the application can switch among the scenarios, thereby adapting its characteristics to its needs. To enable for run-time adaptivity, our SBRS methodology represents a CNN-based application as a novel SBRS model of computation (MoC) which embeds the functionality of the application scenarios as well as mechanisms for run-time adaptivity. Additionally, the methodology proposes an SBRS *transition protocol* which ensures high application responsiveness during the scenarios switching. The experimental results, where we apply our methodology to three real-world applications from two different domains, show that our SBRS methodology: 1) Adapts a CNN-based application to changes in the environment, thereby ensuring optimal service to the needs of the application at any given point in time; 2) Enables for fast switching between the application scenarios due to our novel SBRS transition protocol; 3) Outperforms the most relevant existing methodology called MSDNet [39].

1.5.4 RC4: Methodology for joint memory optimization of multiple CNNs

In this section, we summarize our novel methodology for joint memory optimization of multiple CNNs. The proposed methodology is based on our publication [66] and is explained in details in Chapter 6. As mentioned earlier, to relax Limitation 2, our extended design flow allows a CNN-based application to use multiple CNNs instead of one CNN to perform its functionality. However, this may dramatically increase the application memory cost, while as explained in Section 1.2, low memory cost is required for CNN-based applications executed at the Edge. Thus, execution of a multi-CNN application (an application using multiple CNNs) at the Edge may require aggressive optimizations to reduce the application memory cost. Typically, these optimizations are performed using methodologies such as pruning and quantization [41,99,106]. These methodologies reduce the number or precision of CNN parameters, thereby reducing the CNN memory cost. However, at

high CNN memory reduction rates, these methodologies decrease the CNN accuracy, while as mentioned above, high CNN accuracy is very important for many CNN-based applications. To achieve high CNN memory reduction and avoid substantial decrease of CNN accuracy, the CNN pruning and quantization methodologies can be combined with the CNN memory reuse methodologies such as the methodologies in [47] and [76]. Orthogonal to the pruning and quantization methodologies, the CNN memory reuse methodologies reuse the platform memory allocated to store intermediate CNN computational results, produced by the CNN layers. Thus, these methodologies further reduce the application memory cost without decreasing the CNN accuracy. However, these methodologies account for the state-of-the-art CNN design flow shown in Figure 1.3, and thus adopt Limitation 1 and Limitation 2, outlied in Section 1.4. Due to Limitation 1, these methodologies do not account for non-sequential manners of CNN execution, and are often unfit for CNNs executed in a non-sequential manner. Due to Limitation 2, these methodologies can reuse platform memory within a CNN, but not among multiple CNNs, thereby missing opportunities for inter-CNN memory reuse. To address these issues, we propose our methodology for joint memory optimization of multiple CNNs. Unlike the existing memory reuse methodologies, our proposed methodology reuses memory among multiple CNNs, and is suitable for CNNs executed in a non-sequential manner. To evaluate our proposed methodology, we perform experiments where we apply our methodology to six real-world state-of-the-art CNN-based applications. The experimental results show that our methodology enables for up to 6 times memory reduction, compared to deployment of CNN-based applications with no memory reduction and 10% to 30% memory reduction, compared to other CNN memory reuse methodologies. Additionally, the experimental results demonstrate that our methodology can be efficiently combined with orthogonal methodologies such as CNN pruning and quantization.

1.6 Thesis organization

Below, we give an outline of this thesis, summarizing the contents of the following chapters. **Chapter 2** presents the background, i.e., concepts necessary to understand the contributions of this thesis. Chapter 3 to Chapter 6 contain the main contributions of this thesis. Each chapter is organized in a self-containing manner, meaning that each chapter contains a more specific introduction to the research problem and contribution, a related work, the proposed solution methodology, an experimental evaluation, and a concluding

discussion.

Chapter 3 presents our novel methodology for high-throughput CNN inference. This chapter is based on our publication [67].

Chapter 4 presents our novel methodology for low-memory CNN inference. This chapter is based on our publication [65].

Chapter 5 presents our novel methodology for run-time adaptive inference of CNN-based applications. This chapter is based on our publication [64].

Chapter 6 presents our novel methodology for joint memory optimization of multiple CNNs. This chapter is based on our publication [66].

Finally, **Chapter 7** ends this thesis by providing a summary and concluding remarks for the research work presented in this thesis.

Chapter 2

Background

In this chapter, we present an overview of concepts essential to understand the contributions of this thesis. In Section 2.1, we present the CNN model used to represent a CNN in this thesis. In Section 2.2, we describe the CNN deployment and inference at the Edge, briefly introduced in Section 1.3 because in this thesis, we study and propose novel methodologies for efficient CNN deployment and inference at the Edge. In Section 2.3, we introduce a typical edge platform used to execute CNNs inference. Namely, we introduce the well-known and state-of-the-art NVIDIA Jetson TX2 platform [71], used to perform experiments in this thesis. In Section 2.4, we explain the task- and data-level parallelism available in a CNN. We exploit the aforementioned types of parallelism to ensure efficient inference of CNNs at the Edge. In Section 2.5, we briefly describe the Cyclo-Static Data Flow (CSDF) [10] and the Synchronous Data Flow (SDF) [57] models of computation, widely used in the Embedded Systems community to represent applications executed at the Edge. Unlike the CNN model, introduced in Section 2.1, the SDF model and the CSDF model explicitly specify the parallelism, available within an application (or a part of an application such as a CNN), and enable for modelling of various manners of application execution. In this thesis, we use the CSDF model and the SDF model to represent an augmented design point (i.e., a CNN, executed in a specific manner) briefly introduced in Section 1.5. Finally, in Section 2.6, we describe the basic concepts of a Genetic Algorithm (GA): a well-known heuristic approach, widely used for finding optimal solutions for complex Design Space Exploration (DSE) problems. Some of the methodologies, presented in this thesis, are based on a GA.

2.1 CNN model

A Convolutional Neural Network (CNN) is commonly represented as a directed acyclic computational graph $CNN(L, E)$ with a set of nodes L, also called layers, and a set of edges E. An example of a CNN model with a set of layers $L = \{l_1, l_2, l_3, l_4, l_5\}$ and a set of edges $E = \{e_{12}, e_{23}, e_{34}, e_{45}\}$ is shown in Figure 2.1.

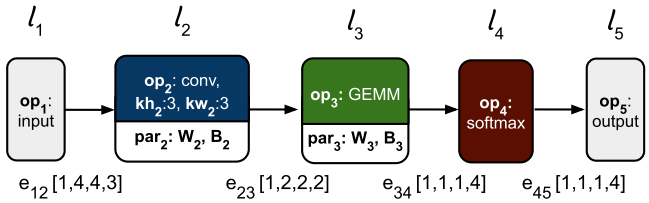


Figure 2.1: *CNN model*

The CNN model specifies transformations over the CNN input data (e.g. an image), that result into the CNN output data (e.g. an image classification result). The transformations are specified by the set of layers L. Edges in the set E specify data dependencies between the layers and determine the flow of data in a CNN. The detailed explanation and formal definition of a layer $l_i \in L$ and an edge $e_{ij} \in E$ of the CNN model are given in Section 2.1.1 and Section 2.1.2, respectively.

2.1.1 Layer in the CNN model

Every layer l_i in the CNN model represents part of the CNN functionality. It accepts as an input some data, produced by other layers, transforms this data using a mathematical operator, and provides output data. Formally, we define layer l_i as a set of attributes, summarized in Table 2.1. Column 1 lists the attributes; Column 2 provides a description of each attribute; Column 3 lists limitations, posed on the attribute by the CNN model; Column 4 shows the *default* value of an attribute, i.e., the value assigned to the attribute which is not defined explicitly. We note that some of the attributes (e.g., attributes I_i and O_i shown in Rows 4 to 5 in Table 2.1) only take the default value. Below, we explain the attributes of layer l_i, summarized in Table 2.1, using as an example layer l_2 shown in Figure 2.1.

Attributes $type_i$ and op_i (Rows 2 to 3 in Table 2.1) specify the type and performed operator of layer l_i, respectively [4]. These attributes determine the main difference between the layers of a CNN. The most common types of

Table 2.1: *Attributes of layer l_i*

attribute	description	limitations	default value
$type_i$	layer type	supported by the CNN model (see Table 2.2)	for known op_i see Table 2.2
op_i	operator	restricted by $type_i$ (see Table 2.2)	-
I_i	input edges	$I_i \subseteq E : \forall e_{ji} \in E, e_{ji} \in I_i$	
O_i	output edges	$O_i \subseteq E : \forall e_{ij} \in E : e_{ij} \in O_i$	
X_i	input data	see Equation 2.1	
Y_i	output data	see Equation 2.2	
Θ_i	sliding window	has smaller or equal size compared to X_i	window of size kh_ix kw_i
kh_i	kernel height	$0 < kh_i \leq X_i.h$; typically $kh_i = kw_i$; $kh_i = X_i.h$ if $type_i \in$*{data,FC}*	$X_i.h$ if $type_i \in$*{data,FC}*, else 1
kw_i	kernel width	$0 < kw_i \leq X_i.w$; typically $kw_i = kh_i$; $kw_i = X_i.w$ if $type_i \in$*{data,FC}*	$X_i.w$ if $type_i \in$*{data,FC}*, else 1
s_i	stride	$s_i = 1$ if $op_i \notin$ *{conv, max pool, average pool}*	1
pad_i	padding	an array of four integer numbers	[0,0,0,0]
par_i	(trainable) parameters	a set of parameters, specific for CNN layer [4]	$\emptyset$

Table 2.2: *Most common CNN layer types and operators*

layer type	operators
convolutional	*conv*
pooling	*(global) max pool, (global) average pool*
activation	*ReLU, thn, sigmoid*
data	*input, output*
fully connected (FC)	*GEMM, MatMUL, dot*
loss	*softmax*
normalization	*BatchNormalization, LRN*
arithmetic	*add*
transformation	*concat*

layers and operators performed by layers of these types are shown in Table 2.2. For example, layer l_2 shown in Figure 2.1 has $type_2 = convolutional$ and performs operator $op_2 = conv$. Operator op_2 performed by layer l_2 is explicitly specified in Figure 2.1, thus the type of layer l_2 is determined using Table 2.2.

Attributes I_i and O_i (Rows 4 to 5 in Table 2.1) specify the input and output edges of layer l_i, respectively. For example, layer l_2 shown in Figure 2.1 has input edges $I_2 = \{e_{12}\}$ and output edges $O_2 = \{e_{23}\}$.

Attributes X_i and Y_i (Rows 6 to 7 in Table 2.1) specify the input and output data of layer l_i, respectively. These attributes always take the default value, computed using Equation 2.1 and Equation 2.2.

$$X_i = \begin{cases} e_{ji}.data : e_{ji} \in I_i & \text{if } |I_i| = 1 \\ \{e_{ji}.data\}, \forall e_{ji} \in I_i & \text{otherwise} \end{cases} \tag{2.1}$$

$$Y_i = \begin{cases} e_{ij}.data : e_{ij} \in O_i & \text{if } |O_i| > 0 \\ \emptyset & \text{otherwise} \end{cases} \tag{2.2}$$

The value of attribute X_i is computed using Equation 2.1, where $e_{ji}.data$ is the data accepted by layer l_i and associated with input edge $e_{ji} \in I_i$ of layer l_i; $|I_i|$ is the total number of input edges of layer l_i. Typically, layer l_i has one input edge, i.e., $|I_i| = 1$. In this case, input data X_i of layer l_i is the data $e_{ji}.data$, associated with the only input edge e_{ji} of layer l_i. For example, layer l_2 shown in Figure 2.1 has one input edge e_{12}, and has input data $X_2 = e_{12}.data$. However, some layers may accept as an input data coming from multiple input edges (e.g., layers performing operator *concat* [4]) or accept no input data (e.g., layers performing the operator *input* [4]). Layers that accept no input data have $|I_i| = 0$ and $X_i = \emptyset$.

Analogously, the value of attribute Y_i is computed using Equation 2.2, where $e_{ij}.data$ is the data produced by layer l_i and associated with output edge $e_{ij} \in O_i$ of layer l_i; $|O_i|$ is the total number of output edges of layer l_i. Typically, layer l_i has at least one output edge and produces data $Y_i \neq \emptyset$, broadcasted to every output edge of layer l_i. For example, layer l_2 shown in Figure 2.1 produces output data $Y_2 = e_{23}.data$ onto its output edge e_{23}. However, some layers (e.g., layers performing the operator *output* [4]) do not produce data. These layers have $|O_i| = 0$ and $Y_i = \emptyset$.

Attributes Θ_i, kh_i, kw_i, s_i, and pad_i (Rows 8 to 12 in Table 2.1) are the *hyper-parameters* of layer l_i [4]. These attributes, obtained during the CNN design, specify how the layer processes its input data. To process its input data X_i, layer l_i moves along X_i with sliding window Θ_i and stride s_i, applying operator op_i to the area of X_i, covered by Θ_i. The sliding window Θ_i has

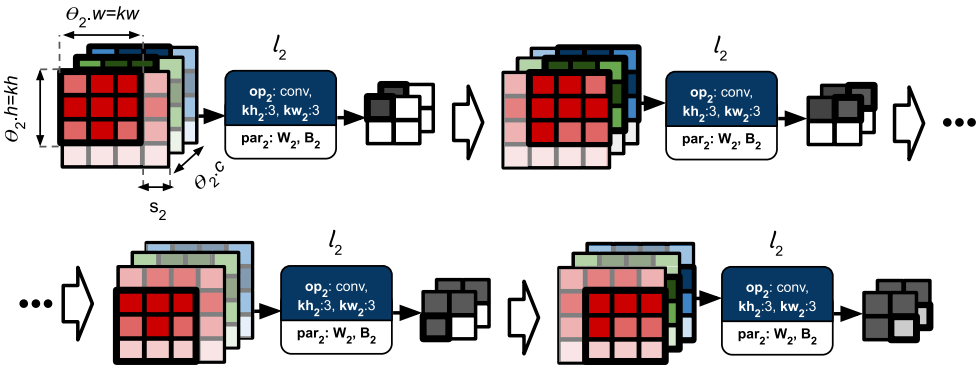


Figure 2.2: *Processing of input data X_2 by layer l_2*

smaller or equal size, compared to X_i. The height and width of window Θ_i are typically equal to the kernel height kh_i and kernel width kw_i of layer l_i, while the number of channels of Θ_i is typically equal to the number of channels of X_i [4]. The areas, covered by Θ_i, can overlap. Figure 2.2 shows an example, where layer l_2 shown in Figure 2.1 processes its input data by four parts, covered by sliding window Θ_2 of size 3 x 3 x 3 pixels, and stride $s_2 = 1$ pixel.

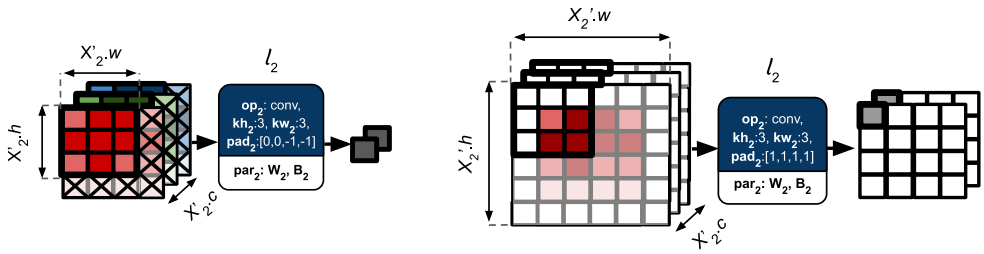
Before processing its input data, layer l_i may crop or extend its input data X_i to data X'_i with *padding* [4]. Typically, this is done to ensure that the input data of layer l_i can be covered by sliding window Θ_i of layer l_i integer number of times [4]. We specify the padding of layer l_i as attribute pad_i (Row 12 in Table 2.1). pad_i is an array of four integer numbers. Elements of pad_i, referred as $pad_i[0]$, $pad_i[1]$, $pad_i[2]$, and $pad_i[3]$, respectively, specify the crop/extension of the height and width of data X_i as given in Equation 2.3 and Equation 2.4, respectively.

$$X'_i.w = pad_i[0] + X_i.w + pad_i[2] \tag{2.3}$$

$$X'_i.h = pad_i[1] + X_i.h + pad_i[3] \tag{2.4}$$

By default, layer l_i has $pad_i = [0,0,0,0]$, which means that layer l_i does not crop or extend its input data X_i before processing. Figure 2.3 shows an example where layer l_2 crops (see Figure 2.3 (a)) and extends (see Figure 2.3 (b)) its input data X_i with padding pad_2 = [0,0,-1,-1] and pad_2 = [1,1,1,1], respectively.

Beside the hyper-parameters, layer l_i has (trainable) parameters such as weights and biases [4], specified as attribute par_i (Row 13 in Table 2.1). As mentioned in Chapter 1, these parameters of layer l_i are obtained during the



(a) *crop* **(b)** *extension*

Figure 2.3: *Padding*

CNN training and are used by operator op_i of layer l_i. For example, layer l_2 has parameters par_2 composed of weights W_2 and biases B_2, used to perform $op_2 = conv$.

2.1.2 Edge in the CNN model

Every edge $e_{ij} \in E$ in the CNN model specifies a data dependency between layers l_i and l_j of a CNN, such that the data produced by layer l_i is accepted as an input by layer l_j. Formally, we define edge e_{ij} as a tuple $(l_i, l_j, data)$, where $data$ is the data produced by layer l_i, accepted by layer l_j, and associated with edge e_{ij}. The data associated with edge e_{ij} is stored in a multidimensional array called tensor [4]. In this thesis, every data tensor has the shape $[batch, h, w, ch]$, where $batch, h, w, ch$ are the batch size [4], the height, the width, and the number of channels of the tensor, respectively. An example of edge $e_{12} = (l_1, l_2, data)$ is shown in Figure 2.1. Edge e_{12} represents the data dependency between layers l_1 and l_2, where layer l_2 accepts as an input the data produced by layer l_1. Edge e_{12} is annotated with shape [1,4,4,3]. This means that the data tensor, exchanged between layers l_1 and l_2, and associated with edge e_{12} has batch size = 1, height and width = 4, and number of channels = 3.

2.2 CNN deployment and inference at the Edge

The CNN inference is a process of applying the CNN to real-world data (e.g., images) and obtaining the CNN output (e.g., results of the input images classification). Nowadays, the CNN inference can be performed on a wide variety of hardware platforms. In this thesis, we concentrate on the CNN inference performed on edge (mobile and embedded) platforms, presented in Section 2.3.

Before the CNN inference can start, the CNN is *deployed* on a target platform, i.e., some memory of the platform is allocated to the CNN. The total amount of memory (in bytes), allocated to a CNN is computed as:

$$m = m_{par} + m_{buf} \tag{2.5}$$

where m_{par} is the memory, required to store the CNN parameters (weights and biases) and computed using Equation 2.6; m_{buf} is the memory, required to store the CNN intermediate computational results and computed using Equation 2.7.

$$m_{par} = \sum_{i \in [1,|L|]} |par_i| * par\_size \tag{2.6}$$

In Equation 2.6, $|par_i|$ is the total number of parameters, associated with layer $l_i \in L$ of the CNN; *par_size* is the size of one parameter in bytes;

$$m_{buf} = \sum_{B_k \in B} B_k.size \tag{2.7}$$

In Equation 2.7, B is a set of *buffers*, i.e., the memory segments, allocated to store the intermediate computational results of a CNN [76]. Every buffer $B_k \in B$ has one or several CNN edges e_{ij} allocated to it. Buffer B_k stores data $e_{ij}.data$, exchanged between CNN layers l_i and l_j during the CNN inference and is characterized with size (in bytes) computed as:

$$B_k.size = max_{e_{ij} \in B_k.edges}\{|e_{ij}.data| * token\_size\} \tag{2.8}$$

In Equation 2.8, $e_{ij} \in B_k.edges$ is an edge, storing data in buffer B_k; $|e_{ij}.data|$ is the total number of data elements (tokens), exchanged through edge e_{ij}; *token_size* is the size of one token in bytes.

A CNN deployed on an edge platform can start its inference phase when the CNN can utilize the memory and the computational resources available on the platform to perform the CNN functionality, i.e., to execute all the layers in the CNN. Every layer can be executed on processors, such as CPUs, GPUs and/or FPGAs [17], available in the platform. If a platform has parallel processors (accelerators), such as GPUs or FPGAs, computations within the layer can be represented as one or multiple kernels [17] and offloaded on these accelerators by the CPUs. Otherwise, these computations are performed on the CPUs. If the computations within a CNN layer are offloaded on an accelerator with local memory, e.g., a GPU, the CNN layer input data and parameters, required to perform the computations, are copied from the main memory into

the local memory of the accelerator and the results of the computation are copied back to the main memory.

The layers of a CNN are executed in a specific order, determined by the data dependencies within the CNN and the manner the CNN is executed. Typically, the CNN layers are executed in a *sequential manner*, where the CNN execution is represented as $|L|$ computational steps and at every i-th computational step, CNN layer $l_i \in L$ is executed. However, as it will be explained in Section 2.4, a CNN can also be executed in *alternative (non-sequential) manners*, that involve exploitation of task-level (pipeline) parallelism, where the layers of the CNN are executed in parallel (pipelined) fashion. In this thesis, we consider both sequential and non-sequential manners of CNN execution. To represent a CNN, executed in a specific manner, we use the CSDF and SDF models of computation, presented in Section 2.5.

2.3 Edge platform used for CNN inference

Modern edge platforms used to execute the CNN inference are complex systems that host a large number of specific hardware components: processors, memory, power supply elements, sensors and others [32, 109]. Figure 2.4 shows a simplified structure of the NVIDIA Jetson TX2 edge platform [71]: one of the best-known edge platforms used to execute CNNs.

To perform computations within a CNN, an edge platform may host multiple heterogeneous processors such as central processing units (CPUs), graphics processing units (GPUs), field-programmable gate arrays (FPGAs), and/or Tensor Processing Units (TPUs) [32, 109]. For example, the Jetson TX2 platform shown in Figure 2.4 hosts a double-core Denver 2 CPU and a

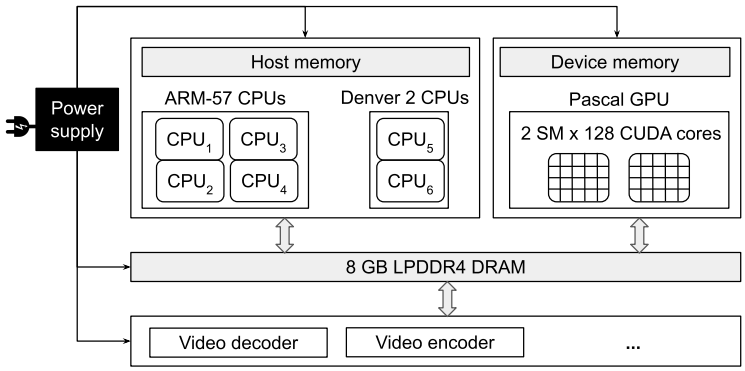


Figure 2.4: *Jetson TX2 edge platform*

quad-core ARM Cortex A57 CPU as well as an integrated Pascal GPU with a total of 256 CUDA cores. When the inference of a CNN is executed on the Jetson TX2 platform, computations within the CNN are typically performed on the GPU.

The memory infrastructure of an edge platform is used to store the CNN data and parameters, required for proper CNN inference. It typically consists of a *main memory*, accessible by all processes available on the platform, and a set of *local memories*, only accessible by specific processor(s). For example, the memory infrastructure of the Jetson TX2 platform shown in Figure 2.4 consists of the 8 GB LDDR4 DRAM, accessible by all the processors, available on the platform, as well as the *host_memory* and *device_memory*, i.e., the local memories, accessible only by the CPUs and the GPU, respectively.

The power supply elements of an edge platform provide power to all components available on the platform. Some edge platforms carry batteries that provide an autonomous limited power supply to the edge device. The Jetson TX2 platform, however, does not have a battery and requires an external power supply.

Finally, other components, available on the platform, e.g., video encoders and decoders, are used to collect data and facilitate parts of a CNN-based application other than the CNN itself.

2.4 Task- and data-level parallelism available in a CNN

As a computational model the CNN model is characterized with large amount of available parallelism. This parallelism can be exploited to speed-up the CNN inference and to efficiently utilize computational resources of an edge platform, where the CNN is executed.

The most widely exploited type of parallelism available within CNNs is the *data-level parallelism*, illustrated in Figure 2.5. This type of parallelism involves the same computation, e.g., Convolution, performed by a CNN layer over the CNN layer input data partitions. Efficient utilization of data-level parallelism allows to speed-up the inference of a CNN by accelerating the execution of individual CNN layers. This type of parallelism is exploited by the majority of existing Deep Learning (DL) frameworks, such as Keras [19], Pytorch [75], Tensorlow [1], TensorRT [72] and others [74]. The data-level parallelism, available within layer l_i of a CNN can be explicitly expressed by decomposition of the layer input data tensor X_i into a set of K sub-tensors $\{X_{i1}, X_{i2}, ..., X_{iK}\}$, where: 1) all sub-tensors $X_{ik}, k \in [1, K]$ can be processed in parallel by operator op_i. When layer l_i applies operator op_i to X_{ik}, it produces

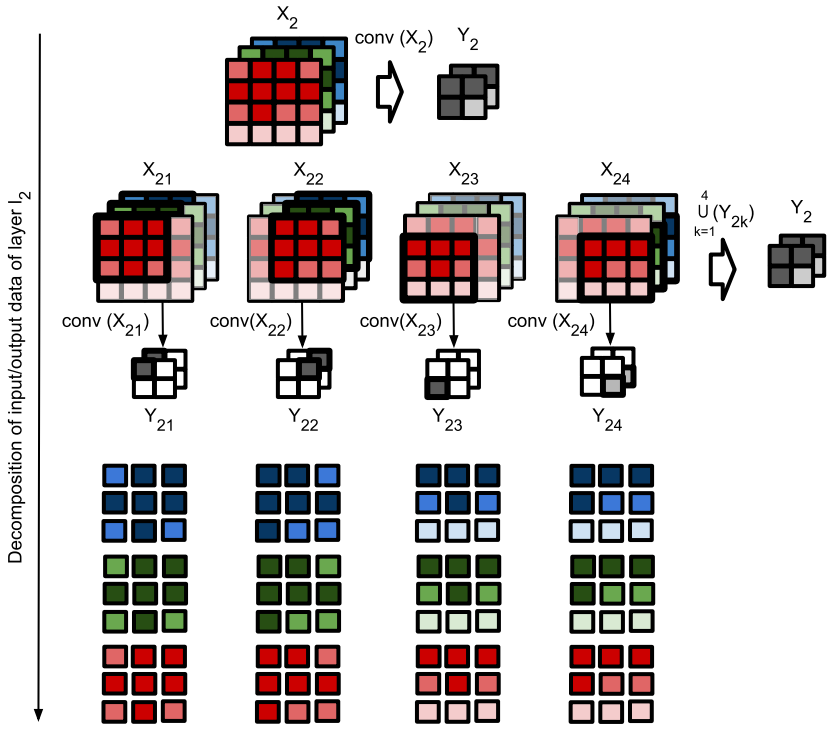


Figure 2.5: *data-level parallelism*

sub-tensor Y_{ik} of output data Y_i; 2) elements (pixels) within every X_{ik} can be processed in parallel. Figure 2.5 illustrates the data-level parallelism, available within convolutional layer l_2, shown in Figure 2.1 and explained in Section 2.1. In Figure 2.5, input data tensor X_2 of layer l_2 is decomposed into $K = 4$ overlapping sub-tensors $X_{2k}, k \in [1, 4]$ that can be processed in parallel. When layer l_2 processes sub-tensor X_{2k} with operator $op_2 = conv$, it produces sub-tensor Y_{2k} of output data Y_2, such that $Y_2 = \cup_{k=1}^{4} Y_{2k}$. Every sub-tensor X_{2k} shown in Figure 2.5 is subsequently decomposed into a set of pixels, where every pixel can be processed in parallel.

Another type of parallelism available in a CNN is known as *task-level parallelism* or *pipeline parallelism* [67, 101] among the CNN layers. This type of parallelism is related to the streaming nature of a CNN-based application, where the application accepts different input frames (images) from an input data stream. When a CNN is executed on a platform with multiple processors, the frames from the input data stream can be processed in a pipelined fashion by different layers of the CNN deployed on different processors.

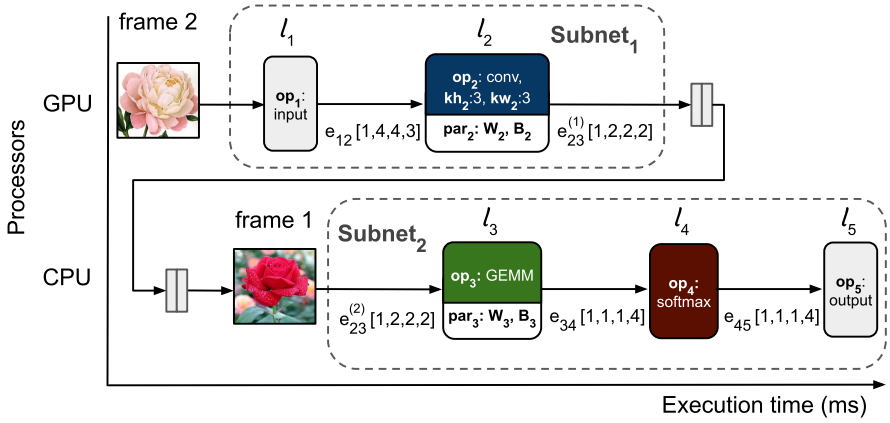


Figure 2.6: *task-level (pipeline) parallelism*

Figure 2.6 shows an example where the CNN shown in Figure 2.1 and explained in Section 2.1 is executed in a pipelined fashion on a platform with two processors: a CPU and a GPU. The layers of the CNN, representing computations within the CNN, are distributed over the platform processors: layers l_1 and l_2 are executed on the GPU, while layers l_3, l_4, and l_5 are executed on the CPU. The distributed layers form two CNN sub-graphs also referred as *partitions* [67, 101], annotated as $Subnet_1$ and $Subnet_2$. Partition $Subnet_1$ accepts frames from the application input data stream, processes these frames as specified by layers l_1 and l_2 and stores the results into a buffer associated with edge e_{23}. Partition $Subnet_2$ accepts the frames processed by partition $Subnet_1$ from edge e_{23}, further processes these, frames and produces the output data of the example CNN. Partitions $Subnet_1$ and $Subnet_2$ are executed on different processors in the platform and do not compete for the platform computational resources. Thus, when applied to different data (i.e., different frames), the partitions can be executed in parallel. In Figure 2.6, partitions $Subnet_1$ and $Subnet_2$ process frames *frame 2* and *frame 1* in parallel. This leads to overlapping execution of layers belonging to different partitions and enables for faster inference of the CNN, compared to conventional layer-by-layer (sequential) execution. However, pipelined CNN execution involves memory overheads. As shown in Figure 2.6, edge e_{23} of the example CNN is duplicated between the partitions $Subnet_1$ and $Subnet_2$ (see edges $e_{23}^{(1)}$ and $e_{23}^{(2)}$ and the corresponding buffers). Such duplication, called the double-buffering [37], is necessary for execution of the CNN as a pipeline. It prevents competition for memory (buffers) between the partitions when accessing data associated with edge e_{23}. If the double buffering is not enabled the CNN partitions compete

for access to edge e_{23}, thereby creating stalls in the pipeline and reducing the CNN throughput.

It is worth noting that the parallelism available in a CNN is not explicitly specified in the CNN model, introduced in Section 2.1. The number of parallel tasks, executed to perform the CNN model functionality, and the exact communication and synchronization mechanisms between these tasks are internally determined by the utilized DL framework, and can vary for different frameworks. For example, the well-known DL frameworks [1,75] represent the functionality of every CNN layer l_i as multiple tasks, where the total number of tasks depends on the number of CPUs available on the target edge platform. The frameworks [94,101] represent the functionality of the same layer l_i as one task or part of a task. Therefore, the task-level parallelism is not explicitly specified in the CNN model. Analogously, the data-level parallelism is not explicitly defined in the CNN model because the number of input/output data sub-tensors K, the number of elements within sub-tensors X_{ik} and Y_{ik}, and other decomposition parameters are determined by every design framework individually, can vary for different frameworks, and even within one framework. For example, the TensorRT framework [72] is capable of representing the *conv* operator as: 1) the $GEMM$ operator, so for every Convolutional layer $K = 1$; 2) a direct convolution where $K >= 1$ is computed from the attributes of a layer, performing the *conv* operator.

2.5 CSDF and SDF models of computation

The CSDF model [10] is a well-known dataflow model of computation, widely used for model-based design in the embedded systems community. An application modelled as a CSDF is a directed graph $G(A, C)$ with set of nodes A, also called actors, communicating with each other through a set of communication FIFO channels C. Figure 2.7 shows an example of a CSDF model $G(A, C)$, where $A = \{a_1, a_2, a_3, a_4, a_5\}$ and $C = \{c_{12}, c_{22}, c_{23}, c_{34}, c_{45}\}$.

Every actor $a_i \in A$ in the CSDF model represents a certain functionality of the application, modelled as a CSDF graph, and performs an execution sequence $F_i = [f_i(1), f_i(2), \cdots, f_i(P_i)]$ of length P_i, where $p \in [1, P_i]$ is called a phase of actor a_i. At every phase actor a_i executes function $f_i(((p-1) mod P_i) + 1)$. An example of CSDF actor a_2 is shown in Figure 2.7. Actor a_2 performs execution sequence $F_2 = [conv, conv]$, shortly written as $[2 * conv]$, meaning actor a_2 has $P_2 = 2$ phases and performs function $f_2(p) = conv$ at each of its phases $p \in [1, 2]$.

Every FIFO communication channel $c_{ij} \in C$ represents a data dependency

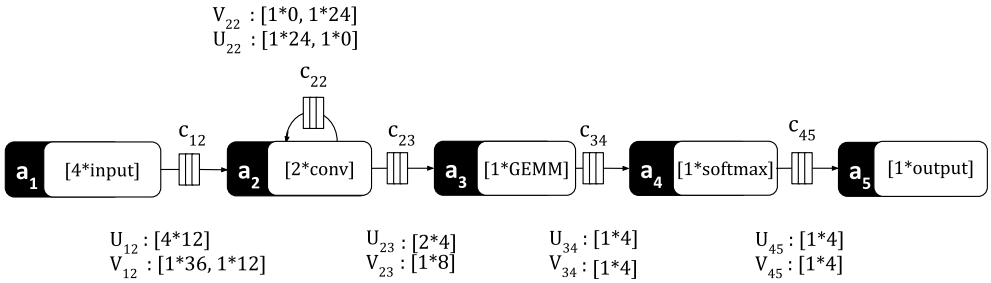


Figure 2.7: *CSDF model of computation*

and transfers data in tokens between its source actor a_i and its sink actor a_j. Every $c_{ij} \in C$ has production sequence U_{ij} and consumption sequence V_{ij}. Production sequence $U_{ij} : [u_{ij}(1), u_{ij}(2),..., u_{ij}(P_i)]$ of length P_i specifies the production of data tokens into channel c_{ij} by its source actor a_i. Analogously, consumption sequence $V_{ij} : [v_{ij}(1), v_{ij}(2),..., v_{ij}(P_j)]$ of length P_j specifies the consumption of data tokens from channel c_{ij} by its sink actor a_j. An example of communication channel c_{22} is shown in Figure 2.7. For communication channel c_{22}, actor a_2 is a source and a sink actor. The production sequence $U_{22} : [24, 0]$, formally written as $U_{22} : [1 * 24, 1 * 0]$ specifies, that at phase $p = 1$ actor a_2 produces 24 tokens to channel c_{22}, and at phase $p = 2$ actor a_2 produces 0 tokens to channel c_{22}. Analogously, the consumption sequence $V_{22} : [1 * 0, 1 * 24]$, specifies that during phase $p = 1$ actor a_2 consumes 0 tokens from channel c_{22}, and at phase $p = 2$ actor a_2 consumes 24 tokens from channel c_{22}.

A special case of the CSDF model, where every actor has only one phase, is called Synchronous Data Flow (SDF) model [57]. An example of a SDF model is shown in Figure 2.8.

At every firing, actor $a_i \in A$ of the SDF model consumes $v_{ki}(1)$ data tokens, executes function $f_i(1)$ and produces $u_{ij}(1)$ data tokens. For example, actor a_2 shown in Figure 2.8, at every firing consumes 48 data tokens from channel c_{12}, executes function $f_2(1) = conv$ and produces 8 data tokens to channel c_{23}. For simplicity, we omit the number of phases while annotating the execution

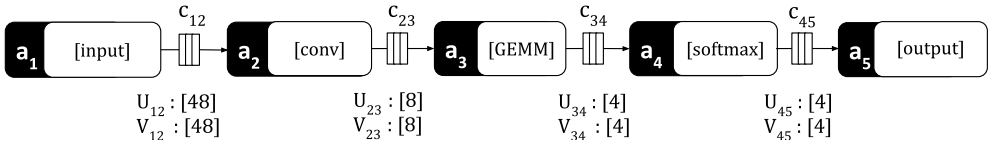


Figure 2.8: *SDF model of computation*

sequence, the production sequence, and the consumption sequence of the SDF model. For example, the consumption sequence of actor a_2 shown in Figure 2.8 is annotated simply as $[48]$ instead of $[1 * 48]$.

2.6 Genetic Algorithm (GA)

Genetic Algorithm (GA) [83] is a well-known heuristic approach, widely used for finding optimal solutions for complex Design Space Exploration (DSE) problems. In a GA, a population of candidate solutions to an optimization problem is evolved toward better solutions. A GA has two important problem-specific attributes: a chromosome and a fitness function.

A *chromosome* is a genetic representation of the solution. Typically, a chromosome is defined as a set of parameters (genes), joined into a string. An example of a chromosome is shown in Figure 2.9 This chromosome shows a distribution (mapping) of the computations within the layers of the CNN shown in Figure 2.1 and explained in Section 2.1 onto the computational resources of the Jetson TX 2 edge platform shown in Figure 2.4 and explained in Section 2.3. The chromosome shown in Figure 2.9 is a string of 5 genes, where every i-th gene, $i \in [1, 5]$, specifies a processor of the Jetson TX 2 platform which performs computations within layer l_i of the CNN. For example, the chromosome specifies that computations within layer l_1 of the CNN are performed on the GPU of the Jetson TX 2 platform.

A *fitness function* is a special function, which evaluates the quality of GA solutions, represented as chromosomes, and guides the GA-based search fro (pareto) optimal solutions. During the search, the fitness function should be minimized or maximized. For example, a fitness function can estimate the throughput of inference of the CNN, shown in Figure 2.1 and executed on the Jetson TX2 platform as specified in the chromosome shown in Figure 2.9.

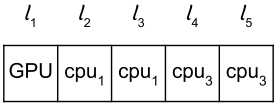


Figure 2.9: *Chromosome*

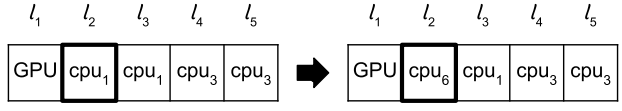


Figure 2.10: *Single-gene mutation*

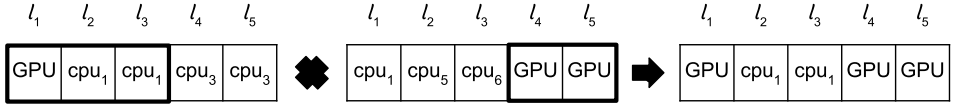


Figure 2.11: *Simple two-parent recombination (cross-over)*

If this fitness function is maximized during a GA-based search, it will guide the search towards finding chromosomes that ensure high CNN inference throughput.

Once the chromosome and the fitness function are defined, a GA can proceed to perform evolution, i.e., search for optimal solutions. The evolution is an iterative process. It starts from a population of randomly generated chromosomes. At each iteration, the fitness of every chromosome in the population is evaluated using the fitness function. Then, the chromosomes with the best score are selected from the population and are subjected to two genetic operators, called recombination (cross-over) and mutation. During the re-combination two selected chromosomes exchange parts (typically, halves) to produce a new chromosome. During the mutation, one or multiple genes of the chromosome randomly change their values. In this thesis, we use a standard two-parent crossover and a single-gene mutation as proposed in [83] and illustrated in Figure 2.11 and Figure 2.10, respectively. The new population of candidate chromosomes, generated using the recombination and mutation, is used in the next iteration of the GA-based search. The GA-based search terminates when either a maximum number of iterations or a termination condition (e.g., satisfactory fitness level) has been reached.

Beside the two problem-specific attributes, mentioned above, a GA also has a number of parameters such as the maximum number of GA iterations, the number of individuals in the initial population, the probability of mutation in the chromosomes and others [83]. These parameters are typically user-defined.

Chapter 3

Methodology for high-throughput CNN inference

Svetlana Minakova, Erqian Tang, Todor Stefanov. "Combining Task- and Data-level Parallelism for High-Throughput CNN Inference on Embedded CPUs-GPUs MPSoCs". *In Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pp. 18-35, Pythagoreio, Samos Island, Greece, July 05-09, 2020.

IN this chapter, we present our methodology for high-throughput CNN inference at the Edge, which corresponds to the first research contribution of this thesis summarized in Section 1.5.1. The proposed methodology is a part of the system-level optimization engine, introduced in Section 1.5 and is aimed at relaxation of Limitation 1, introduced in Section 1.4.1. The reminder of this chapter is organized as follows. Section 3.1 introduces, in more details, the problem addressed by our novel methodology. Section 3.2 summarizes the novel research contributions, presented in this chapter. An overview of the related work is given in Section 3.3. Section 3.4 presents the platform model, used in this Chapter to represent a target edge platform, where the high-throughput CNN inference is executed. Section 3.5 presents our proposed methodology. Section 3.6 presents the experimental study performed by using the proposed methodology. Section 3.7 ends the chapter with conclusions.

3.1 Problem statement

As mentioned in Chapter 1 (see Section 1.2), many CNN-based applications require CNNs to process their input data streams fast, i.e., to have high throughput. These applications are often executed on edge platforms based on CPUs-GPUs multi-processor systems-on-chip (MPSoCs) [63]. Due to their specific design, CPUs-GPUs MPSoCs offer energy-efficient and high-performance solutions, which makes them very suitable for running high-throughput CNN inference at the Edge [14]. However, achieving high-throughput execution of the computationally-intensive CNN inference phase on embedded CPUs-GPUs MPSoCs is a complex task.

On the one hand, it requires effective utilization of parallelism, available in a CNN. When the CNN inference is executed on an embedded CPUs-GPUs MPSoC, the CNN computational workload is distributed among the heterogeneous MPSoC processors: embedded CPUs and GPUs. Due to their specific structure, the CPUs are more suitable for handling task-level parallelism, compared to GPUs, whereas GPUs are more suitable for handling data-level parallelism, compared to CPUs [88]. Thus, for efficient execution of the CNN inference on an embedded MPSoC, the task-level parallelism should be handled by the CPUs, available in an embedded MPSoC, i.e., different CNN layers should, if possible, be executed on different CPUs, and the overall CNN computational workload should be balanced among the CPUs [6]. Additionally, the data-level parallelism, available within CNN layers, should be handled by embedded GPUs, i.e., the embedded CPUs should offload data-parallel computations within the CNN layers onto the embedded GPUs, thereby accelerating the computations within CNN layers for further improvement of the CNN inference throughput, already achieved by efficient task-level parallelism exploitation. Thus, efficient execution of the CNN inference on an embedded CPUs-GPUs MPSoC involves efficient exploitation of both task-level parallelism and data-level parallelism, available in the CNN.

On the other hand, effective utilization of task- and data-level parallelism requires proper communication and synchronization between tasks, executed on different processors of an embedded MPSoC. In this respect, attempting to utilize an unnecessary large amount of CNN parallelism on limited embedded MPSoC resources, results in unnecessary communication and synchronization overheads, that reduce the CNN inference throughput. Thus, to achieve high CNN inference throughput, the CNN inference, executed on an embedded MPSoC, should utilize the right amount of parallelism, which matches the computational capacity of the MPSoC.

Based on the discussion above, we argue, that efficient execution of the

CNN inference on a CPUs-GPUs embedded MPSoC requires:

1. efficient handling of the task-level parallelism, available in a CNN, by CPUs;
2. CPU workload balancing;
3. efficient handling of the data-level parallelism, available in a CNN, by GPUs;
4. efficient exploitation of task- and data-level parallelism, which matches the computational capacity of an embedded MPSoC.

However, the existing Deep Learning (DL) frameworks [1,42,43,49,72,74, 75,90,94,101], that enable execution of the CNN inference on embedded CPUs-GPUs MPSoCs, only partially satisfy requirements 1) to 4), mentioned above. These frameworks can be divided into two main groups. The first group includes frameworks [101] and [94], that exploit only task-level parallelism, available in a CNN, and efficiently utilize only embedded CPUs. Thus, these frameworks satisfy requirements 1) and 2), mentioned above, and do not satisfy requirement 3). The second group includes frameworks [1,42,43,49, 72,74,75,90], that exploit only data-level parallelism, available in a CNN, and efficiently utilize only embedded GPUs. Thus, these frameworks satisfy requirement 3), mentioned above, but do not satisfy requirements 1) and 2). Moreover, all frameworks [1,42,43,49,72,74,75,90,94,101] directly utilize the CNN computational model to execute the CNN inference on embedded CPUs-GPUs MPSoCs. The large amount of parallelism, available in a CNN model, typically does not match the limited computational capacity of embedded CPUs-GPUs MPSoC. Thus, frameworks [1,42,43,49,72,74,75,90,94,101] do not satisfy requirement 4), mentioned above.

Therefore, in this chapter, we propose a novel methodology for efficient execution of the CNN inference on embedded CPUs-GPUs MPSoCs.

3.2 Contributions

In this chapter, we propose a novel methodology for execution of the CNN inference on embedded CPUs-GPUs MPSoCs (Section 3.5). Our methodology exploits task-level (pipeline) and data-level parallelism, available in a CNN and explained in Section 2.4, to efficiently distribute (map) the computations within the CNN to the computational resources of an edge platform. Thus, our methodology takes full advantage of all CPU and GPU resources, available

in an MPSoC, and ensures high-throughput CNN inference execution on the MPSoC. Exploitation of task-level (pipeline) parallelism together with data-level parallelism for high-throughput CNN inference at the edge is our main novel contribution. Other important novel contributions are:

1. the automated conversion of a CNN model into a functionally equivalent SDF model (Section 3.5.1). Unlike the CNN model, presented in Section 2.1 and typically used to represent CNNs, the SDF model, presented in Section 2.5, can explicitly specify task- and data-level parallelism, available in a CNN. Moreover, unlike the CNN model, the SDF model has the tasks communication and synchronization mechanisms, suitable for efficient mapping and execution of a CNN on an embedded MPSoC. Thus, a conversion of a CNN model into a SDF model enables for efficient mapping and execution of a CNN on an embedded CPUs-GPUs MPSoC.

2. the automated conversion of a CNN model into a functionally equivalent platform-aware executable CSDF model (see Section 2.5 for the CSDF model definition), which efficiently utilizes CPUs-GPUs embedded MPSoC computational resources (Section 3.5.3);

3. taking state-of-the-art CNNs from the ONNX models zoo [7] and mapping them on a Nvidia Jetson MPSoC [71], we achieve a 1.36% to 42% higher throughput, when the CNN inference is executed with our methodology, compared to the throughput of the CNN inference, executed by the best-known and state-of-the-art Tensorrt DL framework [72] for Nvidia Jetson MPSoCs (Section 3.6).

3.3 Related work

The well-known Deep Learning (DL) frameworks, such as TensorFlow [1], Pytorch [75] and others [74] and some of the Deep Learning frameworks for embedded devices such as [42, 43, 49, 50, 72, 90] efficiently exploit data-level parallelism, available in a CNN, for efficient utilization of embedded GPUs. However, these frameworks do not exploit task-level parallelism, available in a CNN. They execute the CNN inference layer-by-layer, i.e., at every computational step only one CNN layer is executed. Such layer-by-layer execution of CNN layers is performed either on a single CPU, which utilizes GPU devices for acceleration, or on all available embedded CPUs. Thus, at every computational step, either some of the embedded CPUs are not utilized, or

embedded GPUs are not utilized. Therefore, these frameworks cannot take full advantage of all CPU and GPU resources and cannot achieve high CNN inference throughput, typically required for the CNN inference, executed on embedded MPSoCs [23,24,87]. Unlike these frameworks, our methodology exploits together both task-level parallelism and data-level parallelism, available in the CNN. In our methodology, the CNN layers are distributed on embedded CPUs, such that the CNN workload is balanced among the CPUs, and at every computational step several CNN layers are executed in parallel (pipeline) fashion. At the same time, some of the computations within CNN layers are performed on efficiently-shared embedded GPU devices. Thus, in our methodology, at every computational step all available CPU and GPU resources are efficiently utilized. Therefore, our methodology allows to achieve higher CNN inference throughput, compared to the frameworks, presented in [1,42,43,49,72,74,75,90].

The frameworks, presented in [101] and [94], exploit task-level parallelism, available among CNN layers, for efficient execution of the CNN inference on an embedded MPSoC. In these frameworks, CNN layers are distributed on the embedded CPUs and executed in parallel (pipeline) fashion, which provides higher CNN throughput than sequential (layer-by-layer) execution of CNN layers. However, these frameworks do not utilize embedded GPUs, available in an MPSoC. As a consequence, these frameworks cannot increase further the CNN inference throughput. In contrast, in our methodology, the throughput, achieved by efficient task-level parallelism exploitation, is further increased by exploitation of data-level parallelism, i.e., by exploitation of embedded GPU devices to accelerate the computations within CNN layers. In our methodology, some computations within CNN layers are offloaded onto embedded GPUs and performed in parallel. Parallel execution of computations within CNN layers allows to reduce the execution time of individual CNN layers and to increase the CNN inference throughput. Therefore, our methodology ensures higher CNN inference throughput, compared to frameworks [101] and [94].

3.4 Edge platform model

In this Chapter, we represent an edge platform as a platform model. The platform model provides a simplified, yet accurate description of computational resources, available on the platform. As mentioned above, in this Chapter we concentrate on edge platforms based on embedded CPUs-GPUs MPSoCs, which computational resources are composed of CPUs and embedded GPUs.

Formally, we define a platform model as a set $platform = \{cpu, gpu\}$, where $cpu = \{cpu_1, cpu_2, ..., cpu_n\}$ is a set of CPU cores, available on the platform and used for CNN inference; $gpu = \{gpu_1, gpu_2, ..., gpu_m\}$ is a set of all GPU devices, available in the platform, and typically $m \leq n$. For example, we model the Jetson TX2 edge platform shown in Figure 2.4 and explained in Section 2.3, as platform model $Jetson = \{cpu, gpu\}$, where $cpu = \{cpu_1, cpu_2, ..., cpu_5\}$ is a set of 5 out of 6 CPU cores, available on the platform and used for CNN inference. The sixth core available on the platform is not included in the model because it is allocated to other parts of a CNN-based application and is not used for CNN inference; $gpu = \{gpu_1\}$ is a set of GPUs, available on the platform and used for CNN inference.

3.5 Methodology

In this section, we present our methodology for high-throughput CNN inference at the Edge. Our methodology, shown in Figure 3.1, consists of three main steps. In Step 1 (Section 3.5.1), we convert a CNN, represented as a CNN model (see Section 2.1 for the CNN model definition) into a functionally equivalent SDF model (see Section 2.5 for the SDF model definition). Unlike the CNN model, the SDF model explicitly specifies task- and data-level parallelism, available in a CNN, as well as it explicitly specifies the tasks communication and synchronization mechanisms, suitable for efficient mapping and execution of a CNN on an embedded MPSoC. Thus, a conversion of a CNN model into a SDF model enables for efficient mapping and execution of a CNN on an embedded CPUs-GPUs MPSoC.

In Step 2 (Section 3.5.2), we find an efficient mapping of the SDF model, obtained in Step 1, on an embedded CPUs-GPUs MPSoC represented as the edge platform model, proposed in Section 3.4. The mapping describes the

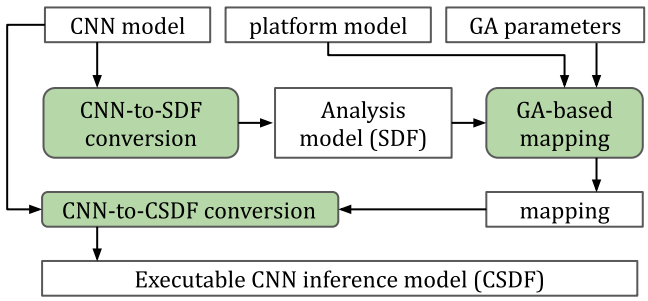


Figure 3.1: *Methodology for high-throughput CNN inference*

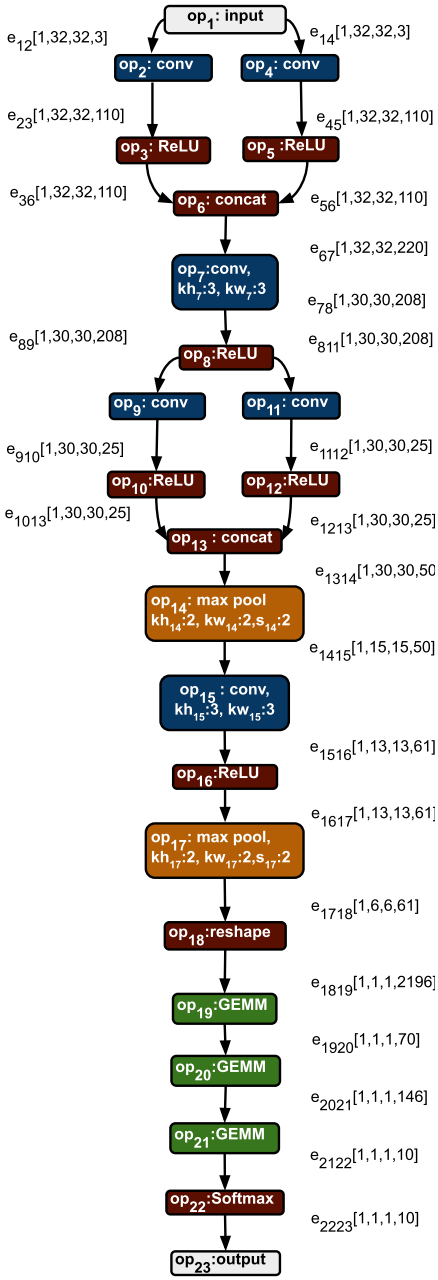


Figure 3.2: *CNN (input) model*

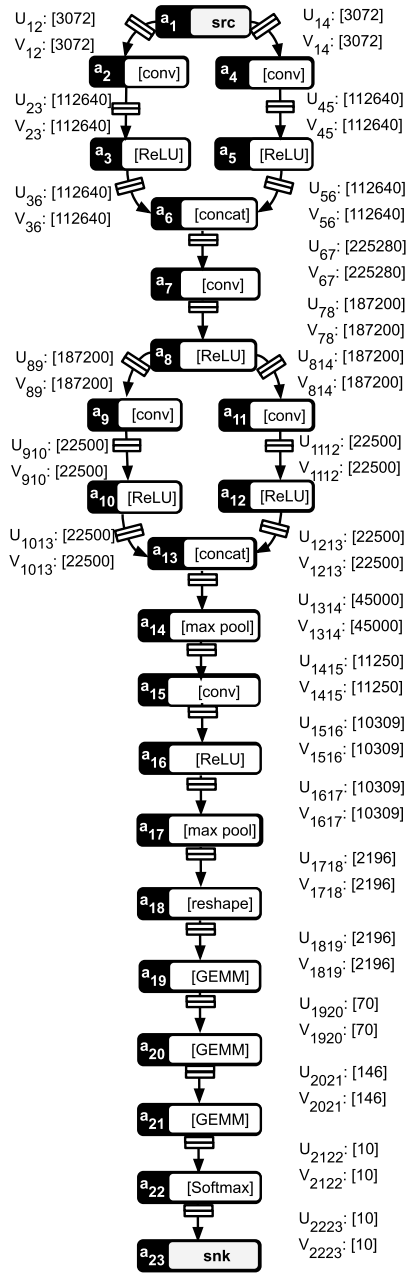


Figure 3.3: *SDF (analysis) model*

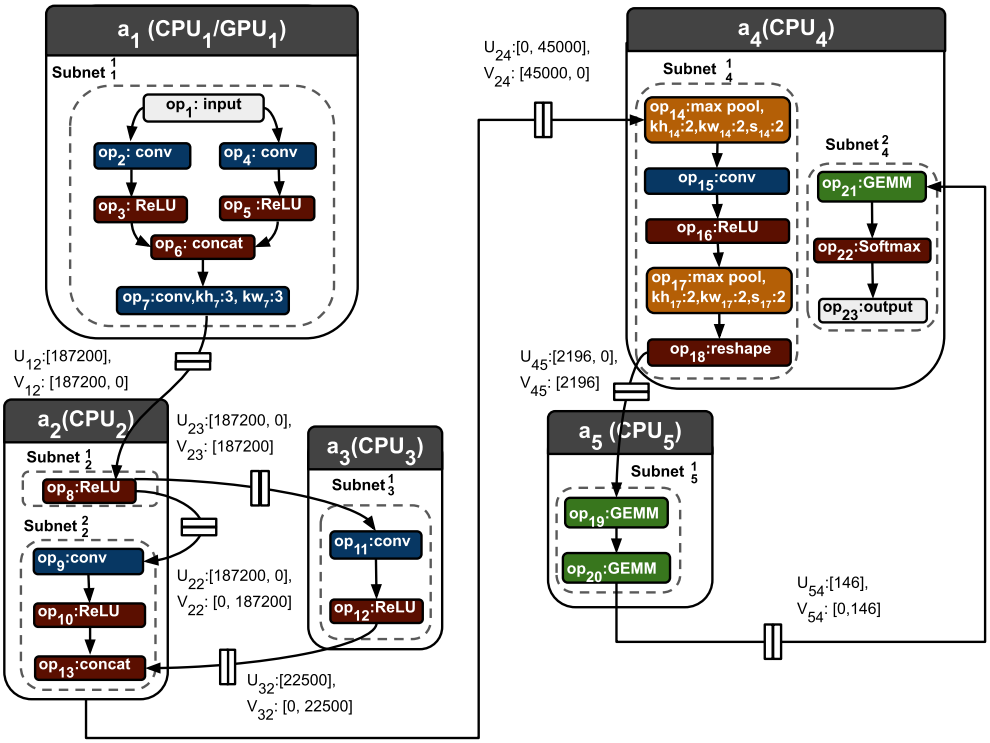


Figure 3.4: *CSDF (executable CNN inference) model*

distribution of the CNN inference computational workload on an embedded MPSoC. The mapping is considered efficient when it ensures high-throughput CNN inference. To find such a mapping, we propose to utilize a simple Genetic Algorithm (GA), which basic concepts and standard parameters are presented in Section 2.6.

Finally, in Step 3 (Section 3.5.3), we use the mapping, obtained in Step 2, to convert a CNN model into a final platform-aware executable application model. The final application model is represented as a Cyclo-Static Dataflow (CSDF) model (see Section 2.5 for the CSDF model definition). The CSDF model, obtained in Step 3, describes the CNN inference as an application, efficiently distributed over embedded MPSoC processors and exploiting the right amount of task- and data-level parallelism, which matches the computational capacity of an embedded MPSoC.

To illustrate the Steps performed by our methodology, we use an example, where we apply our methodology to 1) the CNN model shown in Figure 3.2; 2) the *Jetson* platform model introduced in Section 3.4; 3) a set of GA parameters

where the initial population size = 1000, number of epochs = 500, mutation probability = 5%. The SDF model and the CSDF model, automatically obtained in Step 1 and Step 3 of our methodology from the aforementioned inputs 1), 2) and 3), are shown in Figure 3.3 and Figure 3.4, respectively.

3.5.1 CNN-to-SDF conversion

In this section, we show how we automatically convert a CNN model, introduced in Section 2.1, into a functionally equivalent SDF model, introduced in Section 2.5. The conversion procedure is given in Algorithm 1. An example of the CNN-to-SDF conversion, performed by Algorithm 1, is given in Section 3.5, where the CNN model, shown in Figure 3.2, is automatically converted into the SDF model, shown in Figure 3.3.

Algorithm 1 accepts as an input a CNN model $CNN(L, E)$ and generates as an output a functionally equivalent SDF model $G(A, C)$. In Line 1, it creates an empty SDF model. In Lines 2 to 6, Algorithm 1 converts every CNN layer l_i into a functionally equivalent actor a_i. According to the definition of the SDF model, given in Section 2.5, the sequence F_i, executed by actor a_i, has a single phase. At its single phase, actor a_i executes operator op_i of layer l_i, thereby reproducing the functionality of layer l_i. In Lines 7 to 12, Algorithm 1 converts every CNN edge e_{ij} into FIFO channel c_{ij}. In Lines 9 to 11, Algorithm 1 defines the production sequence U_{ij} and the consumption sequence V_{ij} of channel c_{ij}. Both sequences have a single element, computed as the number of data

Algorithm 1: CNN-to-SDF conversion

Input: $CNN(L, E)$
Result: $G(A, C)$
1 $A, C \leftarrow \emptyset$; $G(A, C) \leftarrow$ SDF model (A, C);
2 **for** $l_i \in L$ **do**
3 $\quad F_i = \emptyset$;
4 $\quad F_i \leftarrow F_i + op_i$;
5 $\quad a_i \leftarrow$ actor (F_i);
6 $\quad A \leftarrow A + a_i$;
7 **for** $e_{ij} \in E$ **do**
8 $\quad c_{ij} \leftarrow$ FIFO channel (a_i, a_j);
9 $\quad U_{ij} \leftarrow \emptyset$; $V_{ij} \leftarrow \emptyset$;
10 $\quad U_{ij} \leftarrow U_{ij} + |e_{ij}.data|$;
11 $\quad V_{ij} \leftarrow V_{ij} + |e_{ij}.data|$;
12 $\quad C \leftarrow C + c_{ij}$;
13 **return** $G(A, C)$

elements $|e_{ij}.data|$, exchanged through edge e_{ij} of the CNN model.

Unlike the CNN model $CNN(L, E)$, accepted as an input by Algorithm 1, the functionally equivalent SDF model $G(A, C)$, generated by Algorithm 1, explicitly specifies both task-level and data-level parallelism, which could be exploited during the CNN inference phase, as well as this SDF explicitly specifies the communication and synchronization mechanism between the actors/tasks, needed to execute the CNN inference properly. The task-level parallelism, available among CNN layers, is explicitly specified in the SDF model topology, where every actor $a_i \in A$ is a task, performing the functionality of CNN layer $l_i \in L$, and the total number of tasks, needed to perform the CNN model functionality, is equal to the number of actors in the SDF model. The communication and synchronization between the tasks, are explicitly specified by the SDF FIFO channels, where every channel $c_{ij} \in C$ specifies, that actor $a_i \in A$ communicates with actor $a_j \in A$ through a FIFO buffer, and the production-consumption rates of the channels $c_{ij} \in C$ determine the frequency and the order of the actors firings - for more details see [57]. The data-level parallelism is explicitly specified in the channels production rates. For example, production rate $U_{36} = [112640]$ of FIFO channel c_{36}, shown in Figure 3.3, explicitly specifies that, when actor a_3 fires, it produces 112640 data tokens, and each token can be obtained in parallel by executing 112640 parallel $ReLU$ operations within each firing of a_3.

The SDF explicit specification of the tasks, that can be potentially performed during the CNN inference, and the SDF explicit specification of the communication and synchronization between the tasks, allow to perform a search for efficient mappings of the CNN onto an embedded CPUs-GPUs MPSoC.

3.5.2 GA-based mapping

In this section, we show how we obtain an efficient mapping of a SDF model $G(A, C)$, generated by Algorithm 1, onto an embedded CPUs-GPUs MPSoC $Jetson = \{\{cpu_1, cpu_2, ..., cpu_5\}, \{gpu_1\}\}$ introduced in Section 3.4. In our methodology, the CNN inference tasks, explicitly specified as SDF actors, are executed on embedded CPU cores, that are able to efficiently handle the task-level parallelism. To efficiently utilize the data-level parallelism, available within the tasks, some of the CPU cores offload computations on the embedded GPUs. Since the number of embedded GPU devices is limited, it may occur, that the efficient exploitation of task-level parallelism, by embedded CPUs, is disrupted due to CPUs competition for the limited embedded GPU devices. To avoid such disruption, for every embedded GPU $gpu_j \in gpu$, we allocate a

Table 3.1: *Mapping example*

cpu_1/gpu_1	cpu_2	cpu_3	cpu_4	cpu_5
$a_1, a_2, a_3, a_4,$ a_5, a_6, a_7	$a_8, a_9,$ a_{10}, a_{13}	a_{11}, a_{12}	$a_{14}, a_{15}, a_{16}, a_{17}, a_{18},$ a_{21}, a_{22}, a_{23}	a_{19}, a_{20}

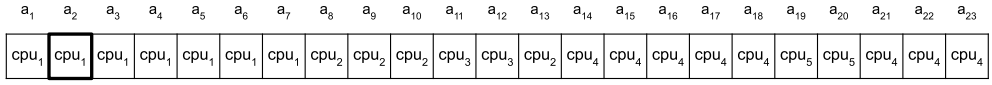


Figure 3.5: *Mapping chromosome example*

single CPU core $cpu_i \in cpu$, which offloads computations on gpu_j.

Based on the discussion above, we define a mapping of SDF model $G(A, C)$ onto *Jetson*, as a partition of actors set A into n subsets, where $n = |cpu|$ is the number of CPU cores, available in the MPSoC. We denote such mapping as ${}^nA = \{{}^nA_1, {}^nA_2, ..., {}^nA_n\}$, where each ${}^nA_i \in {}^nA$ is a subset of actors, mapped on cpu_i, such that $\cap_{i=1}^{n} {}^nA_i = \emptyset$, and $\cup_{i=1}^{n} {}^nA_i = A$. The first $m = |gpu|$ number of CPU cores in mapping ${}^nA$ offload computations on the corresponding embedded GPUs, i.e., the computations within every actor $a_k \in {}^nA_j, j \in [1, m]$ are performed on gpu_j, and the computations within every actor $a_k \in {}^nA_i, i \in [m+1, n]$ are performed on cpu_i. An example of mapping ${}^5A = \{{}^5A_1, {}^5A_2, {}^5A_3, {}^5A_4, {}^5A_5\}$ of the SDF model $G(A, C)$, shown in Figure 3.3 on the *Jetson* CPUs-GPUs MPSoC, is given in Table 3.1. Every Column in Table 3.1 corresponds to a subset ${}^5A_i, i \in [1, 5]$. For example, Column 1 in Table 3.1 corresponds to subset ${}^5A_1 = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7\}$. The actors within subset ${}^5A_1$ are mapped on cpu_1, which offloads computations on gpu_1. Column 2 in Table 3.1 describes subset ${}^5A_2 = \{a_8, a_9, a_{10}, a_{13}\}$. Every actor $a_i \in {}^5A_2$ is mapped on cpu_2. Since the example *Jetson* MPSoC does not have gpu_2, all computations within actors in ${}^5A_2$ are performed only on cpu_2.

We consider that a mapping is efficient, if it ensures that the workload is balanced [6] among all embedded CPU cores, including those, that offload computations on embedded GPUs. We note, that obtaining such an efficient mapping of an SDF graph onto a CPUs-GPUs MPSoC is a complex Design Space Exploration (DSE) problem. In our methodology, to solve this problem, we propose to use a simple Genetic Algorithm (GA) with a standard two-parent crossover and a single-gene mutation, as introduced in Section 2.6. To utilize such a GA for searching of an efficient mapping ${}^nA$, we represent mapping ${}^nA$, as a mapping chromosome: a string of length $|A|$, where every gene is a CPU core $cpu_i \in cpu$. An example of the chromosome, corresponding to mapping ${}^5A$, shown in Table 3.1, is given in Figure 3.5.

In our methodology, we search for a mapping, in which the workload is balanced among all CPU cores, available in the MPSoC, i.e., the difference in execution time between every pair of CPU cores $(cpu_i \in cpu, cpu_j \in cpu), i \neq j$, is minimized. Thus, we define a specific fitness-function $fitness$ to be minimized during the GA-based search as:

$$fitness = \sum_{\forall (cpu_i, cpu_j) \in cpu^2} |\tau_{cpu_i} - \tau_{cpu_j}| \tag{3.1}$$

where τ_{cpu_i} and τ_{cpu_j} are the total execution time of cpu_i and cpu_j, respectively. For every $cpu_i \in cpu$, τ_{cpu_i} is computed as:

$$\tau_{cpu_i} = \tau^t_{cpu_i} + \tau^{com}_{cpu_i} \tag{3.2}$$

where $\tau^t_{cpu_i}$ is the time, required by cpu_i to execute all tasks, mapped on cpu_i; $\tau^{com}_{cpu_i}$ is the time, required for communication of cpu_i with other embedded processors. The time $\tau^t_{cpu_i}$ is computed as:

$$\tau^t_{cpu_i} = \sum_{a_k \in {}^nA_i} \tau_{(f_k(1), cpu_i)} \tag{3.3}$$

where ${}^nA_i$ is the set of all actors, mapped on cpu_i; $f_k(1)$ is the function, performed by actor $a_k \in {}^nA_i$ at every firing; $\tau_{(f_k(1), cpu_i)}$ is the time, taken by cpu_i to execute $f_k(1)$, measured on the MPSoC. The time $\tau^{com}_{cpu_i}$ is computed as:

$$\tau^{com}_{cpu_i} = \sum_{a_k \in {}^nA_i} (\tau_w * \sum_{c_{kj} \in C} u_{kj}(1) + \tau_r * \sum_{c_{qk} \in C} v_{qk}(1)) \tag{3.4}$$

where ${}^nA_i$ is the set of all actors, mapped on cpu_i; $c_{kj} \in C$ is an output channel of actor $a_k \in {}^nA_i$, to where, at each firing, actor a_k produces $u_{kj}(1)$ tokens; $c_{qk} \in C$ is an input channel of actor a_k, from where, at each firing, actor a_k consumes $v_{qk}(1)$ tokens; τ_r and τ_w specify the time, needed by a CPU core, to read and write one data token, respectively. τ_r and τ_w are measured on the MPSoC.

3.5.3 CNN-to-CSDF model conversion

In this section, we show how we automatically convert a CNN model, introduced in Section 2.1, into a final executable platform-aware application, represented as a CSDF model, introduced in Section 2.5. The conversion procedure is given in Algorithm 2.

Algorithm 2: CNN-to-CSDF conversion

Input: $CNN(L, E)$, ${}^{n}A$
Result: $G(A, C)$

$A, C \leftarrow \emptyset$; $G(A, C) \leftarrow$ CSDF model (A, C);
$E_{out} = \emptyset$;
for ${}^{n}A_i \in {}^{n}A$ **do**
  $F_i = \emptyset$; $p = 1$;
  $Q = \emptyset$; $visited = \emptyset$;
  for $l_k : a_k \in {}^{n}A_i \wedge l_k \notin visited$ **do**
    $L_i^p, E_i^p \leftarrow \emptyset$;
    $Q = Q + l_k$;
    while $Q \neq \emptyset$ **do**
      $l_j = Q.pop()$;
      $L_i^p = L_i^p + l_j$;
      $visited = visited + l_j$;
      if $\exists e_{js} \in E : a_s \notin {}^{n}A_i$ **then**
        for $e_{js} \in E$ **do**
          $E_{out} = E_{out} + e_{js}$;
        break;
      else
        for $e_{js} \in E, l_s \notin visited$ **do**
          $Q = Q + l_s$;
          $E_i^p = E_i^p + e_{js}$;
    $Subnet_i^p =$ new Subnet (L_i^p, E_i^p);
    $F_i = F_i + Subnet_i^p$;
    $p = p + 1$;
  $a_i \leftarrow$ actor (F_i);
  $A = A + a_i$;
for $e_{ij} \in E_{out}$ **do**
  $a_k \in A : l_i \in L_k^g$; $a_r \in A : l_j \in L_r^z$;
  $c_{kr} \leftarrow$ FIFO channel (a_k, a_r);
  $u_{kr}(p) = \begin{cases} |e_{ij}.data|, \text{ if p = g} \\ 0, \text{ otherwise} \end{cases}$
  $v_{kr}(p) = \begin{cases} |e_{ij}.data|, \text{ if p = z} \\ 0, \text{ otherwise} \end{cases}$
return $G(A, C)$

Algorithm 2 accepts as inputs a CNN model $CNN(L, E)$ and an efficient mapping ${}^{n}A$, obtained in Section 3.5.2, and generates a CSDF model $G(A, C)$, which performs the functionality of the CNN model $CNN(L, E)$, efficiently mapped on an embedded MPSoC, as specified by mapping ${}^{n}A$. An example of the CSDF model $G(A, C)$, generated by Algorithm 2, using as inputs the CNN

model $CNN(L, E)$, shown in Figure 3.2, and mapping ${}^5A$, shown in Table 3.1 and explained in Section 3.5.2, is given in Figure 3.4.

In Line 1, Algorithm 2 creates an empty CSDF model. In Lines 3-25, Algorithm 2 generates the set of actors A, such that every actor $a_i \in A$ represents the functionality of all CNN layers, mapped on CPU core cpu_i, as specified in mapping ${}^nA$, where for $\forall l_k \in L$, executed on cpu_i, $\exists a_k \in {}^nA_i$. At every phase $p \in [1, P_i]$ actor a_i executes function $Subnet_i^p$, implemented by means of an existing DL framework. Every $Subnet_i^p$ performs layer-by-layer execution of layers $L_i^p \subseteq L$, mapped on cpu_i, and connected via edges E_i^p. For example, actor a_3, shown in Figure 3.4, represents the functionality of all CNN layers, mapped on cpu_3. It executes $F_3 = \{Subnet_3^1\}$, where $Subnet_3^1$ performs layer-by-layer execution of layers $L_3^1 = \{l_{11}, l_{12}\}$, connected via edges $E_3^1 = \{e_{1112}\}$, on cpu_3.

Every edge $e_{js} \in E$ between layers l_j and l_s, sequentially executed on the same CPU core, is implemented by means of an existing DL framework, e.g. as device memory, shared by layers l_j and l_s [72]. If layers l_j and l_s, connected via edge $e_{js} \in E$, are executed on different CPU cores, the task-level parallelism is exploited between these layers, and edge e_{js} is converted into a FIFO channel, which explicitly specifies and implements the communication and synchronization between actors, executing layers l_j and l_s. For example, edge e_{811}, shown in Figure 3.2, connects layer l_8, executed by actor a_2 on cpu_2, and layer l_{11}, executed by actor a_3 on cpu_3. Thus, edge e_{811} is converted into a FIFO channel c_{23}, shown in Figure 3.4, where c_{23} explicitly specifies and implements the communication and synchronization between actor a_2, executing layer l_8 and actor a_3, executing layer l_{11}.

Between some actors, cyclic dependencies occur, that may lead to deadlocks in the CSDF model. To avoid the deadlocks, Algorithm 2 specifies the execution of every actor a_i in one or more phases, such that at every phase $p \in [1, P_i]$, actor a_i has no cyclic dependencies. For the example, shown in Figure 3.4, a cyclic dependency occurs between actors a_2 and a_3. If actor a_2 would execute layers l_8 and l_{13} in one phase, according to the semantics of the CSDF model [10], it would expect 187200 data tokens to be present in channel c_{12} and 22500 data tokens to be present in channel c_{32}, before it can fire. However, data in channel c_{32}, should be produced by actor a_3, which, before it can fire, expects actor a_2 to produce 187200 data tokens in channel c_{23}. Thus, such execution would lead to a deadlock in the CNN inference. To avoid the deadlock, Algorithm 2 specifies the execution of actor a_2 in 2 phases. At phase $p = 1$, actor a_2 executes only layer l_8. It consumes data only from channel c_{12}, and produces data to channel c_{23}, such that actor a_3 can fire.

At phase $p = 2$, actor a_2 consumes data only from channel c_{32}, and executes layers l_9, l_{10} and l_{13}. Thus, at every phase $p = [1, 2]$, actor a_2 has no cyclic dependencies, and no deadlock occurs in the CSDF model execution.

In Lines 5-23, Algorithm 2 performs a mapping-aware Breadth-First Search (BFS) [26] over the CNN model graph and determines functions $Subnet_i^p$, $p \in [1, P_i]$, executed by actor a_i. In Line 7, for every not-visited layer l_k, mapped on cpu_i, Algorithm 2 creates an empty set of layers L_i^p and an empty set of edges E_i^p. In Line 8, it adds layer l_k to the BFS queue [26] Q, and starts BFS. In Lines 10-12, Algorithm 2 extracts layer l_j from Q and adds l_j to L_i^p. In Line 13, Algorithm 2 checks, if layer l_j, mapped on cpu_i, has at least one child layer l_s, which is not mapped on cpu_i. If the condition in Line 13 is met, to avoid the deadlocks, which can occur in a CSDF model, as discussed above, Algorithm 2 stops adding layers to L_i^p and goes to Lines 14-15, where it adds every output edge of layer l_j to the list of outer edges E_{out}, utilized in Lines 26-30 of Algorithm 2 for CSDF channels generation. If every child layer l_s of layer l_j is mapped on cpu_i (condition in Line 13 of Algorithm 2 is not met), in Lines 18-20, Algorithm 2 adds every connection e_{js} to the set E_i^p, and every layer l_s to Q and continues BFS.

In Line 21, Algorithm 2 creates function $Subnet_i^p$, which performs layer-by-layer execution of layers L_i^p, connected via edges E_i^p. In Line 22, Algorithm 2 adds function $Subnet_i^p$ to execution sequence F_i of actor a_i. When all layers, mapped on cpu_i, are visited, Algorithm 2 adds actor a_i, which executes F_i, to the CSDF model actors set (see Lines 24-25).

In Lines 26-30, Algorithm 2 converts every outer edge $e_{ij} \in E_{out}$ into a CSDF channel c_{kr}, specifying and implementing the communication and synchronization between actor $a_k \in A$ executing layer l_i, and actor $a_r \in A$ executing layer l_j . For example, for edge e_{78}, shown in Figure 3.2, Algorithm 2 creates FIFO channel c_{12}, shown in Figure 3.4, where actor a_1 executes layer l_7, and actor a_2 executes layer l_8.

3.6 Experimental results

In this section, we present our results from an experiment, where real-world CNNs from the ONNX models zoo [7] are mapped and executed on the NVIDIA Jetson TX2 embedded CPUs-GPUs MPSoC [71]. We compare the CNN inference throughput, which we measure, when the CNN is mapped on the NVIDIA Jetson TX2 by: 1) the popular ARM CL framework [8], which on the NVIDIA Jetson MPSoC can exploit only task-level parallelism, available in the CNN; 2) the best-known and state-of-the-art for the NVIDIA Jetson

Table 3.2: *Experimental results, average over 100 runs*

CNN	Throughput (fps)			Thr. increase, compared to TensorRT (%)
	ARM CL	TensorRT	Our	
bvlc alexnet	8.7	104	140	35
VGG 19	1.84	15	21.3	42
bvlc googlenet	3.9	118	154	31
tiny yolo v2	3.2	131	133	1.36
inception v1	4.25	122	166	36
resnet18	8.7	137	143	4.37
densenet121	3	62	69	12
Emotion FER	21.2	325	416	28

TX2 MPSoC, TensorRT DL framework [72], which exploits only data-level parallelism, available in the CNN; 3) our methodology, explained in Section 3.5, which exploits both task- and data-level parallelism and uses the ARM CL framework to implement CNN layers on embedded CPUs together with the TensorRT framework to implement CNN layers on embedded GPUs. For every CNN in the experimental results: 1) The throughput is measured on the platform as an average value over 100 CNN inference executions; 2) the original (float32) data precision is utilized, such that the baseline CNN accuracy is preserved; 3) The dataset parameters, such as size and precision of input data samples as well as the batch size are obtained from the ONNX model representation; 4) The GA, utilized for efficient mapping search (see Section 3.5.2) is executed with initial population size = 1000, number of epochs = 500, mutation probability = 5%. If for 50 epochs no improvements are achieved by the GA, the GA stops.

The experimental results are given in Table 3.2. Column 1 lists the CNNs. Columns 2-4 show the CNN inference throughput in frames per second (fps) for ARM CL, TensorRT, and our methodology, respectively. Columns 2 and 4 in Table 3.2 show that the throughput achieved by the ARM CL framework is much lower than the throughput, achieved by our methodology. This difference occurs because our methodology exploits both task- and data-level parallelism, available in the CNN, whereas the ARM CL framework, executing the CNN inference on the NVIDIA Jetson MPSoC, does not offload computations on the embedded GPU, available in the MPSoC. Therefore, ARM CL does not efficiently exploit the data-level parallelism, available in the CNN. Columns 3 and 4 in Table 3.2 show that our methodology achieves higher inference throughput than the TensorRT framework. This difference occurs because our methodology exploits both task- and data-level parallelism, whereas TensorRT executes the CNN inference layer-by-layer and exploits only data-level parallelism, available in the CNN. Column 5 shows the throughput increase

achieved by our methodology in comparison with the TensorRT framework, which achieves highest throughput for every CNN among the TensorRT and ARM CL frameworks. The numbers in Column 5 indicate that our methodology enables to achieve 1.36% to 42% throughput increase compared to the TensorRT framework.

3.7 Conclusion

We propose a novel methodology which exploits both task- and data-level parallelism, available in a CNN, and takes full advantage of all CPU and GPU resources, available in a MPSoC, to achieve high-throughput CNN inference execution. We evaluated our proposed methodology by mapping a set of real-world CNNs on the NVIDIA Jetson TX2 embedded CPUs-GPUs MPSoC. The evaluation results show that taking real-world CNNs from the ONNX models zoo and mapping them on the Jetson MPSoC, a 1.36% to 42% higher throughput is achieved when the CNN inference is executed with our methodology compared to the throughput of the CNN inference, executed by the best-known and state-of-the-art TensorRT DL framework for the Jetson MPSoC.

Chapter 4

Methodology for low-memory CNN inference

Svetlana Minakova and Todor Stefanov. "Buffer Sizes Reduction for Memory-efficient CNN Inference on Mobile and Embedded Devices". *In Proceedings of 23rd Euromicro Conference on Digital System Design (DSD'20)*, pp. 133-140, Portoroz, Slovenia, August 26-28, 2020.

In this chapter, we present our methodology for low-memory CNN inference at the Edge, which corresponds to the second research contribution of this thesis summarized in Section 1.5.2. The proposed methodology is a part of the system-level optimization engine, introduced in Section 1.5, and is aimed at relaxation of Limitation 1, introduced in Section 1.4.1. The reminder of this chapter is organized as follows. Section 4.1 introduces, in more details, the problem addressed by our novel methodology. Section 4.2 gives a summary of the contributions, presented in the chapter. An overview of the related work is given in Section 4.3. Section 4.4 provides a motivational example. Section 4.5 presents the proposed methodology. Section 4.6 presents the experimental study performed by using the proposed methodology. Finally, Section 4.7 ends the chapter with conclusions.

4.1 Problem statement

As mentioned in Chapter 1 in Section 1.2, in order to be deployed and executed on an edge platform, a CNN is required to have low memory footprint. This is because modern edge platforms have limited memory resources. For example,

the basic version of the Raspberry Pi 4 [30] embedded platform has 1 GB of memory. For comparison, deployment and inference of the state-of-the-art VGG-19 CNN [4], requires about 700 MB of memory. If deployed on the Raspberry Pi 4, VGG-19 CNN would occupy almost all memory available on the platform and leave insufficient memory space for the operating system running on the platform, libraries required to execute the CNN inference, storage of the CNN input and output data, etc.

To enable inference of a state-of-the-art CNN such as VGG-19 on an edge platform such as Raspberry Pi 4, the CNN memory footprint should be reduced. To this aim, the CNN memory reduction methodologies [5,11,17,31, 73,76,98] have been proposed. The most common of these methodologies, namely pruning and quantization [11,17,31,98], reduce the memory footprint of a CNN by reducing the number or size of CNN parameters (weights and biases). However, at high memory reduction rates, these methodologies may decrease the CNN accuracy, while, as mentioned in Section 1.2, high accuracy is very important for most CNN-based applications. Moreover, for many state-of-the-art CNNs [4], the intermediate computational results, exchanged between CNN layers and stored in the platform memory during the CNN inference, take even more space than the CNN parameters. For example, for the MobileNet V2 [81] and DenseNet [40] CNNs, the intermediate computational results comprise up to 63% and 80% of the total CNN memory requirement, respectively[1]. For these CNNs, the memory reduction achieved only by methodologies such as pruning and quantization (i.e., only by reducing the amount of memory needed to store CNN parameters) may not be sufficient to fit the CNN into the memory of the target edge platform. In other words, CNN inference at the edge requires a methodology, which reduces the amount of memory required to store the intermediate computational results of a CNN, that is complimentary to the pruning and quantization methodologies. In this chapter, we propose such a methodology.

4.2 Contributions

We propose a novel methodology, which reduces the amount of memory required to store intermediate computational results of a CNN, thereby reducing the CNN memory footprint. Our proposed methodology is based on the ability of CNN operators to process data by parts, illustrated in Figure 2.2 and explained in Section 2.1. In our methodology, the execution of every CNN layer is performed in several *phases*, such that: 1) at each phase, the layer

[1]The percentage is given for the CNNs deployed and executed with no memory reduction

processes only a part of its input data; 2) phases are executed in a specific order; 3) the platform memory, allocated to store intermediate computational results of a CNN is reused between the data parts. As the data processing by parts may cause CNN execution time overheads (e.g. CNN layers may require time to switch among the data parts), our methodology may reduce the CNN throughput. However, unlike the most common pruning and quantization methodologies [11,17,31,98], our methodology does not change the number and precision of CNN parameters and therefore does not decrease the CNN accuracy. Thus, our methodology is orthogonal to the pruning and quantization methodologies, and can be combined with these methodologies for further CNN memory footprint reduction. Our proposed methodology, presented in Section 4.5, is our main novel contribution. Other important novel contributions are:

1. the phases derivation algorithm (see Section 4.5.1). This algorithm automatically derives the number of phases, performed by every CNN layer. The number of phases is computed such that at each phase, every layer of a CNN processes a minimum part of the layer input data and produces a minimum part of the layer output data. Thus, the phases derivation algorithm ensures that every layer requires minimum amount of memory to store its intermediate computational results at every phase;

2. the CNN-to-CSDF conversion algorithm (see Section 4.5.2). This algorithm automatically converts a CNN, represented as the CNN model (see Section 2.1) into a functionally equivalent CSDF model (see Section 2.5). Unlike the CNN model, the CSDF model has means for explicit specification of the CNN inference with phases. Thus, the CNN-to-CSDF conversion algorithm enables for CNN inference with phases, underlying our proposed methodology;

3. 2.8% to 38% CNN memory reduction, compared to the most relevant buffers reuse methodology, exploited by the well-known and widely used TensorRT [72] DL framework for CNN deployment and inference at the Edge (see Section 4.6).

Scope of work: in this chapter, we assume that every input CNN is executed with the smallest possible batch size (i.e., $batch = 1$) typical for CNN execution at the Edge. This restriction comes from the fact that data batching (i.e., using $batch > 1$) [35] is the opposite to the data processing by parts, used by our proposed methodology. The data processing by parts involves splitting of the intermediate CNN computational results into parts, which enables for

reduction of the CNN memory footprint at the cost of possible CNN throughput decrease. The data batching, on the other hand, involves aggregating the intermediate CNN computational results in the platform memory, which leads to increase of the CNN throughput at the cost of CNN memory footprint increase.

4.3 Related Work

The most common CNN memory reduction methodologies, namely pruning and quantization, reviewed in surveys [11,17,31,98], reduce the memory cost of a CNN by reducing the number or size of CNN parameters (weights and biases) [4]. However, at high memory reduction rates these methodologies decrease the CNN accuracy, whereas high accuracy is very important for many CNN-based applications [4]. In contrast, our memory reduction methodology does not change the CNN model parameters and therefore does not decrease the CNN accuracy. Moreover, our methodology reduces the platform memory occupied by the CNN intermediate computational results, while the pruning and quantization methodologies reduce the platform memory occupied by the CNN parameters (weights and biases). Therefore our methodology is orthogonal to the pruning and quantization methodologies, and can be combined with these methodologies for further CNN memory footprint reduction.

The Knowledge Distillation (KD) methodologies try to shift knowledge from an initial CNN into another CNN, with smaller size but with the same accuracy. However, KD methodologies involve training from scratch and do not guarantee that the accuracy of the initial CNN can be preserved. Moreover, KD methodologies can only be applied to CNNs designed to perform classification [17], while many CNNs are designed to perform other tasks, such as object detection or segmentation [4]. In contrast, our memory reduction methodology is a general systematic methodology, which always guarantees preservation of the CNN accuracy, and is not limited to CNNs designed to perform classification tasks.

The CNN layers fusion methodologies, such as the methodologies [5,73] and the methodologies adopted by DL frameworks, such as TensorRT [72] or PyTorch [75], enable to reduce the CNN memory cost by transforming the network into a simpler form but preserving the same overall behavior. Being a part of the CNN model definition, the CNN layer fusion methodologies are orthogonal to our proposed methodology and can be combined with our methodology for further CNN memory optimizations. In our experimental study (Section 4.6), we implicitly use the CNN layers fusion by implementing

the CNNs inference with the TensorRT DL framework [72], which has built-in CNN layers fusion.

The buffers reuse methodologies, such as the methodology proposed in [76] and the methodology, employed by the TensorRT framework for efficient CNN inference at the edge [72], reduce the CNN memory footprint by sharing memory between CNN layers, executed at different computational steps. However, these methodologies do not reuse memory within CNN layers. As a result, these methodologies are not very efficient for: 1) CNNs with residual connections, such as ResNets [36] and DenseNets [40], because in these CNNs the data associated with different layers has to be stored for many computational steps; 2) CNNs that process high-resolution input data, because in these CNNs one layer can occupy significant amount of platform memory to store its input and output data. In contrast, our methodology reuses memory within layers of a CNN, which makes our methodology more efficient at reducing the memory of CNNs that have residual connections or/and process high-resolution data. We note that the CNN buffers reuse methodologies are the only methodologies among the related work that can be directly compared to other proposed methodology. Other related works, discussed above, are either orthogonal to our proposed methodology (e.g., pruning and quantization methodologies [11,17,31,98]) or cannot be directly compared to our proposed methodology (e.g., the KD methodologies [17]). Therefore, in our experimental results (see Section 4.6), we compare our methodology only to the buffers reuse methodologies. More precisely, we compare our methodology to the buffers reuse methodology, exploited by the TensorRT [72] DL framework.

4.4 Motivational Example

Layers of a CNN do not process their input data at once. As shown in Figure 2.2 and explained in Section 2.1, to process its input data, a layer of a CNN moves along the input data with a sliding window, applying an operator to the parts of the input data. In this section, we show how this feature can be used to reduce the memory cost of a CNN. We define the processing of an input data part by a layer as a *phase*. If a layer has one phase, it processes its input data as one part. If a layer has two phases, it processes its input data in two parts, etc.

In Table 4.1, we give four examples (Ex1, Ex2, Ex3, Ex4) of inference of the CNN, shown in Figure 4.1 and represented as the CNN model, introduced in Section 2.1. In each of these examples the CNN inference is executed on the Jetson TX2 platform introduced in Section 2.3 and shown in Figure 2.4. Every layer of the CNN is executed in one or multiple phases. For every

Table 4.1: *Execution of CNN inference with phases*

Ex.	Layer phases					Phases execution order	Buffer sizes (Bytes)					Thr. (fps)
	l_1	l_2	l_3	l_4	l_5		B_1	B_2	B_4	B_5	Total	
Ex1	$X_{1k} = \emptyset$, $Y_{1k}^{[1,32,32,1]}$ Φ_1=1	$X_{2k}^{[1,32,32,1]}$, $Y_{2k}^{[1,16,16,4]}$ Φ_2=1	$X_{3k}^{[1,16,16,4]}$, $Y_{3k}^{[1,4,4,3]}$ Φ_3=1	$X_{4k}^{[1,4,4,3]}$, $Y_{4k}^{[1,1,1,2]}$ Φ_4=1	$X_{5k}^{[1,1,1,2]}$, $Y_{5k} = \emptyset$ Φ_5=1	$l_{11}, l_{21}, l_{31}, l_{41}, l_{51}$	1024	1024	48	2	2098	334
Ex2	$X_{1k} = \emptyset$, $Y_{1k}^{[1,24,32,1]}$ Φ_1=2	$X_{2k}^{[1,24,32,1]}$, $Y_{2k}^{[1,8,16,4]}$ Φ_2=2	$X_{3k}^{[1,16,16,4]}$, $Y_{3k}^{[1,4,4,3]}$ Φ_3=1	$X_{4k}^{[1,4,4,3]}$, $Y_{4k}^{[1,1,1,2]}$ Φ_4=1	$X_{5k}^{[1,1,1,2]}$, $Y_{5k} = \emptyset$ Φ_5=1	$l_{11}, l_{21}, l_{12}, l_{22}, l_{31}, l_{41}, l_{51}$	768	1024	48	2	1842	333
Ex3	$X_{1k} = \emptyset$, $Y_{1k}^{[1,1,32,1]}$ Φ_1=32	$X_{2k}^{[1,17,32,1]}$, $Y_{2k}^{[1,1,16,4]}$ Φ_2=16	$X_{3k}^{[1,16,16,4]}$, $Y_{3k}^{[1,4,4,3]}$ Φ_3=1	$X_{4k}^{[1,4,4,3]}$, $Y_{4k}^{[1,1,1,2]}$ Φ_4=1	$X_{5k}^{[1,1,1,2]}$, $Y_{5k} = \emptyset$ Φ_5=1	$l_{1(1-17)}, l_{21}, [l_{1(18-32)}, l_{2(2-16)}], l_{31}, l_{41}, l_{51}$	544	1024	48	2	1618	310
Ex4	$X_{1k} = \emptyset$, $Y_{1k}^{[1,1,32,1]}$ Φ_1=32	$X_{2k}^{[1,17,32,1]}$, $Y_{2k}^{[1,1,16,4]}$ Φ_2=16	$X_{31}^{[1,6,16,4]}$, $X_{32}^{[1,5,16,4]}$, $X_{33}^{[1,5,16,4]}$, $X_{34}^{[1,6,16,4]}$, $Y_{3k}^{[1,1,4,3]}$ Φ_3=4	$X_{4k}^{[1,4,4,3]}$, $Y_{4k}^{[1,1,1,2]}$ Φ_4=1	$X_{5k}^{[1,1,1,2]}$, $Y_{5k} = \emptyset$ Φ_5=1	$l_{1(1-17)}, l_{21}, [l_{1(18-22)}, l_{2(2-6)}], l_{31}, [l_{1(23-25)}, l_{2(7-9)}], l_{32}, [l_{1(26-28)}, l_{2(10-12)}], l_{33}, [l_{1(29-32)}, l_{2(13-16)}], l_{34}, l_{41}, l_{51}$	544	384	48	2	978	308

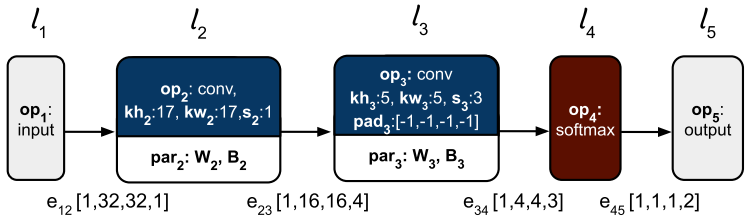
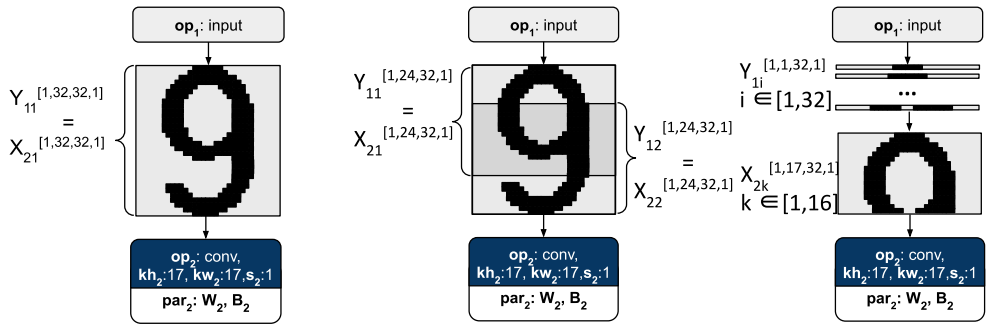


Figure 4.1: *Example CNN*

layer $l_i, i \in [1,5]$, Columns 2 to 6 in Table 4.1 list: 1) the number of phases Φ_i; 2) part X_{ik} of the input data X_i, processed by layer l_i at its k-th phase, $k \in [1,\Phi_i]$; 3) part Y_{ik} of the output data Y_i, produced by layer l_i at its k-th phase, $k \in [1,\Phi_i]$. The data parts are annotated with the shape, introduced for CNN data tensors in Section 2.1. Recall that in this thesis, every data tensor is represented as a 4-dimensional tensor of shape $[batch, h, w, ch]$, where $batch, h, w, ch$ are the tensor batch size, the height, the width, and the number of channels, respectively. All phases are executed in a specific order, given in Column 7, where l_{ik} denotes the execution of the k-th phase of layer l_i. The execution order ensures functional equivalence of all examples, given in Table 4.1, and allows to reduce the CNN buffer sizes as explained below. Columns 8 to 12 in Table 4.1 show the sizes of the CNN buffers, introduced in Section 2.2, i.e., segments of platform memory, allocated to store intermediate computational results, produced by the CNN layers. Every CNN edge e_{ij} is allocated its own buffer B_k. The size of each buffer is computed using Equation 2.8 explained in Section 2.2 with the assumption that one element in any CNN data tensor requires 1 byte of memory for its storage. Column 13 in Table 4.1 shows the CNN throughput in frames per second (fps). As shown in Column 13, the CNN inference throughput differs for examples Ex1, Ex2, Ex3, Ex4. This is because data processing by parts may cause CNN execution time overheads (e.g. CNN layers may require time to switch among the data parts), leading to CNN throughput decrease. The more phases are performed by a CNN (i.e., the more data parts are accepted and produced by the CNN layers), the larger the throughput overhead is. For example, the throughput of Ex4, where the CNN layers processes data in 32, 16, 4, 1 and 1 phases respectively, is 26 fps smaller than the throughput of Ex1, where every CNN layer processes data in one phase.

Example Ex1, given in Row 3 of Table 4.1, describes the CNN inference typically performed by state-of-the-art DL frameworks, such as TensorFlow, Keras, Caffe2, and other [74]. In Ex1, every layer has one phase. The CNN inference is performed in 5 computational steps. At every i-th computational step, $i \in [1,5]$, the single phase of layer l_i is executed. During its single



(a) *Ex1* **(b)** *Ex2* **(c)** *Ex3, Ex4*

Figure 4.2: *Input data processing by layer l_2*

phase, layer l_i processes its whole input data. Figure 4.2(a) shows how layer l_2 processes its input data in Ex1. Layer l_2 processes its whole input data X_2 as a single data part $X_{21} = X_2$. Data X_{21} is provided to layer l_2 by layer l_1 and stored in buffer B_1. To store data $X_{21}^{[1,32,32,1]}$ buffer B_1 occupies 1 * 32 * 32 *1 = 1024 bytes of memory.

Example Ex2, given in Row 4 of Table 4.1, shows how processing data by parts, combined with specific execution order of the phases, allows to reduce the CNN buffer sizes at the cost of decreasing the CNN throughput. In Ex2, CNN layer l_2 processes its input data X_2 in two overlapping parts, X_{21} and X_{22}, as shown in Figure 4.2(b). Data parts X_{21} and X_{22} are provided to layer l_2 by layer l_1 and stored in buffer B_1 during the CNN inference. The CNN inference is performed in 7 computational steps. At step 1, phase l_{11} is executed and data $Y_{11} = X_{21}$ is produced in B_1. At step 2, phase l_{21} is executed and data X_{21} is processed by layer l_2. After being processed, data X_{21} is not needed anymore and is removed from B_1. At step 3, phase l_{12} is executed and data $Y_{12} = X_{22}$ is produced in B_1. At step 4, phase l_{22} is executed and data X_{22} is processed by layer l_2. Steps 1 to 4 in Ex2 are functionally equivalent to steps 1 to 2 in Ex1. However, in Ex2 at every computational step, buffer B_1 has to store only a part of the input data ($X_{21}^{[1,24,32,1]}$ for steps 1 to 2 and $X_{22}^{[1,24,32,1]}$ for steps 3 to 4, respectively). Therefore, in Ex2, B_1 occupies 1* 24 * 32 * 1= 786 bytes of memory, instead of 1024 bytes, as in Ex1. Compared to Ex1, Ex2 reduces the total buffer sizes by 12% at the cost of only 0.3% throughput decrease due to the increased number of CNN computational steps in Ex2, compared to Ex1.

Example Ex3, given in Row 5 of Table 4.1, demonstrates one more way of executing layers l_1 and l_2 with phases, shown in Figure 4.2(c). In Ex3, layer l_1 has 32 phases and layer l_2 has 16 phases. The CNN inference is performed in 51 computational step. During the first 17 steps, phases l_{11}, l_{12}, ..., l_{117}, shortly written as $l_{1(1-17)}$, are executed. At every phase, layer l_1 produces

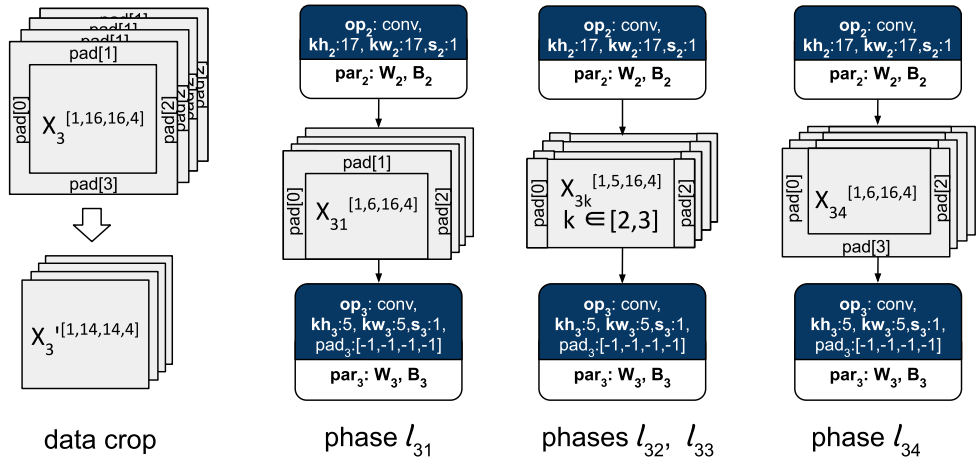


Figure 4.3: *Input data processing by layer l_3, Ex4*

data $Y_{1k}^{[1,1,32,1]} \subset X_{21}$ in buffer B_1, until sufficient data $X_{21}^{[1,17,32,1]}$ is accumulated. Then, at step 18, phase l_{21} is executed. To execute phase l_{22}, data $X_{22}^{[1,17,32,1]}$ should be accumulated in B_1. However, some of this data is already in B_1. As explained in Section 2.1, data between subsequent execution steps of layer l_2 is overlapping. If the overlapping part is stored in buffer B_1, only new (non-overlapping) data should be produced in B_1 to enable the execution of phase l_{22}. This new data can be produced by execution of one phase of layer l_1. Thus, phases 18-32 of layer l_1 and phases 2-16 of layer l_2 are executed in order $[l_{1(18-32)}, l_{2(2-16)}]$, meaning, that a phase of layer l_1 is followed by a phase of layer l_2, e.g., phase l_{118} is followed by phase l_{22}, and this pattern repeats, until all phases of layers l_1 and l_2 are executed. The maximum amount of data, stored between layers l_1 and l_2 per computational step corresponds to data part $X_{2k}^{[1,17,32,1]}$, accumulated in B_1. Thus, in Ex3, buffer B_1 occupies 1 * 17 * 32 * 1 = 544 bytes of memory. Compared to Ex1, Ex3 reduces the total buffer sizes by 23% at the cost of 7% throughput decrease.

Example Ex4, given in Row 6 of Table 4.1, demonstrates how several Convolutional layers in one CNN can be executed with phases, and how data padding is processed with phases. In Ex4, the CNN inference is executed in 54 computational steps. Layers l_1 and l_2 have 32 and 16 phases, respectively, as in Ex3. Additionally, layer l_3 has 4 phases, i.e., processes its input data in four parts. As explained in Section 2.1, layer l_3 has padding pad_3, which crops its input data. With data processing by parts, the data crop is also performed by parts, as shown in Figure 4.3. At phases l_{31} and l_{34}, layer l_3 accepts data $X_{3k}^{[1,6,16,4]}$ and crops it to data $X_{3k}^{\prime[1,5,14,4]}$. At phases l_{32} and l_{33}, it accepts data

$X_{3k}^{[1,5,16,4]}$ and crops it to data $X_{3k}^{\prime[1,5,14,4]}$. The maximum amount of data to be stored in B_2 is $X_{3k}^{[1,6,16,4]}$. Thus, buffer B_2 occupies 1 * 6 * 16 * 4 = 384 bytes of memory. Compared to Ex1, Ex4 reduces the total buffer sizes by 53% at the cost of 12.7% throughput decrease. As can be seen from Column 12 of Table 4.1, Ex4 is the most memory-efficient example among all presented examples.

The examples, provided in this section, demonstrate that there are many possible ways to execute the CNN inference with phases. Obtaining the most memory-efficient way is not trivial even for our small example CNN, shown in Figure 4.1, let alone for real-world state-of-the-art CNNs that are much larger and much more complex. Therefore, a systematic and automated methodology for finding the CNN inference execution with phases, which ensures minimum buffer sizes, is required. In the next section, we propose such a methodology.

4.5 Methodology

In this section, we present our three-step methodology for low-memory CNN inference at the Edge. Our methodology is shown in Figure 4.4. In Step 1 (Section 4.5.1), we automatically derive the number of phases for every CNN layer. The number of phases is computed such that at each phase, every CNN layer processes a minimum part of the layer input data and produces a minimum part of the layer output data. Thus, we ensure that every layer requires minimum amount of memory to store its input and output data at every phase. In Step 2 (Section 4.5.2), we model the CNN inference with phases, obtained at Step 1. We note that the CNN model, introduced in Section 2.1 and widely used to represent CNNs, does not have means for explicit specification of the CNN execution with phases, while the CSDF model, introduced in Section 2.5, has such means. Moreover, unlike the CNN model, the CSDF model is accepted as an input by many existing embedded systems

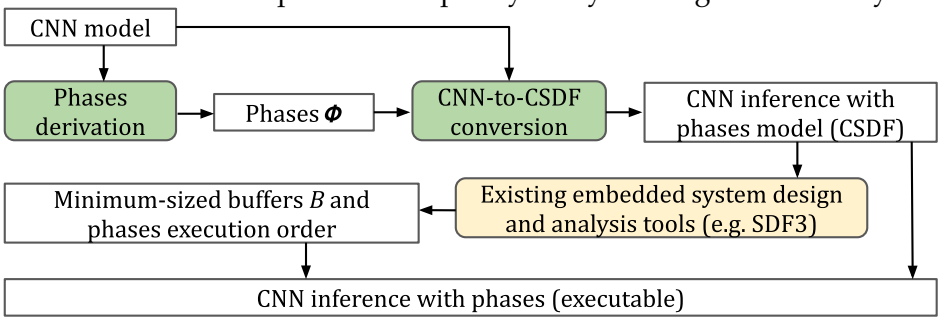


Figure 4.4: *Methodology for low-memory CNN inference*

design tools for automated performance/memory analysis, transformations and optimizations. Therefore, to enable for CNN execution with phases and utilization of existing embedded design tools, e.g., SDF3 [91], for the CNN analysis, in Step 2, we automatically convert a CNN model into a functionally equivalent CSDF model. In Step 3, we use the SDF3 tool to analyse the CNN and obtain a set of buffers B, used to store the intermediate computational results of the CNN, represented as the CSDF model at Step 2. Every buffer $B_k \in B$ is characterized with minimum size. Together with buffers B, the SDF3 tool obtains specific execution order of phases, which enables to correctly execute the CNN inference with buffers B. Thus, in our 3-step methodology, we use processing data by parts to ensure the CNN inference with minimum buffer sizes.

4.5.1 Phases derivation

In this section, we present our automated phases derivation algorithm - see Algorithm 3. Algorithm 3 accepts as an input a CNN, represented as the CNN model, explained in Section 2.1. As an output, Algorithm 3 provides a set of phases $\Phi = \{\Phi_1, \Phi_2, ..., \Phi_{|L|}\}$, where $\Phi_i \in \Phi$ is the number of phases, performed by layer l_i of the input CNN. For example, for the CNN shown in Figure 4.1, Algorithm 3 automatically derives a set of phases $\Phi = \{32, 16, 4, 1, 1\}$, which specifies that layers l_1, l_2, l_3, l_4, and l_5 of the CNN process data in $32, 16, 4, 1$, and 1 phases, respectively.

In Line 1, Algorithm 3 defines the set of phases Φ as an empty set. In Lines 2 to 8, Algorithm 3 computes the number of phases Φ_i for every layer l_i of the input CNN. Φ_i is computed such that at each phase, layer l_i accepts a part of the input data and produces the corresponding part of the output data. Each

Algorithm 3: Phases derivation

Input: $CNN(L, E)$
Result: Set of phases Φ

1 $\Phi \leftarrow \emptyset$;
2 **for** $l_i \in L$ **do**
3 **if** *size of* Θ_i = *size of* X_i **then**
4 $h^{out}_{min} = Y_i.h$;
5 **else**
6 $h^{out}_{min} = 1$;
7 $Y_{ik} \leftarrow$ part of Y_i of shape $[Y_i.batch, h^{out}_{min}, Y_i.w, Y_i.c]$;
8 $\Phi_i \leftarrow Y_i.h / Y_{ik}.h$;
9 $\Phi \leftarrow \Phi + \Phi_i$;
10 **return** Φ

part of input and output data of layer l_i is characterized with minimum height, determined by the attributes of layer l_i, shown in Table 2.1 and explained in Section 2.1. An example of such layer execution is shown in Figure 4.2(c), explained in Section 4.4, where layer l_2 performs $\Phi_2 = 16$ phases. At each phase $k \in [1, 16]$ layer l_2 accepts an input data part X_{2k} with minimum height of 17 pixels, and produces an output data part Y_{2k} with height of 1 pixel.

In Lines 3 to 6, Algorithm 3 computes the minimum height h_{min}^{out} of output data part Y_{ik}, produced by layer l_i at each phase. h_{min}^{out} is 1 pixel for every layer l_i, except of layers that process their input data at once (i.e., layers for which condition in Line 3 is met). In Line 7, Algorithm 3 defines output data part Y_{ik}, produced by layer l_i at each phase. Y_{ik} has the shape $[Y_i.batch, h_{min}^{out}, Y_i.w, Y_i.c]$, where $Y_i.batch$, $Y_i.w$, and $Y_i.c$ are the batch size, the width and the number of channels of output data Y_i, produced by layer l_i, and h_{min}^{out} is the minimum output data height, computed in Lines 3 to 6. In Line 8, Algorithm 3 computes the number of phases Φ_i performed by layer l_i as the number of output data parts with minimum height, produced by layer l_i. In Line 9, Algorithm 3 adds Φ_i to the set Φ. Finally, in Line 10, Algorithm 3 returns the set of phases Φ.

4.5.2 CNN-to-CSDF model conversion

The automated conversion of a CNN into a functionally equivalent CSDF model, utilized in our memory reduction methodology, is given in Algorithm 4. Algorithm 4 accepts as inputs a CNN, represented as the CNN model, explained in Section 2.1 and a set of phases Φ, automatically generated for the CNN by Algorithm 3 presented in Section 4.5.1. In Lines 1-16, explained in the *CSDF model topology generation* subsection below, Algorithm 4 generates the topology of the CSDF model $G(A, C)$. In Lines 17-36, explained in the *Production/consumption sequences derivation* subsection below, Algorithm 4 derives the production/consumption sequences for every channel in $G(A, C)$. Finally, in Line 37, Algorithm 4 returns $G(A, C)$, which is functionally equivalent to

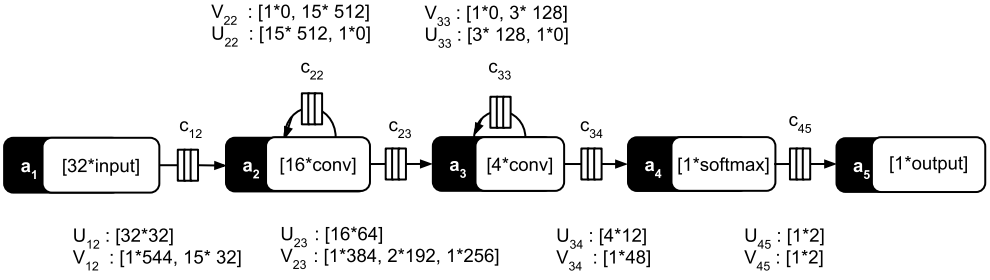


Figure 4.5: *CSDF model, derived from the CNN model shown in Figure 4.1*

the input $CNN(L, E)$ model. Figure 4.5 shows the CSDF model $G(A, C)$, automatically derived by Algorithm 4 from the CNN model $CNN(L, E)$ shown in Figure 4.1 and phases $\Phi = \{32, 16, 4, 1, 1\}$, derived for this CNN model by Algorithm 3. The examples, provided in this section for Algorithm 4, are referring to this CNN-to-CSDF conversion.

Algorithm 4: CNN-to-CSDF conversion

Input: $CNN(L, E), \Phi = \{\Phi_1, \Phi_2, ..., \Phi_{|L|}\}$
Result: $G(A, C)$

$A, C \leftarrow \emptyset$; $G(A, C) \leftarrow$ CSDF model (A, C) ;
foreach $l_i \in L$ **do**
  $F_i \leftarrow \emptyset$;
  $a_i \leftarrow$ actor (F_i);
  $A \leftarrow A + a_i$;
  $\{\Theta_i, op_i, s_i\} \leftarrow$ attributes of l_i (see Table 2.1);
  $P_i = \Phi_i$;
  for $p \in [1, P_i]$ **do**
    $f_i(p) = op_i$;
    $F_i = F_i + f_i(p)$;
  if $s_i < \Theta_i.h$ **then**
    $c_{ii} \leftarrow channel(a_i, a_i)$;
    $C \leftarrow C + c_{ii}$;
foreach $e_{ij} \in E$ **do**
  $c_{ij} \leftarrow channel(a_i, a_j)$;
  $C \leftarrow C + c_{ij}$;
foreach $c_{ij} \in C$ **do**
  $\{X_i, Y_i, kh_i, s_i, pad_i\} \leftarrow$ attributes of l_i (see Table 2.1);
  $\{X_j, Y_j, kh_j, s_j, pad_j\} \leftarrow$ attributes of l_j (see Table 2.1);
  if $i = j$ **then**
    for $p \in [1, P_i]$ **do**
      $u_{ij}(p) = \begin{cases} 0 & if \;\; p = P_i \\ X_i.batch * (\Theta_i.h - s_i) * X_i.w * X_i.ch & otherwise \end{cases}$
      $v_{ij}(p) = \begin{cases} 0 & if \;\; p = 1 \\ X_i.batch * (\Theta_i.h - s_i) * X_i.w * X_i.ch & otherwise \end{cases}$
  else
    for $p \in [1, P_i]$ **do**
      $u_{ij}(p) = Y_i.batch * 1 * Y_i.w * Y_i.ch$;
    $v_{ij}(1) = X_j.batch * (kh_j - pad_j[1]) * X_j.w * X_j.ch$;
    $hpad_j = \begin{cases} pad_j[1] + pad_j[3] & if \;\; P_j = 1 \\ pad_j[3] & otherwise \end{cases}$;
    if $\nexists c_{jj} \vee P_j = 1$ **then**
      for $p \in [2, P_j - 1]$ **do**
        $v_{ij}(p) = X_j.batch * kh_j * X_j.w * X_j.ch$;
      $v_{ij}(P_j) = X_j.batch * (kh_j - hpad_j) * X_j.w * X_j.ch$;
    else
      for $p \in [2, P_j - 1]$ **do**
        $v_{ij}(p) = X_j.batch * s_j * X_j.w * X_j.ch$;
      $v_{ij}(P_j) = X_j.batch * (s_j - hpad_j) * X_j.w * X_j.ch$;
return $G(A, C)$

CSDF model topology generation

The CSDF model topology generation is performed in Lines 1-16 of Algorithm 4. In Line 1, Algorithm 4 generates a new CSDF model $G(A, C)$ with an empty set of actors A and an empty set of communication channels C. In Lines 2-10 Algorithm 4 converts every layer l_i of the CNN model $CNN(L, E)$ into a functionally equivalent CSDF actor $a_i \in A$. Every actor $a_i \in A$ performs execution sequence $F_i = \{f_i(p)\}, p \in [1, P_i]$, where every function $f_i(p) \in F_i$ is specified as $f_i(p) = op_i$ (Lines 9-10 of Algorithm 4). On each phase $p \in [1, P_i]$, actor a_i applies operator op_i performed by layer l_i to the part of input data X_{ip} of the layer l_i and produces a part of output data Y_{ip}. Thus, actor a_i reproduces data processing by parts, performed by the layer l_i and explained in Section 4.4. The number of phases Φ_i of actor a_i representing layer l_i is specified in the input set of phases Φ. For example, actor a_3 performs execution sequence $F_3 = [P_3 * op_3] = [4 * conv]$, where $op_3 = conv$ is the operator, performed by layer l_3, 4 is the number of phases Φ_3, specified for layer l_3 in the input set of phases Φ.

In Lines 11-13 Algorithm 4 models overlapping data reuse, explained in Ex3 in Section 4.4. In Line 11, Algorithm 4 checks, if the data overlapping occurs in layer $l_i \in L$. If data overlapping occurs in layer l_i, in Lines 12-13 Algorithm 4 models data overlapping for corresponding actor a_i. Since the CSDF model does not allow internal state specification in actors, the data overlapping/reuse is modeled as self-loop FIFO channels c_{ii}, that store and reuse the overlapping data between subsequent firings of actor a_i. For example, the data overlapping occurs in layer l_3 ($s_3 = 3 < \Theta_3.h = 5$). Therefore, in Lines 12-13, Algorithm 4 creates self-loop channel c_{33}, which stores the overlapping/reuse data for actor a_3.

Finally, in Lines 14-16, Algorithm 4 converts every input CNN model edge $e_{ij} \in E$, representing a data dependency between layers $l_i \in L$ and $l_j \in L$, into communication FIFO channel $c_{ij} \in C$, representing data dependency between actors $a_i \in A$ and $a_j \in A$.

Production/consumption sequences derivation

The production sequence $U_{ij} = \{u_{ij}(p)\}$, $p \in [1, P_i]$ and the consumption sequence $V_{ij} = \{v_{ij}(p)\}$, $p \in [1, P_j]$ are derived for every channel $c_{ij} \in C$ of CSDF graph $G(A, C)$ in Lines 24 to 36 of Algorithm 4. For every data reuse channel $c_{ij} \in C, i = j$, storing the overlapping/reuse data between subsequent firings of actor a_i, the elements of the production/consumption sequences are computed in Lines 21 to 23 of Algorithm 4. Since at the last phase P_i of actor a_i there is no need to produce data to be reused, the last

element of the production sequence $u_{ij}(P_i)$ is set to 0 in Line 22 of Algorithm 4. Since at the first phase actor a_i has not yet produced data in the data reuse channel c_{ij}, the first element of the consumption sequence $v_{ij}(1)$ is set to 0 in Line 23 of Algorithm 4. For all other phases of actor a_i the elements of the production/consumption sequences are computed as the number of tokens in a tensor of shape $[X_i.batch, (\Theta_i - s_i), X_i.w, X_i.ch]$, reused between the subsequent firings of actor a_i. For example, data reuse channel c_{33} has production sequence $U_{33} : [3 * 128, 1 * 0]$ and consumption sequence $V_{33} : [1 * 0, 3 * 128]$.

For CSDF channels c_{ij}, that are not data reuse channels, i.e. $i \neq j$, the elements of the production/consumption sequences are computed in Lines 25 to 36 of Algorithm 4. The elements of the production sequence U_{ij} are computed as the number of elements in the output data part Y_{ip}, produced by actor a_i at phase p. For example, actor a_3 at its every phase $p \in [1, 4]$ produces data $Y_{3p}^{[1,1,4,3]}$, $p \in [1, 4]$, to channel c_{34}. Therefore, the elements of production rate of channel c_{34} are computed in Lines 25 to 26 of Algorithm 4 as $u_{34}(p) = 1 * 1 * 4 * 3 = 12$.

Every element of the consumption sequences $v_{ij}(p), p \in [1, P_j]$ is computed in Lines 27 to 36 of Algorithm 4 as the number of elements in data tensor, consumed by actor a_j from non-overlapping channel c_{ij} on the actors phase $p \in [1, P_j]$ in order to produce data Y_{jp}. The first element of the consumption sequences $v_{ij}(1)$ is computed in Line 27 of Algorithm 4. If no padding occurs at the first phase of actor a_j ($pad[1] = 0$ in Line 27 of Algorithm 4), actor a_j consumes from c_{ij} data X_{jp} with shape $[X_j.batch, X_j.ch, kh_j, X_j.w]$. If actor a_j crops data at the first phase ($pad_j[1] < 0$ in Line 27 of Algorithm 4), actor a_j consumes from c_{ij} data X_{jp} and data to be cropped. If actor a_j extends data at the first phase ($pad[1] > 0$ in Line 27 of Algorithm 4), actor a_j consumes from c_{ij} part of data X_{jp}, which is not provided by padding.

The computation of consumption sequence elements $v_{ij}(p), p \in [2, P_j]$ is divided in two different cases, determined by the presence of data overlapping in the channel sink actor a_j, corresponding to layer l_j. If data overlapping is not presented in actor a_j (Lines 29-32 of Algorithm 4), actor a_j consumes all input data from its non-overlapping input channel c_{ij}. If data overlapping/reuse is presented in actor a_j (Lines 34-36 of Algorithm 4), actor a_j consumes from channel c_{ij} only non-overlapping data. The overlapping/reuse data is consumed by actor a_j from its self-loop channel c_{jj}. In total, actor a_j consumes data X_{jp} at phases $p \in [2, P_j - 1]$ (Lines 30-31, 34-35 of Algorithm 4), and all the remaining data at phase $p = P_j$ (Lines 32, 36 of Algorithm 4). Consumption of all the remaining data from CSDF channels allows to empty the FIFO buffers

and ensure the CSDF model consistency [10].

For example, communication channel c_{23} has consumption sequence V_{23} : $[1 * 384, 2 * 192, 1 * 256]$. The first element of the consumption sequence is computed in Line 27 of Algorithm 4 as $v_{23}(1) = (5 - (-1)) * 16 * 4 = 384$, where $5 * 16 * 4 = 320$ elements are elements of input data tensor X_{31} of shape $[1, 5, 16, 4]$, used by actor a_3 to produce data Y_{31}, and $1 * 16 * 4 = 64$ elements are cropped by actor a_3 according to the padding pad_3. As data overlapping/reuse is presented for a_3 ($\exists c_{33}$), $v_{23}(p), p \in [2, 4]$ are computed in Lines 34-36 of Algorithm 4. At phases $p \in [2, 3]$ actor a_3 consumes non-overlapping data $3 * 16 * 4 = 192$ from channel c_{23}, i.e., $v_{23}(p) = 192, p \in [2, 3]$. At the last phase actor a_3 consumes the remaining data $(3 - (-1)) * 16 * 4 = 256$ from channel c_{23}, i.e. $v_{23}(4) = 256$.

4.6 Experimental Results

In this section, we evaluate our memory reduction methodology in terms of achieved memory footprint reduction as well as we show the cost of this memory footprint reduction in terms of decreased CNN inference throughput. To this end, we take real-world CNNs from the ONNX models Zoo [7] and obtain their memory footprint and inference throughput, when the memory footprint of the CNNs is reduced using: 1) the most relevant CNN buffers reuse methodology, briefly introduced in Section 4.3, and employed by the TensorRT framework for efficient CNN execution at the Edge [72]; 2) our memory reduction methodology, presented in Section 4.5. The results of the experiment are given in Table 4.2.

We perform our experiment in two steps. In Step 1, for every CNN from the ONNX models Zoo, we derive: 1) a TensorRT C++ executable application, which represents the CNN inference with the TensorRT buffers reuse

Table 4.2: *Evaluation of our memory reduction methodology*

CNN	Memory footprint (MB)		Memory reduction (%)	Throughput (fps)		Throughput reduction (%)
	TensorRT	ours		TensorRT	ours	
resnet18	51.6	49	5	137	121	12
googlenet	38.7	31	19.8	118	103	13
tiny yolo v2	77.3	64.3	16.8	131	105	20
inception v1	38.3	31	19	122	106	13
VGG 19	594	577	2.8	15	14.7	2
densenet121	43	40	7.5	62	49	21
squeezenet	10.4	6.4	38	342	262	23

methodology. This application is automatically generated by the TensorRT DL framework from the input CNN description in .onnx format; 2) a C++ CNN inference with phases application, which implements the CNN inference with phases, derived from the same input CNN by our methodology presented in Section 4.5. To implement this application, we use the TensorRT DL framework as well as a custom code generation component, which offers support of the CNN inference with phases (CSDF) model, unsupported by the TensorRT DL framework. The TensorRT DL framework is used to define CNN operators, while our custom code is used to define the CSDF model.

In Step 2, we execute the applications, obtained in Step 1. We measure and compare the memory footprint as well as the throughput of the CNNs, when the memory footprint of the CNNs is reduced using: 1) the TensorRT buffers reuse methodology; 2) our memory reduction methodology. Columns 2 and 3 in Table 4.2 show the memory footprint (in MegaBytes) of every CNN, i.e., the total amount of memory required to store the CNN parameters (weights and biases) together with the CNN intermediate computational results. Column 4 in Table 4.2 shows the memory reduction (in %), achieved by our methodology in comparison with the TensorRT DL framework. It shows that our methodology achieves 2.8% to 38% memory reduction, compared to the TensorRT buffers reuse methodology. The difference in memory reduction can be explained using the CNN characteristics shown in Table 4.3. First of all, as explained in Section 4.5, our methodology only reduces the amount of memory required to store the intermediate computational results of a CNN. Therefore, our methodology is most efficient for CNNs for which the intermediate computational results (stored in the CNN buffers as explained in Section 2.2) constitute the largest part of the total CNN memory requirement. Columns 2 to 4 in Table 4.3 show the amount of memory (in MegaBytes) required to store intermediate computational results (see Columns 2 and 3) and parameters (see Column 4) of the CNNs from the ONNX models Zoo. For example, the

Table 4.3: *CNN characteristics affecting CNN memory reduction and throughput decrease*

CNN	Buffer sizes (MB)		parameters (MB)	Total phases	
	TensorRT	ours		TensorRT	ours
resnet18	4.8	2.2	46.8	68	1962
googlenet	10.7	3	28	143	2630
tiny yolo v2	14.2	0.8	63.5	33	3796
inception v1	10.3	3	28	143	2494
VGG 19	19.3	2.3	574.7	46	2354
densenet121	10.9	7.9	32.1	428	8935
squeezenet	5.1	1.4	5	66	1870

squeezenet CNN (see Row 9 in Table 4.3) requires 1.4 to 5.1 MegaBytes of memory to store its intermediate computational results and and 5 MegaBytes of memory to store its parameters. Analogously, the VGG 19 CNN (see Row 7 in Table 4.3) requires 2.3 to 19.3 MegaBytes of memory to store its intermediate computational results and 574.5 MegaBytes of memory to store its parameters. In other words, the squeezenet CNN requires similar amount of memory to store its intermediate computational results and its parameters, while the VGG 19 CNN requires much more memory to store its parameters than to store its intermediate computational results. Consequently, our methodology achieves a significant, 38%, memory reduction for the squeezenet CNN and small, 2.8%, memory reduction for the VGG 19 CNN (see Row 7 and Row 9, Column 4 in Table 4.2). Secondly, as explained in Section 4.3, unlike the TensorRT buffers reuse methodology, our methodology reuses data within CNN layers. As shown in Section 4.4, the more phases are performed by layers of a CNN, the more memory is reused within the CNN layers and the more memory reduction can our methodology achieve. Thus, the number of phases performed by the CNN layers affects the memory reduction, achieved by our methodology. The number of phases performed by the CNN layers, when the CNNs are executed with the TensoRT buffers reduction methodology and with our methodology, is shown in Columns 5 and 6 in Table 4.3, respectively. When a CNN is executed with the TensoRT buffers reduction methodology, every layer of the CNN performs one phase. Therefore, the total number of phases performed by the layers of a CNN corresponds to the number of layers in the CNN. When a CNN is executed with our methodology, the total number of phases performed by the CNN layers is computed as $\sum_{\Phi_i \in \Phi} \Phi_i$, where Φ is a set of phases, derived for the CNN using Algorithm 3 introduced in Section 4.5.1. For example, Row 5, Columns 5 and 6 in Table 4.3 shows that the tiny yolo v2 CNN performs 33 phases when is executed with the TensoRT buffers reduction methodology and 3796 phases when executed with our methodology. We believe that the high, 16.8%, memory reduction, achieved by our methodology for the tiny yolo v2 CNN (see Column 4, Row 5 in Table 4.2) is due to the large amount of memory reuse within the CNN layers that our methodology introduced into the tiny yolo v2 CNN by increasing the total number of CNN phases $3796/33 \approx 115$ times.

Columns 5 and 6 in Table 4.2 show the throughput (in frames per second), demonstrated by the CNNs executed with the TensorRT buffers reuse methodology and our methodology, respectively. Column 7 shows the throughput decrease (in %), introduced into the CNNs inference by our methodology. It shows that our methodology decreases the CNN throughput by 2% to 23%,

depending on the CNN. As mentioned in Section 4.4, the throughput decrease, possibly introduced in a CNN by our proposed methodology, depends on the amount of phases performed by the CNN layers. The more phases are performed by the CNN layers, the larger is the possible throughput decrease. For example, our methodology introduces more throughput decrease into the tiny yolo v2 CNN than into the resnet18 CNN (see Row 3 and Row 5 in Table 4.2), because it introduces more phases in the tiny yolo v2 CNN than in the resnet18 CNN (see Row 3 and Row 5, Column 6 in Table 4.3). However, being a relative value, the amount of throughput decrease also depends on the overall CNN throughput. For example, the throughput reduction is larger for the squeezenet CNN than for the VGG 19 CNN (see Row 7 and Row 9 in Table 4.2), because the squeezenet CNN has much higher throughput than the VGG 19 CNN, and thus is more sensitive to the throughput decrease, introduced by our methodology.

4.7 Conclusion

We propose a novel CNN memory footprint reduction methodology. Our proposed methodology is based on the ability of CNN operators to process data by parts. By splitting input and output data of CNN layers into parts, and efficiently reusing the platform memory among these parts, our methodology allows to reduce the CNN memory footprint at the cost of decreasing the CNN throughput. The key feature of our methodology is the exploitation of CNNs ability to process data by parts for the CNN memory footprint reduction. The evaluation results show that, compared to the memory reduction, achieved by the most relevant CNN buffers reuse methodology, employed by the TensorRT DL framework for efficient CNN execution at the Edge, our memory reduction methodology allows to reduce the CNN memory footprint by 2.8% to 38% at the cost of 2% to 23% decrease of the CNN throughput.

Chapter 5

Methodology for run-time adaptive inference of CNN-based applications

Svetlana Minakova, Dolly Sapra, Todor Stefanov, Andy Pimentel. "Scenario Based Run-time Switching for Adaptive CNN-based Applications at the Edge". *In ACM Transactions on Embedded Computing Systems (TECS), vol. 21, Iss. 2, Article 14, March 2022.*

IN this chapter, we present our methodology for run-time adaptive inference of CNN-based applications, which corresponds to the third research contribution of this thesis summarized in Section 1.5.3. The proposed methodology is a part of the post-selection optimization component, introduced in Section 1.5, and is aimed at relaxation of Limitation 2, introduced in Section 1.4.2. The reminder of this chapter is organized as follows. Section 5.1 introduces, in more details, the problem addressed by our novel methodology. Section 5.2 summarizes the novel research contributions, presented in this chapter. An overview of the related work is given in Section 5.3. Section 5.4 provides a motivational example. Sections 5.5 to 5.9 present the proposed methodology and its steps. Section 5.10 presents the experimental study performed by using the proposed methodology. Section 5.11 ends the chapter with conclusions.

5.1 Problem statement

As mentioned in Section 1.4.2, a CNN-based application designed using the state-of-the-art design flow shown in Figure 1.3 and explained in Section 1.3,

uses a single CNN to perform its task. This CNN is characterized with certain accuracy and platform-aware characteristics (see Section 1.1) corresponding to requirements posed on the CNN by the application and target edge platform (see Section 1.2). The CNN characteristics remain unchanged during the application run-time. However, the needs of a CNN-based application, and hence the requirements posed on the CNN, may change under the influence of the application environment during the application run-time. For example, a CNN-based road traffic monitoring application, executed on a drone [53], can have different needs, dependent on the situation on the roads and the level of the device's battery. If the traffic is heavy, the application should provide high throughput and high accuracy to process its input data, which typically means high energy cost. However, during a traffic jam, when the high throughput is not required, or in case the battery of the drone is running low, the application would function optimally by prioritizing energy efficiency over the high throughput. This example shows that CNN-based applications need a mechanism that can adapt their characteristics to the changes in the application environment (such as a change of the situation on the roads or a change of the device's battery level) at the application run-time. Moreover, such a mechanism should provide a high level of responsiveness, e.g., if a drone battery is running low, the CNN-based application, executed on the drone, should switch to an energy-efficient mode as soon as possible. However, to the best of our knowledge, neither existing Deep Learning (DL) methodologies [3, 16, 38, 41, 46, 77, 92, 99, 100, 105, 106] for resource-efficient CNN execution at the Edge, nor existing embedded systems design methodologies [13, 68, 108] for execution of run-time adaptive applications at the edge, provide such a mechanism. Therefore, in this chapter, we propose a novel methodology, which enables to adapt a CNN-based application to changes in the application environment during run-time.

5.2 Contributions

In this chapter, we propose a novel methodology which provides run-time adaptation of a CNN-based application, executed at the Edge, to changes in the application environment. Our methodology, shortly referred as scenario-based run-time switching (SBRS) methodology, is based on the concept of scenarios [15], widely used in embedded systems design. According to this concept, an application can have different internal operation modes, called scenarios, each with its own typical characteristics or/and functionality. During run-time, the application can switch among the scenarios, thereby adapting its

characteristics or functionality to changes in the application environment. In our SBRS methodology a scenario is a CNN designed to conform to a specific set of requirements in terms of accuracy and platform-aware characteristics. During the application execution, the application environment can trigger the application to switch between the scenarios, thereby adapting the application characteristics to changes in the application environment. The SBRS methodology, proposed in Section 5.5, is our main novel contribution. Other important novel contributions within the methodology, are:

- A novel SBRS Model of Computation (MoC) (see Section 5.7). The SBRS MoC captures the functionality of a CNN-based application with multiple scenarios and allows for run-time switching between these scenarios.
- An algorithm for automated derivation of the SBRS MoC from a set of application scenarios (see Section 5.8);
- A transition protocol for efficient switching between the CNN-based application scenarios (see Section 5.9).

5.3 Related Work

The platform-aware neural architecture search (NAS) methodologies, proposed in [3,38,46,92,100,105] and reviewed in survey [16], allow for automated generation of CNNs that solve the same problem, and are characterized with different accuracy and platform-aware characteristic. However, these methodologies do not propose a mechanism for run-time switching between these CNNs, while such mechanism is necessary to ensure that application needs are best served at every moment in time. In contrast to the NAS methodologies from [3,16,38,46,92,100,105], our methodology proposes such a mechanism, and ensures that application needs are best served at every moment in time.

The methodologies presented in [12,39,61,96,102,107] propose resource-efficient runtime-adaptive CNN execution at the Edge. These methodologies represent a CNN as a dynamic computational graph, where for every CNN input sample only a subset of the graph nodes is utilized to compute the corresponding CNN output. The subset of graph nodes is selected during the application run-time by special control mechanisms (e.g., control nodes, augmenting the CNN graph topology). The utilization of only a subset of graph nodes at every CNN computational step can increase the CNN throughput and accuracy, and typically reduces the CNN energy cost. However, the

methodologies in [12, 39, 61, 96, 102, 107] cannot adapt a CNN to changes in the application environment, like changes of the device's battery level, which affect the CNN needs during the run-time. The adaptation in these methodologies is driven either by the complexity of the CNN input data [12,39,61,96,102] or by the number of floating-point operations (FLOPs), required to perform the CNN functionality [39, 107], while the changes in the application environment often cannot be captured in the CNN input data or estimated using FLOPs. In contrast to these methodologies, our SBRS methodology adapts a CNN-based application to the changes in the application environment, and therefore, allows to best serve the application needs, affected by such changes.

A number of embedded systems design methodologies, proposed in [13, 68,108], allow for efficient execution of runtime-adaptive scenario-based applications at the Edge. These methodologies represent an application, executed at the Edge, in a specific model of computation (MoC), able to capture the functionality of a runtime-adaptive application associated with several scenarios, and ensure efficient run-time switching between the application scenarios. However, the methodologies in [13, 68, 108] cannot be (directly) applied to CNN-based applications due to a significant semantic difference between the MoCs, utilized in these methodologies and the CNN model [2], typically utilized by CNN-based applications. First of all, the MoCs utilized in [13, 68, 108] lack means for explicit definition of various CNN-specific features, such as CNN parameters and hyperparameters, while, as we show in Section 5.7, explicit definition of these features is required for the application analysis. Secondly, the MoCs utilized in methodologies [13, 68, 108] are not accepted as input by existing Deep Learning (DL) frameworks, such as Keras [19] or TensorRT [72], widely used for efficient design, deployment and execution of CNN-based applications at the Edge. In our methodology, we propose a novel application model, inspired by the methodologies [13, 68, 108], to represent a run-time adaptive CNN-based application and ensure efficient switching between the CNN-based application scenarios. However, unlike the methodologies [13, 68, 108], our methodology 1) explicitly defines and utilizes CNN-specific features for efficient execution of CNN-based applications at the Edge, and 2) allows for utilization of existing DL frameworks for design, deployment, and execution of the CNN-based application at the Edge.

5.4 Motivational Example

In this section, we show the necessity of devising a new methodology for execution of adaptive CNN-based applications at the Edge. To do so, we

present a simple example of a CNN-based application where the requirements change at run-time due to the changes in its environment. The application is discussed in the context of the existing methodologies reviewed in Section 5.3, and the scenario-based run-time switching (SBRS), our proposed methodology.

The example application performs CNN-based image recognition on a battery powered unmanned aerial vehicle (UAV). The UAV battery capacity defines a power budget, which is available for both the flight and the CNN-based application execution. The distribution of the power budget between the flight and the application is irregular, and depends on the weather conditions, which can change during the run-time (the UAV flight). In a calm weather, the UAV requires less power to fly and can thus spend more power on the CNN-based application. Conversely, when the weather is windy, the UAV requires a large amount of power to fly, and therefore has less power available for the CNN-based application. The weather prediction at the application design time is an impossible task. Nevertheless, the CNN-based application should be designed such that it: 1) meets the power constraint, imposed on the application by the UAV battery and affected by weather conditions; 2) demonstrates high image recognition accuracy (the higher the better).

Figure 5.1 illustrates an example of how the execution of such CNN-based application will transpire, when designed using the existing methodologies and our SBRS. Subplots (a), (b), (c) juxtapose the power available for the application execution (dashed line), against the power used by the application (solid line) during the UAV flight, which lasts 2 hours. The power available for the application execution is dependant on the UAV battery capacity and weather conditions. In this example, we assume that the CNN-based application is allowed to use up to 12 Watts of power in turbulent weather (0 to 0.1 hours and 1.0 to 1.5 hours) and up to 32 Watts of power in calm weather (0.1 to 1.0 hours and 1.5 to 2.0 hours). However, the actual power used by the application is ultimately determined by the application design methodology. Further, the subplots (d), (e), (f) show the image recognition accuracy demonstrated by the application. Subplots (g), (h), (i) show the current charge state (solid line) and minimum charge level (dashed line) of the UAV battery. If the current battery charge reaches the minimum allowed battery level, it may lead to an emergency landing of the UAV.

As a first case, we discuss the multi-objective NAS methodologies [3, 38, 46, 92, 100, 105] for the execution of the example application, that are typically designed and utilized without considering a run-time changing environment. In these methodologies, a CNN is obtained via an automated multi-objective search and characterized with constant accuracy and power consumption. To

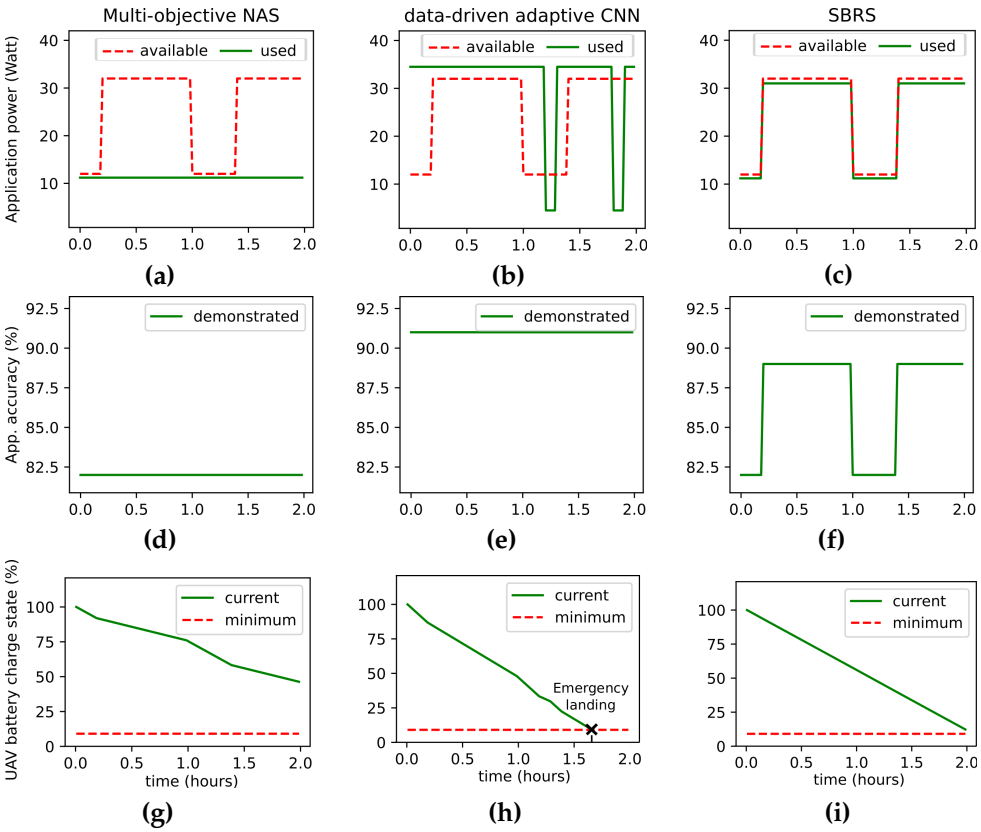


Figure 5.1: *Execution of a CNN-based application, affected by the application environment and designed using different methodologies*

guarantee that the application meets a power constraint, such a CNN has to account for the worst-case scenario, i.e., when the weather is always windy and therefore only 12 Watts are available for the application execution at any moment. In our illustrative example, such a CNN is characterized with 11.2 Watts of power and 82% accuracy (see Figure 5.1(a) and Figure 5.1(d), respectively). As shown in Figure 5.1(g), when the UAV reaches its destination after 2 hours of flight, it still has ≈50% battery charge left. On the one hand, it means that the application always meets the power constraint. On the other hand, the application could have spent ≈40% remaining UAV battery charge by utilizing a more accurate CNN, though demanding additional power . In other words, *the methodologies in [3, 38, 46, 92, 100, 105] can guarantee that the application meets the given platform-aware constraint, but cannot guarantee efficient use of available platform resources*.

As a second case, when the application is designed using data-driven

adaptive methodologies, such as [12, 39, 61, 96, 102], the CNN execution is sensitive to the input data complexity. To process "easy" images, they may use a lower resolution or fewer layers, whereas processing "hard" images requires more computation. In this manner, an adaptive CNN-based application is able to adapt its power consumption depending on the input data complexity, while demonstrating similar accuracy for all the inputs. However, such a CNN cannot adapt to the changing environmental conditions, which can not be explicitly captured in the input images. The application power consumption can change during the application run-time, based on the input images, although these changes may conflict with the application's requirements, driven by the weather conditions. For example, in Figure 5.1(b), between 1.0 and 1.25 hours, the CNN consumes significant amount of power despite the necessity to switch to the low power mode. This may lead to increased UAV power consumption over the flight duration and, eventually, to the violation of the application power constraint, causing an emergency landing as illustrated in Figure 5.1(h). Thus, *the methodologies in [12, 39, 61, 96, 102] are not suitable for CNN-based applications executed at the Edge in changing environment, because these can neither properly adapt the application to the environment variations, nor guarantee that the application constantly meets platform-aware constraints*.

Another case of adaptive CNN-based application methodologies, is where the application can adaptively change the number of floating-point operations (FLOPs) spent on the image recognition, such as those in [39, 107]. However, as shown in numerous works [54, 103, 105] FLOPs is an inaccurate indicator for real-world platform-aware characteristics such as power consumption or throughput. These characteristics depend on many other factors, for instance, the ability of the platform to perform parallel computations, time and energy overheads caused by the data transfers, internal hardware limitations, etc. Consequently, the number of FLOPs spent during the application run-time, neither guarantee that the application meets power constraint nor estimate the application efficiency in terms of real-world platform-aware characteristics. In other words, *even though, the methodologies in [39, 107] enable run-time CNN adaptivity, these cannot be directly deployed for applications with real-world platform-aware requirements and constraints*.

To summarize, the existing works lack a methodology to design an adaptive CNN-based application, for real-world platform-aware requirements and constraints, specifically affected by the environment variations at run-time. The motivation behind our current proposal, SBRS, is to enable such run-time adaptivity. To design an application using our SBRS, we perform multi-objective NAS, similar to those in [3, 38, 46, 92, 100, 105]. However, unlike these

methodologies, we derive multiple CNNs for each scenario. For example, the first scenario for our example application for windy weather, can have an associated CNN with 11.2 Watts power consumption and 82% accuracy. The second scenario, for calm weather, is represented by a CNN with 31.0 Watts power consumption and 89% accuracy. At run-time, the application switches between these scenarios, based on the weather conditions. Additionally, our methodology explicitly defines the switching mechanism based on triggers generated due to an environment change at run-time. The execution of the CNN-based application with SBRS is shown in Figure 5.1 (c), (f), (i). Particularly, Figure 5.1(i) highlights that the application meets the given power constraint, i.e. the UAV battery charge does not go below the minimum level before 2 hours, and SBRS uses all available power to achieve higher application accuracy in comparison with Figure 5.1(d). Thus, *by switching among the scenarios, SBRS guarantees that a CNN-based application, affected by the environment, meets platform-aware constraints while efficiently exploiting the available platform resources to improve its accuracy*.

5.5 SBRS methodology

In this section, we present our novel scenario-based run-time switching (SBRS) methodology, which allows for run-time adaptation of a CNN-based application, executed at the Edge, to changes in the application environment. The general structure of our methodology is given in Figure 5.2. Our methodology accepts as an input a baseline CNN and one or more requirements sets, associated with the CNN-based application. A baseline CNN is an existing CNN (e.g., AlexNet [4], ResNet [36], or another), proven to achieve good results at solving a CNN-based application task (e.g., classification). The requirements sets describe a scope of needs, associated with the devised application. Every application requirements set $r = (r_a, r_t, r_m, r_e)$ specifies the application priority for high accuracy (r_a), high throughput (r_t), low memory cost (r_m), and low energy cost (r_e), respectively. One application can have one or several sets of requirements, characterising the application needs at different times of the application execution. The requirements sets are defined by the application designer at the application design time. As an output, our methodology provides a CNN-based application with SBRS capabilities, able to adapt its characteristics to the changes in the application environment during the application run-time.

Our methodology consists of three main steps. In Step 1 (see Section 5.6), for every set of application requirements r, accepted as an input by our method-

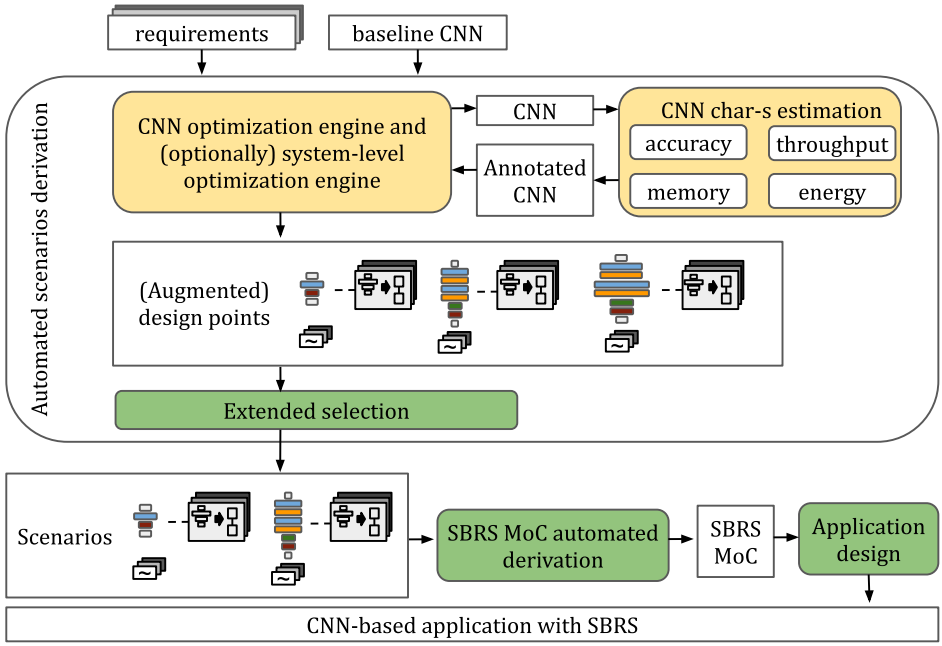


Figure 5.2: *SBRS methodology*

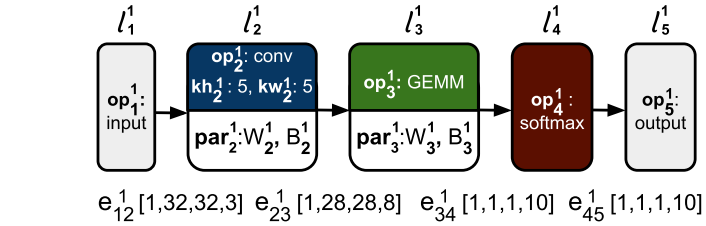
ology, we derive an application scenario, i.e., a CNN which conforms to the given set r of application requirements.

In Step 2, we use the scenarios generated by Step 1, and the algorithm proposed in Section 5.8, to automatically derive a SBRS MoC of a CNN-based application with scenarios. The SBRS MoC, proposed in Section 5.7, captures the scenarios associated with the CNN-based application, and allows for run-time switching among these scenarios. Moreover, the SBRS MoC features efficient reuse of the components (layers and edges) among and within application scenarios, thereby ensuring efficient utilization of the platform memory by the CNN-based application with SBRS.

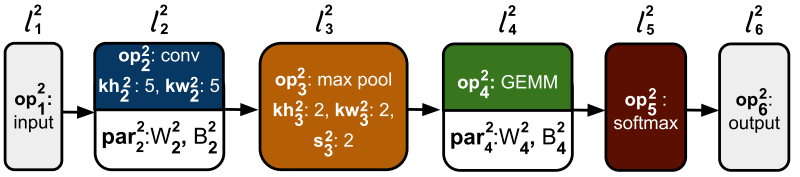
Finally, in Step 3, we use the SBRS MoC derived at Step 2 to design a final implementation of the CNN-based application with SBRS. The final implementation of the CNN-based application performs the application functionality with run-time adaptive switching among the application scenarios, illustrated in Section 5.4, and following the switching protocol presented in Section 5.9.

5.6 Automated scenarios derivation

In this section, we discuss the automated derivation of application scenarios, i.e., CNNs characterized with different accuracy and platform-aware charac-



(a) CNN^1, $r_1 = (1.0, 0, 0, 0)$



(b) CNN^2, $r_2 = (0.2, 0.4, 0, 0.4)$

Figure 5.3: *Application scenarios*

teristics. An example set of $S = 2$ scenarios, derived using our methodology, is shown in Figure 5.3. As mentioned in Section 5.5, every intended scenario $CNN^i, i \in [1, S]$ is first depicted by a user-defined set of requirements $r_i = (r_a, r_t, r_m, r_e)$, where r_a, r_t, r_m, r_e refer to the importance of high CNN accuracy, high CNN throughput, low CNN memory footprint and low CNN energy cost, respectively. Together, these variables constitute the influence factor of each requirement in the scenario by assigning a weight value to the requirements such that $r_a + r_t + r_m + r_e = 1.0$. For example, in scenario CNN^1 shown in Figure 5.3(a) only high accuracy is pivotal, i.e. $r_a = 1.0$, the requirements set is $r_1 = (1.0, 0, 0, 0)$. For scenario CNN^2 shown in Figure 5.3(b), the throughput and energy are critical factors while accuracy is still moderately significant, and the requirements set is defined as $r_2 = (0.2, 0.4, 0, 0.4)$.

To derive a set of scenarios, depicted by their respective sets of requirements, we use a part of the extended CNN design flow shown in Figure 1.5 and explained in Section 1.5. First, the sets of requirements are passed to the *CNN optimization engine*, introduced in Section 1.3. The CNN optimization engine performs automated search for optimal CNN architecture and weights using techniques such as platform-aware NAS [9, 25, 34, 38, 46, 92, 105] and CNN compression [41, 99, 106]. The search results into a set of CNNs, characterized with different architecture, weights, accuracy, and platform-aware characteristics. The platform-aware characteristics of the CNNs may be further improved by the use of the *system-level optimization engine*, introduced in Section 1.5. Recall, that the system-level optimization engine explores and

exploits alternative manners of CNN execution to improve the CNNs characteristics, and produces as an output a set of *augmented design points*, i.e., CNNs, annotated with a specific manner of execution. Finally, the extended selection component, introduced in Section 1.5, selects scenarios from the (augmented) design points, produced using the CNN optimization engine and (possibly) the system-level optimization engine. The selection of every scenario is based on existing multi-objective ranking algorithms [29], able to score a CNN, based on the CNN accuracy and platform-aware characteristics, and a set of requirements, posed on a CNN.

To estimate the accuracy and platform-aware characteristics, the CNN optimization engine and the system-level optimization engine use the CNN characteristics estimation component, briefly introduced in Section 1.3. In our methodology, the CNN characteristics estimation component provides means to evaluate the CNN accuracy, throughput, memory cost, and energy cost.

To evaluate the *accuracy* of a CNN, we use means of existing DL frameworks, such as Keras [19], Pytorch [75], Tensorlow [1], TensorRT [72] and others [74]. These frameworks offer a wide range of a state-of-the-art techniques for evaluating CNN accuracy. Mainly, we use the widely known cross-validation technique [78]. In this technique, a CNN efficiency metric is measured by application of a CNN to a special set of data, called validation dataset [78]. The CNN accuracy is computed as the number of correctly processed input frames to the total number of the CNN input frames. It is important to note that even though we refer to estimation of a CNN as accuracy, it is possible to use alternative estimation metrics suitable to the application, and offered by the DL frameworks. For instance, F-1 score, precision, recall, PR-AUC (Area under curve for precision recall) [89] are some of the metrics that can be used for CNNs estimation as well.

To estimate the platform-aware characteristics of a CNN, we use analytical formulas as well as measurements on the platform. The *memory cost* of a CNN is estimated analytically, using Equation 2.5 explained in Section 2.2. To estimate the *CNN throughput and energy cost* that are notoriously hard to evaluate analytically [54, 60, 103, 105], we use direct measurements on the platform.

5.7 SBRS application model

In this section, we propose our SBRS MoC, which models a CNN-based application with scenarios. The SBRS MoC captures multiple scenarios associated with a CNN-based application, and allows for run-time switching among

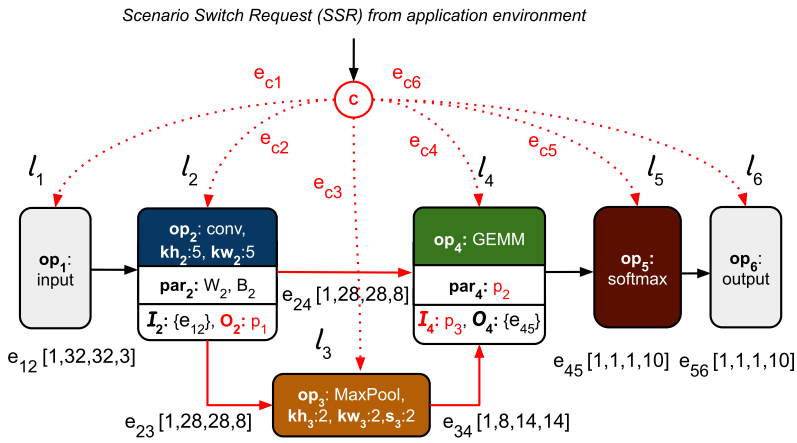


Figure 5.4: *SBRS MoC*

these scenarios. Every scenario in the SBRS MoC is a CNN. Figure 5.4 shows an example of the SBRS MoC, which models a CNN-based application associated with scenarios CNN^1 and CNN^2 shown Figure 5.3 and explained in Section 5.6. In this section, we use the example in Figure 5.4 to explain the SBRS MoC in detail. Formally, the SBRS MoC is defined as a scenarios supergraph, augmented with a control node c and a set of control edges E_c. The scenarios supergraph (see Section 5.7.1), captures all components (layers and edges) in every scenario of a CNN-based application. Therefore, it captures the functionality of every scenario, used by the application. To represent the functionality of a specific scenario, the SBRS MoC uses a sub-graph of the scenarios supergraph. The execution of a specific scenario (i.e., the use of a specific sub-graph of the scenarios supergraph) as well as run-time adaptive switching among the scenarios is determined by the control node c of the SBRS MoC (see Section 5.7.2). Finally, control edges E_c (see Section 5.7.3) specify the communication between the control node c and the scenarios supergraph. The details of the SBRS MoC deployment and inference at the Edge are provided in Section 5.7.4.

5.7.1 Scenarios supergraph

The scenarios supergraph of an SBRS MoC is a graph $SBRS(L, E)$ with a set of layers L which captures the functionality of every layer in every scenario of a CNN-based application, and a set of edges E which captures every data dependency in every scenario of the CNN-based application. Every layer l_i^s of every scenario CNN^s is captured by the functionally equivalent layer $l_n \in L$ of the scenarios supergraph, and every edge e_{ij}^s of every scenario

Table 5.1: *Capturing of scenarios' components (layers and edges) in the scenarios supergraph*

		layers						edges					
SBRS	component	l_1	l_2	l_3	l_4	l_5	l_6	e_{12}	e_{23}	e_{24}	e_{34}	e_{45}	e_{56}
CNN^1	component	l_1^1	l_2^1	-	l_3^1	l_4^1	l_5^1	e_{12}^1	-	e_{23}^1	-	e_{34}^1	e_{45}^1
	control par.	-	O_2=p_1 ={e_{24}}		par_4=p_2= {W_3^1, B_3^1}; I_4=p_3={e_{24}}	-	-	-	-	-	-	-	-
CNN^2	component	l_1^2	l_2^2	l_3^2	l_4^2	l_5^2	l_6^2	e_{12}^2	e_{23}^2	-	e_{34}^2	e_{45}^2	e_{56}^2
	control par.	-	O_2=p_1 ={e_{23}}	-	par_4=p_2= {W_4^2, B_4^2}; I_4=p_3={e_{34}}	-	-	-	-	-	-	-	-
	reuse	all attri-butes of l_1	op_2,s_2, kw_2,kh_2, par_2, I_2	-	op_4,s_4, kw_4,kh_4, O_4	all attri-butes of l_5	all attri-butes of l_6	e_{12}	-	-	-	e_{45}	e_{56}

CNN^s is captured by the functionally equivalent edge $e_{nk} \in E$ of the scenarios supergraph. Table 5.1 shows how layers and edges of scenarios CNN^1 and CNN^2, shown in Figure 5.3 are captured in the scenarios supergraph of the SBRS MoC shown in Figure 5.4. For example, Column 9 in Table 5.1 shows that edge e^1_{12} of scenario CNN^1 and edge e^2_{12} of scenario CNN^2 are captured by edge e_{12} of the scenarios supergraph. We note that edge e_{12} is used by scenario CNN^1 and scenario CNN^2, i.e., edge e^1_{12} is *reused* among the application scenarios.

The reuse of components (layers and edges) of the scenarios supergraph is shown in Row 7 in Table 5.1. The reuse is introduced in the SBRS MoC because it allows for reduction of the CNN-based application memory cost and efficient utilization of the target edge platform memory by the CNN-based application. For example capturing of edge e^1_{12} of scenario CNN^1 and edge e^2_{12} of scenario CNN^2 by edge e_{12} of the SBRS MoC enables to reuse target edge platform memory allocated to data tensors $e^1_{12}.data$ and $e^2_{12}.data$, thereby reducing the application memory cost. Analogously, reuse of weights among and within application scenarios, enables to reuse the edge platform memory allocated to store these weights, thereby reducing the application memory cost. The reuse of a scenarios supergraph component can be full or partial. When a component is *fully reused*, all attributes of the component are reused. For example, layer l_1 of the scenarios supergraph shown in Column 3 is fully reused between scenarios CNN^1 and CNN^2, because all attributes of layer l_1 are reused between the scenarios[1]. When a component is *partially reused*, only some of its attributes are reused. For example, layer l_4 of the scenarios supergraph shown in Column 6 is partially reused between scenarios CNN^1 and CNN^2 because only attributes op_4, s_4, kw_4, kh_4, and O_4 of layer l_4 are reused among the scenarios.

The attributes that are not reused between the scenarios, are specified via run-time adaptive control parameters, introduced into the scenarios supergraph by the SBRS MoC to support partial components reuse. For example, as shown in Row 4 and Row 6, Column 6 in Table 5.1, attributes par_4 and I_4 of supergraph layer l_4 are specified by control parameters p_2 and p_3, respectively. During the application run-time, control parameter p_2 takes values from the set $\{\{W^1_3, B^1_3\}, \{W^2_4, B^2_4\}\}$ and control parameter p_3 takes values from the set $\{\{e_{24}\}, \{e_{34}\}\}$. When $p_2 = \{W^1_3, B^1_3\}$ and $p_3 = \{e_{24}\}$, supergraph layer l_4 is functionally equivalent to layer l^1_3 of scenario CNN^1. When $p_2 = \{W^2_4, B^2_4\}$ and $p_3 = \{e_{34}\}$, supergraph layer l_4 is functionally equivalent to layer l^2_4 of scenario CNN^2.

The capturing of scenarios' components (layers and edges) in the scenarios

[1] Attributes of a layer are defined in Table 2.1 in Section 2.1

supergraph (example of capturing is shown in Table 5.1 explained above) is determined at the application design time, and is stored in the control node c of the SBRS MoC during the application run-time.

5.7.2 Control node

The control node c of the SBRS MoC communicates with the application environment, and determines the execution of scenarios in the application supergraph as well as the switching between these scenarios.

Execution of scenario $CNN^s(L^s, E^s), s \in [1, S]$ captured by the SBRS MoC is defined as an execution sequence ϕ^s. The execution sequence is composed of computational steps, performed in a specific order, determined by the CNN topology and manner of execution as explained in Section 2.2. Every computational step $\phi_i^s \in \phi^s, i \in [1, |L^s|]$ involves execution of scenarios supergraph layer l_n, capturing layer l_i^s of scenario CNN^s. If layer l_n is associated with control parameters, step ϕ_i^s specifies values for these parameters such that layer l_n becomes functionally equivalent to layer l_i^s. For example, the execution sequence of scenario CNN^1 is specified as $\phi^1 = \{(l_1, \emptyset), (l_2, \{(p_1, \{e_{24}\})\}), (l_4, \{(p_2, \{W_3^1, B_3^1\}), (p_3, \{e_{24}\})\}), (l_5, \emptyset), (l_6, \emptyset)\}$. At step $\phi_1^1 = (l_1, \emptyset)$ of sequence ϕ^1 layer l_1 of the scenarios supergraph, capturing layer l_1^1 of scenario CNN^1, is executed. The $\emptyset$ in step ϕ_1^1 specifies that there are no control parameter values set during the execution of ϕ_1^1; at step $\phi_2^1 = (l_2, \{(p_1, \{e_{24}\})\}$ layer l_2 of the scenarios supergraph is executed with control parameter p_1=$\{e_{24}\}$, etc.

The switching between the application scenarios is triggered by the application environment, communicating with the control node c. During the application run-time, control node c can receive a scenario switch request (SSR) from the application environment. Upon receiving the SSR, control node c changes old scenario CNN^o, executed by the node, to a new scenario CNN^n, more suitable for the application needs according to SSR. The switching from scenario CNN^o to scenario CNN^n is performed under the SBRS transition protocol, which will be explained in Section 5.9.

5.7.3 Control edges

The set of control edges E_c specifies control dependencies between the control node c and the supergraph layers L. Every control edge $e_{cn} \in E_c$ transfers control data, such as the aforementioned control parameters needed for the layer execution, from control node c to supergraph layer l_n.

5.7.4 Deployment and inference

When a CNN-based application represented as the SBRS MoC is deployed on an edge platform, all the MoC SBRS scenarios supergraph components (layers and edges) as well as all associated parameters (weights and biases) are placed in the platform memory. During the application run-time, the control node c of the SBRS MoC uses part of these components and parameters to execute one of the application scenarios, captured by the SBRS MoC. The current scenario, also referred as an old scenario, is executed until the control node c receives a scenario switching request (SSR) from the application environment. Upon receiving the SSR, the control node c switches to a new scenario, more suitable for the application needs according to SSR. After the switching is finished, the scenarios supergraph continues to execute the new scenario, until a new SSR is received. If a new SSR is received during an ongoing scenarios switching, it is ignored.

5.8 SBRS MoC automated derivation

In this section, we propose an algorithm - see Algorithm 5 that automatically derives the SBRS MoC, explained in Section 5.7, from a set of S application scenarios $\{CNN^s\}, s \in [1, S]$, provided by the automated scenarios derivation component explained in Section 5.6. Algorithm 5 accepts as inputs: 1) the set of scenarios $\{CNN^s\}, s \in [1, S]$, where every scenario is a CNN, annotated with a specific manner of execution; 2) a set of adaptive layer attributes A.

The set A controls the amount of components reuse exploited by the SBRS MoC by explicitly specifying which attributes of the SBRS MoC layers are run-time adaptive. The more layers' attributes are specified in the set A, the more components reuse is exploited by the SBRS MoC. For example, $A = \emptyset$ specifies that the layers of the SBRS MoC have no runtime-adaptive attributes, i.e., only fully equivalent layers (and their input/output edges) are reused among the scenarios. If $A = \{par\}$, in addition to reuse of fully equivalent layers, the SBRS MoC reuses layers that have different parameters (weights and biases) but matching operator, hyperparameters, and sets of input/output edges.

As an output, Algorithm 5 provides an SBRS MoC, which captures application scenarios $\{CNN^s\}, s \in [1, S]$, and exploits the components reuse specified by set A. Figure 5.4 provides an example of the SBRS MoC, derived using Algorithm 5 for scenarios $\{CNN^1, CNN^2\}$ shown in Figure 5.3, and set $A = \{par, I, O\}$ of adaptive layer attributes.

Algorithm 5: SBRS MoC automated derivation

Input: $\{CNN^s\}, s \in [1, S]$; A
Result: $SBRS(L, E, c, E_c)$

$L \leftarrow \emptyset$; $E \leftarrow \emptyset$; $\Pi \leftarrow \emptyset$; $L^{capt} \leftarrow \emptyset$; $E^{capt} \leftarrow \emptyset$;
for $\text{CNN}^s(L^s, E^s), s \in [1, S]$ **do**
  for $l_i^s \in L^s$ **do**
    find $l_n \in L : eq(l_i^s, l_n, A)$//*Equation 5.1* $\wedge \nexists (l_n, l_j^q) \in L^{capt} : l_j^q$ and l_i^s are executed in parallel;
    if l_n *does not exist* **then**
      $n = |L|$;
      $l_n \leftarrow$ new layer $(type_i^s, op_i^s, -, -, -, -, \Theta_i^s, kh_i^s, kw_i^s, s_i^s, pad_i^s, par_i^s)$;
      $L \leftarrow L + l_n$;
    $L^{capt} \leftarrow L^{capt} + (l_n, l_i^s)$;
  for $e_{ij}^s \in E^s$ **do**
    find $l_k \in L : (l_k, l_i^s) \in L^{capt}$ and $l_n \in L : (l_n, l_j^s) \in L^{capt}$;
    if $\nexists e_{kn} \in E : eq(e_{kn}, e_{ij}^s, A)$ //*Equation 5.2* **then**
      $e_{kn} \leftarrow$ new edge (l_k, l_n);
      $E \leftarrow E + e_{kn}$;
    $E^{capt} \leftarrow E^{capt} + (e_{kn}, e_{ij}^s)$;
for $l_n \in L$ **do**
  if $\exists l_i^s \neq l_j^q : (l_n, l_i^s) \in L^{capt} \wedge (l_n, l_j^q) \in L^{capt}$ **then**
    for $attr \in l_n$ **do**
      for $l_i^s \in L^s : eq(l_i^s, l_n, A), s \in [1, S]$ **do**
        $sattr = attr_i^s \in l_i^s : attr_i^s.name = attr.name$;
        if $sattr.value \neq attr.value \wedge attr.value \notin \Pi$ **then**
          $attr =$ new control parameter p;
          $\Pi \leftarrow \Pi + p$;
for $\text{CNN}^s(L^s, E^s), s \in [1, S]$ **do**
  $\phi^s = \emptyset$;
  for $i \in [1, |L^s|]$ **do**
    find $l_n \in L : (l_n, l_i^s) \in L^{capt}$;
    $P \leftarrow \emptyset$;
    for $attr \in l_n : attr.value = p_q \in \Pi$ **do**
      $sattr = attr_i^s \in l_i^s : attr_i^s.name = attr.name$;
      if $attr.name = I \vee attr.name = O$ **then**
        $value \leftarrow \emptyset$;
        for $e_{ij}^s \in sattr.value$ **do**
          $e = e_{nk} \in E : (e_{nk}, e_{ij}^s) \in E^{capt}$;
          $value \leftarrow value + e$;
      else
        $value = sattr.value$;
      $P \leftarrow P + (p_q, value)$;
    $\phi^s \leftarrow \phi^s + (l_n, P)$;
$c \leftarrow$ new control node $(\{\phi^1, \phi^2, ..., \phi^S\}, L^{capt}, E^{capt})$;
$E_c \leftarrow \emptyset$;
for $l_n \in L$ **do**
  $e_{cn} \leftarrow$ new control edge (c, l_n);
  $E_c \leftarrow E_c + e_{cn}$;
return $SBRS(L, E, c, E_c)$

In Lines 1 to 23, Algorithm 5 generates the scenarios supergraph of the SBRS MoC. In Line 1, it defines an empty set of scenarios supergraph layers L, an empty set of scenarios supergraph edges E, an empty set of control parameters Π, an empty set of captured layers L^{capt}, and an empty set of captured edges E^{capt}. The latter two sets represent capturing of scenarios' components (layers and edges, respectively) in the scenarios supergraph.

In Lines 3 to 9, Algorithm 5 adds layers to the supergraph layers set L. For every layer l_i^s of every scenario CNN^s, Algorithm 5 first checks if set L contains a layer l_n that can be reused to capture layer l_i^s. The check is performed in Line 4 and consists of two parts. First, Algorithm 5 checks functional equivalence of layer l_n and layer l_i^s. This check is performed using Equation 5.1, which compares attributes of layers l_i^s and l^n that are not run-time adaptive (i.e., they are not specified in the set of adaptive attributes A). Then, Algorithm 5 ensures than layer l_n does not capture layer l_j^q, executed in parallel with layer l_i^s. This check is performed using annotation of CNN^s which specifies a manner of execution of CNN^s. If condition in Line 4 is met, layer l_n is used to capture the functionality of layer l_i^s (Line 9 in Algorithm 5). Otherwise, a new layer l_n, capturing the functionality of layer l_i^s, is added to the scenarios supergraph (Lines 5 to 9 in Algorithm 5). To define a new layer, Algorithm 5 specifies a value for each attribute given in Table 2.1 and explained in Section 2.1. The values are listed in the order in which they appear in Table 2.1. If Algorithm 5 specifies a value as symbol "$-$", it means that the respective attribute takes the default value.

$$eq(l_i^s, l_n, A) = \begin{cases} true & \text{if } attr_n = attr_i^s, \forall attr \notin A \\ false & \text{otherwise} \end{cases} \tag{5.1}$$

In Lines 10 to 15, Algorithm 5 adds edges to the supergraph edges set E such that 1) every edge e_{ij}^s of every scenario CNN^s is captured in a supergraph edge e_{kn}, and 2) functionally equivalent edges are reused among the scenarios. To check the functional equivalence of a supergraph edge e_{kn} and edge e_{ij}^s of scenario CNN^s, Algorithm 5 uses Equation 5.2.

$$eq(e_{ij}^s, e_{nk}, A) = \begin{cases} true & \text{if } eq(l_i^s, l_n, A) \wedge eq(l_j^s, l_k, A) \\ false & \text{otherwise} \end{cases} \tag{5.2}$$

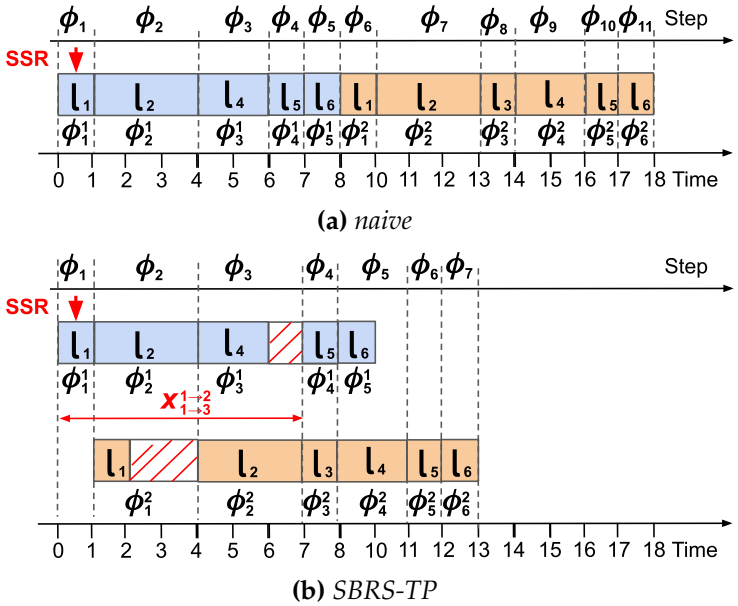
In Lines 16 to 23, Algorithm 5 introduces control parameters into the partially reused layers of the scenarios supergraph to capture those attributes that cannot be reused among the scenarios. For example, to capture attribute I_4

of scenarios supergraph layer l_4, shown in Figure 5.4, Algorithm 5 introduces control parameter p_3 into layer l_4 (as explained in Section 5.7).

In Lines 24 to 44, Algorithm 5 augments the scenarios supergraph, derived in Lines 2 to 23, with a control node c and a set of control edges E_c. In Lines 24 to 39, it generates execution sequence ϕ^s for every scenario CNN^s, captured by the scenarios supergraph. Every computational step $\phi_i^s, i \in [1, |L^s|]$ of the sequence ϕ^s is derived in Lines 26 to 38. In Line 27, Algorithm 5 determines layer l_n of scenarios supergraph, capturing functionality of layer l_i^s of scenario CNN^s. In Lines 28 to 38, Algorithm 5 derives set P of parameter-value pairs that specifies the values for every control parameter p_q associated with layer l_n. In Lines 29 to 38, Algorithm 5 visits every attribute $attr$ of layer l, specified as control parameter p_q, and determines the value taken by the parameter p_q (and, therefore, by attribute $attr$) at the execution step ϕ_i^s. In Line 30, Algorithm 5 finds attribute $sattr$ of layer l_i^s, corresponding to the attribute $attr$ of layer l_n. For example, if attribute $attr \in l_n$ is a set of parameters par of layer l_n, Algorithm 5 finds attribute $sattr \in l_i^s$, which is a set parameters par_i^s of layer l_i^s. If attribute $attr$, specified by the control parameter p_q, is a list of input or output edges of layer l (the condition in Line 31 is met), the value for parameter p_q is specified in Lines 32 to 35 of Algorithm 5, as a subset of supergraph edges, functionally equivalent to the corresponding subset of edges in scenario CNN^s. Otherwise, the value of parameter p_q is specified in Line 37 of Algorithm 5 as the value of attribute $sattr$ of layer l_i^s. In Line 40, Algorithm 5 creates a new control node c, which stores the execution sequence ϕ^s of every scenario as well as sets L^{capt} and E^{capt} that specify capturing of the components (layers and edges) of every scenario by the scenario supergraph. In Lines 41 to 44, Algorithm 5 creates a set of control edges E_c, such that for every scenarios supergraph layer l_n, set E_c contains a control edge e_{cn}, representing control dependency between layer l_n and the control node c. Finally, in Line 45, Algorithm 5 returns the SBRS MoC, capturing the functionality of every scenario $CNN^s, s \in [1, S]$, associated with the CNN-based application.

5.9 Transition protocol

In this section, we present our novel transition protocol, called SBRS-TP, that ensures efficient switching between scenarios of a CNN-based application, represented using the SBRS MoC. As explained in Section 5.7, the control node c of the SBRS MoC can perform switching from an old application scenario CNN^o to a new application scenario CNN^n, upon receiving a scenario switch



(a) *naive*

(b) *SBRS-TP*

Figure 5.5: *Switching from scenario* CNN^1 *to scenario* CNN^2

request (SSR) from the application environment. In the SBRS MoC, where the execution of scenarios CNN^o and CNN^n is represented using execution sequences ϕ^o and ϕ^n, respectively, switching between scenarios CNN^o and CNN^n means switching between the sequences ϕ^o and ϕ^n. We evaluate the efficiency of such switching by the response delay Δ, defined as the time between a SSR arrival during the execution of the current scenario CNN^o, and the production of the first output data by the new scenario CNN^n. The larger the delay Δ is, the less responsive the application is during a scenarios transition, thus the less efficient the switching is.

The most intuitive way of switching between scenarios CNN^o and CNN^n, hereinafter referred to as naive switching, is to start the execution of the new scenario CNN^n after all computational steps of the old scenario CNN^o are executed. An example of the naive switching is shown in Figure 5.5(a), where the CNN-based application represented by the SBRS MoC from Figure 5.4 switches from scenario CNN^1 to scenario CNN^2 upon receiving a SSR at the first execution step of scenario CNN^1. The layers of scenario CNN^1 and scenario CNN^2 are executed in a sequential manner, explained at the end of Section 2.2. The upper axis in Figure 5.5(a) shows steps $\phi_i, i \in [1, 11]$, performed by the control node c during the scenarios switching. For example, Figure 5.5(a) shows that at step ϕ_1 (upon SSR arrival), control node c schedules step ϕ_1^1 of scenario CNN^1 for execution. The lower axis in Figure 5.5(a)

indicates the start and end time of every step ϕ_i performed by the control node c. Every rectangle, annotated with layer l_n in Figure 5.5(a), shows the time needed to execute layer l_n. The response delay Δ of the naive switching, shown in Figure 5.5(a), is computed as $18 - 0.5 = 17.5$, where 0.5 is the time of SSR arrival and 18 is the time when scenario CNN^2 produces its first output, i.e., finishes its last step ϕ_6^2.

We note that this response delay can be reduced. Figure 5.5(b) shows an example of an alternative switching mechanism, referred to as the SBRS-TP transition protocol. Unlike in the naive switching, in SBRS-TP, every step $\phi_i^2, i \in [1,6]$ of the new scenario CNN^2 is executed as soon as possible. For example, step ϕ_1^2 of the new scenario CNN^2 is executed at step ϕ_2, where ϕ_2 is the earliest step after the SSR arrival, at which step ϕ_1^2 can be executed. Step ϕ_1^2 cannot be executed earlier, i.e., at step ϕ_1, due to the components reuse. As explained in Section 5.7, layer l_1 and the platform resources allocated for execution of this layer are reused between scenarios CNN^1 and CNN^2, and thus cannot be used by scenarios CNN^1 and CNN^2 simultaneously. At step ϕ_1, layer l_1 is used by scenario CNN^1, executing step ϕ_1^1, and therefore, cannot be used for execution of step ϕ_1^2 of scenario CNN^2. However, step ϕ_1^2 of the new scenario CNN^2 can be executed at step ϕ_2, in parallel with step ϕ_2^1 of the old scenario CNN^1, because no components reuse occurs between these steps: step ϕ_2^1 uses layer l_2 for its execution, while step ϕ_1^2 uses layer l_1 (where $l_1 \neq l_2$) for its execution. Analogously, step ϕ_2^2 of the new scenario CNN^2 is executed at step ϕ_3, where ϕ_3 is the earliest step after the SSR arrival, at which step ϕ_2^2 can be executed. As explained in Section 5.7, according to the execution order adopted by scenario CNN^2, step ϕ_2^2 should be executed after step ϕ_1^2. Thus, in the example shown in Figure 5.5(b), step ϕ_2^2 should start after step ϕ_2, at which step ϕ_1^2 is executed. Moreover, step ϕ_2^2 of the new scenario CNN^2 cannot be executed at step ϕ_2, because at step ϕ_2 reused layer l_2, required for execution of step ϕ_2^2, is occupied by step ϕ_2^1 of scenario CNN^1. However, step ϕ_2^2 can be executed at step ϕ_3, when layer l_2 that is required for execution of step ϕ_2^2 is not occupied by scenario CNN^1, and step ϕ_1^2 is already executed. The response delay Δ of the switching mechanism shown in Figure 5.5(b) is $13 - 0.5 = 12.5$, and is much smaller than the response delay $\Delta = 17.5$ of the naive switching shown in Figure 5.5(a). Thus, the switching mechanism shown in Figure 5.5(b) is more efficient compared to the naive switching.

Our methodology performs efficient switching between scenarios of a CNN-based application using the SBRS-TP transition protocol, as illustrated in Figure 5.5(b). The SBRS-TP is carried out in two phases: the analysis phase, and the scheduling phase. The analysis phase is performed during

Algorithm 6: SBRS-TP analysis phase

```
   Input: φ^o, φ^n
   Result: X^{o→n}
1  X^{o→n} ← ∅; x = 0;
2  for i ∈ [1, |L^n|] do
3      (l_k, P^n) ← φ_i^n;
4      for φ_j^o ∈ φ^o do
5          (l_z, P^o) ← φ_j^o;
6          if k = z then
7              if j ≥ x then
8                  x = j;
9      X^{o→n} ← X^{o→n} + x;
10     x = x + 1;
11 return X^{o→n}
```

the application design time, for every pair (CNN^o, CNN^n), with $o \neq n$, of the CNN-based application scenarios. During this phase, for every step ϕ_i^n of the new scenario CNN^n, SBRS-TP derives a minimum delay in steps $x_{1\rightarrow i}^{o\rightarrow n}$ between step ϕ_i^n and the first step ϕ_1^o of the old scenario CNN^o. The delay $x_{1\rightarrow i}^{o\rightarrow n}$ is computed with respect to the data dependencies within scenarios CNN^o and CNN^n, and the components reuse between these scenarios, as discussed above. An example of delay $x_{1\rightarrow i}^{o\rightarrow n}$ is delay $x_{1\rightarrow 3}^{1\rightarrow 2} = 3$ of step ϕ_3^2, shown in Figure 5.5(b). Delay $x_{1\rightarrow 3}^{1\rightarrow 2} = 3$ specifies that step ϕ_3^2 of the new scenario CNN^2 cannot start earlier than 3 steps after the first step ϕ_1^1 of the old scenario CNN^1 has started, i.e., earlier than step ϕ_4.

The analysis phase of the SBRS-TP is presented in Algorithm 6. Algorithm 6 accepts as inputs execution sequences ϕ^o and ϕ^n, representing the old scenario CNN^o and the new scenario CNN^n, respectively. As an output, Algorithm 6 provides a set $X^{o\rightarrow n}$, where every element $x_{1\rightarrow i}^{o\rightarrow n} \in X^{o\rightarrow n}$, with $i \in [1, |L^n|]$, is the minimum delay in steps between step ϕ_i^n of the new scenario CNN^n and the first step ϕ_1^o of the old scenario CNN^o. An example of set $X^{o\rightarrow n}$ generated by Algorithm 6 for the scenario switching, shown in Figure 5.5(b), is the set $X^{1\rightarrow 2} = \{1, 2, 3, 4, 5, 6\}$. In Line 1, Algorithm 6 defines an empty set $X^{o\rightarrow n}$ and a variable x, equal to 0. Variable x is a temporary variable used to store delay $x_{1\rightarrow i}^{o\rightarrow n}$ of every execution step ϕ_i^n in Lines 2 to 10 of Algorithm 6. In Lines 2 to 10, Algorithm 6 visits every step ϕ_i^n of the new scenario CNN^n and computes delay $x_{1\rightarrow i}^{o\rightarrow n}$ associated with this step. In Lines 4 to 8, Algorithm 6 increases delay $x_{1\rightarrow i}^{o\rightarrow n}$, stored in variable x, with respect to the components reuse, as discussed above. It visits every step ϕ_j^o of the old scenario CNN^o, and if step ϕ_j^o and step ϕ_i^n share a reused layer (the condition in Line 6 is

Algorithm 7: SBRS-TP scheduling phase

Input: $\phi^o, \phi^n, X^{o \to n}$
1 $q = 1; i = 1; j = step^o_{SSR}$;
2 wait until step ϕ^o_j is finished; $j = j + 1; q = q + 1$;
3 **while** $j \leq |L^o|$ **do**
4 start ϕ^o_j; $j = j + 1$;
5 **if** $q \geq x^{o \to n}_{1 \to i} - step^o_{SSR} + 2$ **then**
6 start ϕ^n_i; $i = ((i + 1) \mod |L^n|)$;
7 wait until started scenarios' steps are finished; $q = q + 1$;
8 **while** $i \leq |L^n|$ **do**
9 start ϕ^n_i;
10 wait until ϕ^n_i finishes; $i = i + 1; q = q + 1$;

met), it delays the execution of step ϕ^n_i until step ϕ^o_j is finished. In Line 9, Algorithm 6 adds the delay of step ϕ^n_i, stored in variable x, to the set $X^{o \to n}$. In Line 10, Algorithm 6 increases the delay by one step, thereby defining an initial delay for the next step ϕ^n_{i+1} of the new scenario CNN^n. Finally, in Line 11, Algorithm 6 returns the set $X^{o \to n}$. The set $X^{o \to n}$ derived using Algorithm 6 for every pair of scenarios (CNN^o, CNN^n) is stored in the control node c of the scenarios supergraph, and used by the scheduling phase of the SBRS-TP at the application run-time.

The scheduling phase of the SBRS-TP is performed by the control node c during the application run-time, upon arrival of an SSR. During this phase, control node c performs switching from the old scenario CNN^o to the new scenario CNN^n, such that the steps of the new scenario CNN^n are executed as soon as possible with respect to the data dependencies within scenario CNN^n and the components reuse between scenarios CNN^o and CNN^n (as discussed above). The scheduling phase of the SBRS-TP is given in Algorithm 7. It accepts as inputs execution sequences ϕ^o and ϕ^n of the old scenario CNN^o and the new scenario CNN^n, respectively, and the set $X^{o \to n}$ derived by Algorithm 6 for scenarios CNN^o and CNN^n at the SBRS-TP analysis phase. In Line 1, Algorithm 7 defines variables i, j, and q, representing indexes of the current step ϕ^n_i of the new scenario CNN^n, current step ϕ^o_j in the old scenario CNN^o, and current step ϕ_q performed by the control node c, respectively. Upon SSR arrival, $i = 1$, $q = 1$, and $j = step^o_{SSR}$ where $step^o_{SSR} \geq 1$ is the step in the old scenario CNN^o at which the SSR arrived. For the example shown in Figure 5.5(b), $step^o_{SSR} = 1$ because SSR arrives at step ϕ^1_1 of the old scenario CNN^1. In Line 2, Algorithm 7 performs the first step ϕ_1 of the scenarios switching. During this step, Algorithm 7 waits until step ϕ^o_j, during which the SSR

arrived, finishes. In Lines 3 to 7, Algorithm 7 schedules the remaining steps of the old scenario CNN^o, until scenario CNN^o is finished (the condition in Line 3 is false) and, if possible, schedules steps of the new scenario CNN^n in parallel with the steps of the old scenario CNN^o. Step ϕ_i^n of the new scenario CNN^n can start in parallel with step ϕ_j^o of the old scenario CNN^o if the minimum distance $x_{1\rightarrow i}^{o\rightarrow n}$ between steps ϕ_1^o and ϕ_i^n is observed (the condition in Line 5 is met). In Line 7, Algorithm 7 waits until the steps of scenarios CNN^o and CNN^n, started in Lines 4 to 6, finish. In Lines 8 to 10, Algorithm 7 schedules the remaining steps of scenario CNN^n, until scenario CNN^n produces an output data (the condition in Line 8 is false). After Algorithm 7 finishes, scenario CNN^n becomes the current scenario and will be executed for every input given to the CNN-based application until the next SSR.

5.10 Experimental Study

To evaluate our novel SBRS methodology, we perform an experiment, where we apply our methodology to real-world CNN-based applications with scenarios. We conduct our experiment in four steps. The first three steps perform in-depth per-step analysis of our methodology and demonstrate the merits of our methodology through three real-world CNN-based applications from different domains. The fourth step compares our methodology to the most relevant existing work.

In Step 1 (Section 5.10.1), we derive scenarios for three real-world CNN-based applications with scenarios. We illustrate the diversity among the selected scenarios and compare the use of multiple scenarios (CNNs), enabled by our methodology, to use of a single CNN, adopted by the state-of-the-art design flow, introduced in Section 1.3 and shown in Figure 1.3. By performing this experiment, we evaluate the effectiveness of run-time adaptivity, offered by our methodology.

In Step 2 (Section 5.10.2), we use Algorithm 5, proposed in Section 5.7.1, to automatically generate SBRS MoCs for the CNN-based applications, derived at Step 1. For every application, we generate two SBRS MoCs with different sets of adaptive layer attributes A: $A = \{I, O, par\}$ and $A = \{I, O\}$, respectively. We measure and compare the memory cost of every CNN-based application, when the application is represented as 1) the SBRS MoCs with $A = \{I, O, par\}$; 2) an SBRS MoC with $A = \{I, O\}$; 3) a set of scenarios, where every scenario is represented as a CNN model, explained in Section 2.1. By performing this experiment, we evaluate the efficiency of the memory reuse, exploited by the SBRS MoC, proposed in Section 5.7.

In Step 3 (Section 5.10.3), we measure and compare the responsiveness of the CNN-based applications, represented as SBRS MoCs, derived in Step 2, during the scenarios switching, when switching is performed: 1) under our SBRS-TP transition protocol; 2) using the naive switching mechanism. By performing this experiment, we evaluate the efficiency of the SBRS-TP transition protocol, proposed in Section 5.9.

Finally, in Step 4 (Section 5.10.4), we compare our SBRS methodology with the most relevant existing work. As explained in Section 5.3 and demonstrated in Section 5.4, none of the currently existing works can design an adaptive CNN-based application, which considers platform-aware requirements and constraints that are specifically affected by the environment changes at run-time. Within this context, none of the existing works is completely comparable to our methodology. Nonetheless, we perform a partial comparison between our methodology and the most relevant existing work. Among the existing works, reviewed in Section 5.3 and Section 5.4, the MSDNet adaptive CNN work [39] is the most relevant to our methodology. Similarly to our methodology and unlike other reviewed existing work, the methodology in [39] associates a CNN-based application with multiple alternative CNNs that are characterized with different trade-offs between accuracy and resources utilization, and can be used to process application inputs of any complexity. Additionally, both the work in [39] and our methodology provide means to reduce the memory cost of a CNN-based application by reusing the memory among the alternative CNNs. In this sense, the methodology in [39] and our SBRS methodology can be compared via 1) run-time adaptive trade-offs between application accuracy and resources utilization; 2) memory efficiency. In Section 5.10.4, we perform such comparison, using the image recognition CIFAR-10 dataset [51].

To perform the measurements, required for Step 1 to Step 4, we implement the executable CNN-based applications, using the TensorRT Deep Learning framework and operators library [72], providing state-of-the-art performance of deep learning inference on the NVIDIA Jetson TX2 embedded device [71], and custom C++ code. The TensorRT library is used to implement the functionality of CNN layers and edges. The custom C++ code implements the run-time adaptive functionality of the applications.

5.10.1 Automated scenarios derivation

In this section, we derive scenarios for three CNN-based applications from two different domains, namely Human Activity Recognition (HAR) and image classification. We used the PAMAP2 [79] dataset for HAR and the Pascal VOC [27]

Table 5.2: *CNN-based applications*

App	task	baseline CNN	dataset	app. requirements sets
Pascal VOC	Image recongition	ResNet [36]	Pascal VOC [27]	$r_1 = (1.0, 0.0, 0.0, 0.0)$ $r_2 = (0.7, 0.0, 0.3, 0.0)$ $r_3 = (0.6, 0.1, 0.0, 0.3)$ $r_4 = (0.5, 0.5, 0.0, 0.0)$ $r_5 = (0.1, 0.1, 0.4, 0.4)$
PAMAP2	Human activity monitoring	PAMAP (CNN-2) [69]	PAMAP2 [79]	$r_1 = (1.0, 0.0, 0.0, 0.0)$ $r_2 = (0.2, 0.4, 0.0, 0.4)$ $r_3 = (0.5, 0.0, 0.0, 0.5)$ $r_4 = (0.5, 0.5, 0.0, 0.0)$
CIFAR-10	Image recognition	ResNet [36]	CIFAR-10 [51]	$r_1 = (1.0, 0.0, 0.0, 0.0)$ $r_2 = (0.25, 0.25, 0.25, 0.25)$ $r_3 = (0.5, 0.25, 0.0, 0.25)$ $r_4 = (0.5, 0.0, 0.0, 0.5)$

and CIFAR-10 [51] datasets for image classification. PAMAP2 has data from body-worn sensors and predicts the activity performed by the wearer, while Pascal VOC and CIFAR-10 are multi-label image classification datasets with 20 classes and 10 classes, respectively. The sensor data in PAMAP2 is down-sampled to 30 Hz and a sliding window approach with a window size of 3s (100 samples) and a step size of 660ms (22 samples) is used to segment the sequences.

The main features and requirements for each CNN-based application are listed in Table 5.2. Column 1 lists the applications names, corresponding to the names of the datasets, the applications are using. Hereinafter, we refer to the applications by their names; Column 2 shows the task performed by the applications; Column 3 lists the baseline CNN that was deployed to perform the application tasks; Column 4 lists the real-world datasets, which were used to train and validate the applications' baseline CNNs; Column 5 shows sets of application requirements $r_i, i \in [1, S]$, where every set r_i characterizes a scenario, associated with the CNN-based application, S is the total number of CNN-based application scenarios. The applications use extremely different baseline CNNs (from the deep and complex ResNet based topology [36] to the small and shallow PAMAP topology) and diverse datasets (from the large Pascal VOC [27] dataset to the small PAMAP2 [79] and CIFAR-10 [51] datasets). The ResNet based baseline topologies for VOC and CIFAR-10 application are custom Resnets, both of which are smaller than the popular ResNet-18. This leads to diversity in scenarios and SBRS MoCs, derived for these applications and, thereby providing a sufficient basis for evaluation of the effectiveness of

Table 5.3: *Algorithm parameters for platform-aware NAS [82]*

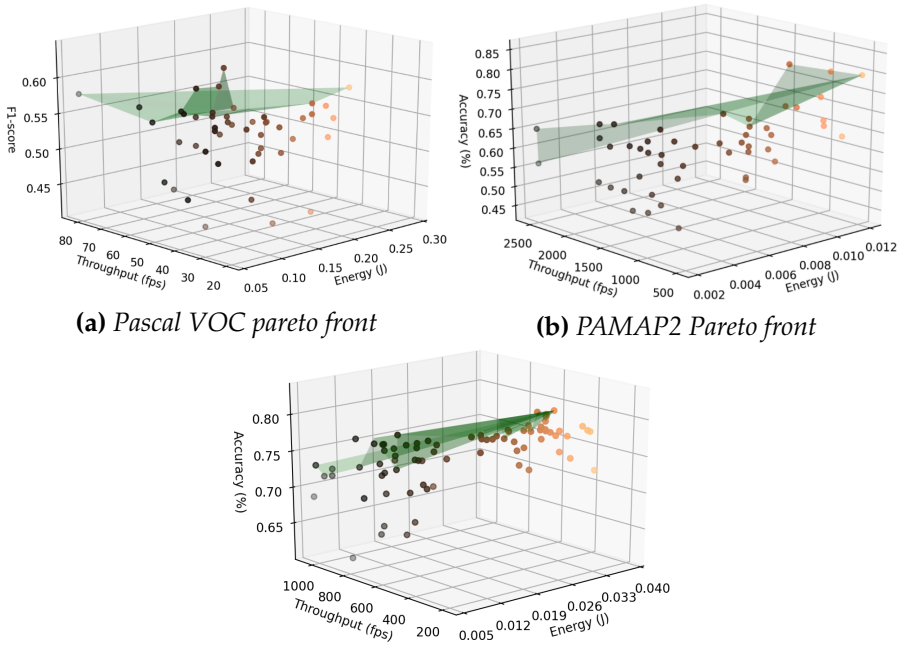
Parameter		VOC	PAMAP2	CIFAR10
Mutation change rate	ϱ_m	0.10	0.12	0.12
Mutation probability	P_m	0.3	0.3	0.3
Initial Crossover probability	$P_r(0)$	0.3	0.4	0.3
Population size	N_p	60	50	100
No of iterations	N_g	30	60	120
Population replacement rate	Ω	0.02	0.03	0.02
Training Parameters	τ_{params}			
Training size per iteration		1 epoch	1/5 epoch	1/8 epoch
Optimizer		Adam	Adam	Adam
Learning rate		$1e^{-3}$	$1e^{-4}$	$1e^{-3}$
Batch size		10	50	64

our methodology.

To derive the scenarios for each application, described in Table 5.2, we used the multi-objective platform-aware NAS methodology proposed in [82] and the scenarios selection based on the ranking dominance concept introduced in [52]. In addition to the baseline CNN and the dataset, specified in Table 5.2, the platform-aware NAS methodology in [82] accepts as input a set of NAS hyper-parameters. The NAS hyper-parameters used in our experiments are summarized in Table 5.3. For the explanation of the NAS hyper-parameters, we respectfully refer the reader to work [82].

The methodology in [82] performs automated search for optimal CNNs, which arrives at a set of CNNs pareto-optimal in terms of accuracy, memory, throughput and energy characteristic. Every CNN in the set is executed in a sequential manner, explained in Section 2.2. The pareto-fronts obtained for Pascal VOC, PAMAP2 and CIFAR-10 applications listed in Table 5.2, are shown in Figure 5.6(a), Figure 5.6(b) and Figure 5.6(c), respectively. The pareto-fronts are shown in three-dimensional (accuracy, throughput and energy cost) space to allow for a comprehensible visualization, while the actual design points exist in four-dimensional (accuracy, throughput, energy cost, and memory cost) space.

The scenarios selected from the pareto-fronts shown in Figure 5.6 for the three multi-scenario applications given in Table 5.2, are presented in Table 5.4, where Column 1 shows the CNN-based applications; Column 2 shows the requirements sets, depicting scenarios, associated with every application; Column 3 shows the scenarios, derived for the requirements sets given in Column 2; Columns 4 to 7 show the accuracy and platform-aware characteristics of every scenario given in Column 3. For the PAMAP2 and CIFAR-10 applica-



(a) *Pascal VOC pareto front*

(b) *PAMAP2 Pareto front*

(c) *CIFAR-10 Pareto front*

Figure 5.6: *Pareto-fronts based on 3 evaluation parameters, namely, accuracy (F1-score for Pascal VOC), throughput and energy*

Table 5.4: *Scenarios derived from pareto-fronts shown in Figure 5.6 for three applications shown in Table 5.2*

App.	app. req. set	scenario	Accuracy (PR-AUC or %)	Throughput (fps)	Memory (MB)	Energy (J)
Pascal VOC	r_1	CNN^1	77.78	15.41	292.61	0.384
	r_2	CNN^2	76.28	21.78	210.69	0.281
	r_3	CNN^3	77.69	20.26	242.72	0.291
	r_4	CNN^4	73.99	59.27	155.48	0.101
	r_5	CNN^5	72.85	75.07	130.21	0.078
PAMAP2	r_1	CNN^1	94.17	510.20	10.02	0.0083
	r_2	CNN^2	91.34	1333.33	4.30	0.0033
	r_3	CNN^3	92.56	970.87	4.86	0.0037
	r_4	CNN^4	92.93	1052.63	4.11	0.0039
CIFAR-10	r_1	CNN^1	94.86	231.80	52.87	0.0242
	r_2	CNN^2	92.84	754.15	13.07	0.0055
	r_3	CNN^3	93.46	538.79	18.30	0.0081
	r_4	CNN^4	94.46	403.71	28.07	0.0121

tions, the accuracy is estimated using the cross-validation technique [78] and measured in percent. For Pascal-VOC, accuracy was estimated as the PR-AUC (Area under precision-recall curve) [89]. Columns 5 to 7 show the scenario throughput (in frames per second), memory (in MegaBytes) and Energy (in Joules), respectively.

Columns in 4 to 7 in Table 5.4 clearly demonstrate that scenarios (CNNs) obtained for different application requirements have vastly different characteristics. If Pascal VOC, PAMAP2, and CIFAR-10 CNN-based applications would use only one of the selected scenarios, as proposed in the state-of-the-art design flow, shown in Figure 1.3 and explained in Section 1.3, the applications' needs would not be optimally served.

For example, let us assume that the Pascal VOC application, shown in Row 2 in Table 5.2 and associated with scenarios CNN^1, CNN^2, CNN^3, and CNN^4, shown in Row 2 in Table 5.4: 1) only uses scenario CNN^1 when is designed using the state-of-the-art design flow; 2) adaptively switches between scenarios CNN^1, CNN^2, CNN^3 and CNN^4, when designed using our proposed methodology. Under requirements set r_4, specifying that high CNN throughput is as important as high CNN accuracy, the application would use CNN^1 when is designed using the state-of-the-art design flow, and CNN^4 when is designed using our proposed methodology. Compared to CNN^1, CNN^4 demonstrates 3.8 times higher throughput and only 4% lower accuracy. Thus, the Pascal VOC application would better serve the application needs, specified in the requirements set r_4.

The example above shows that our methodology ensures more efficient serving of CNN-based applications affected by the environment at run-time when compared to the the state-of-the-art design flow shown in Figure 1.3 and explained in Section 1.3.

5.10.2 SBRS MoC memory reuse efficiency

In this experiment, we measure and compare the memory cost of every CNN-based application, presented in Table 5.2 in Section 5.10.1, when the application is represented as: 1) an SBRS MoC with a set of adaptive layer attributes $A = \{I, O, par\}$; 2) an SBRS MoC with a set of adaptive layer attributes $A = \{I, O\}$; 3) a set of scenarios, where every scenario is represented as a CNN and no memory is reused within or among the CNNs. The results of this experiment are given in Table 5.5.

In Table 5.5, Column 1 lists the CNN-based applications with scenarios, explained in Section 5.10.1. Column 2 shows the sets of adaptive layer attributes A, used by Algorithm 5 to generate the SBRS MoCs for the CNN-based appli-

Table 5.5: *SBRS MoC memory reuse efficiency evaluation*

Application	A	Memory use (MB) M^{SBRS}	Memory use (MB) M^{naive}	memory reduction (%)
Pascal VOC	$\{I, O, par\}$	230	1032	78
	$\{I, O\}$	547		47
PAMAP 2	$\{I, O, par\}$	22.43	23.28	3.64
	$\{I, O\}$	23.21		0.31
CIFAR-10	$\{I, O, par\}$	83.3	112.31	25.9
	$\{I, O\}$	107.17		4.57

cations. Column 3 shows the memory use M^{SBRS} (in MB) of the CNN-based applications, represented as the SBRS MoCs. As shown in Columns 2 and 3 of Table 5.5, the more attributes are specified in the set A, the more memory is reused by the application, and the application memory cost is less. For example, as shown in Rows 3-4, Columns 2-3 in Table 5.5, Pascal VOC uses 230 MB of platform memory, when generated with $A = \{I, O, par\}$ and 547 MB of platform memory, when generated with $A = \{I, O\}$. Column 4 in Table 5.5 shows the memory use M^{naive} (in MB) of the CNN-based applications, when every application is represented as a set of scenarios and no memory reuse is exploited by the application. Column 5 in Table 5.5 shows the memory reduction (in %), enabled by the memory reuse, exploited by our proposed SBRS MoC. The memory reduction is computed as $(M^{naive} - M^{SBRS}) / M^{naive} * 100\%$, where M^{SBRS} and M^{naive} are listed in Columns 3 and 4, respectively. As shown in Column 5, the memory reuse, exploited by the SBRS MoC, varies for different applications: Pascal VOC (Row 3 to Row 4) demonstrates high (47% - 78%) memory reduction; PAMAP2 (Row 5 to Row 6) demonstrates low (0.31% - 3.64%) memory reduction; CIFAR-10 (Row 7 to Row 8) demonstrates (4.57% - 25.9%) memory reduction, which is higher, compared to PAMAP2 but lower than Pascal VOC. The difference occurs due to the different amounts of components reuse exploited by the Pascal VOC, PAMAP2 , and CIFAR-10 applications . Pascal VOC has 5 scenarios, where every scenario is a deep CNN with a larger number of similar layers. In other words, Pascal VOC is characterised by a large amount of repetitive CNN components, reused by the SBRS MoC (see Section 5.7.1), which leads to a significant memory reduction. PAMAP2 has 4 scenarios, compared to 5 scenarios of Pascal VOC, and every scenario in PAMAP2 has less layers and edges than the scenarios of Pascal VOC. Thus, in PAMAP2, the SBRS MoC can reuse only a small number of components, which leads to a small memory reduction. CIFAR-10 has 4 scenarios, and every scenario in CIFAR-10 has less layers and edges than the scenarios of Pascal VOC, but more layers and edges than the scenarios of

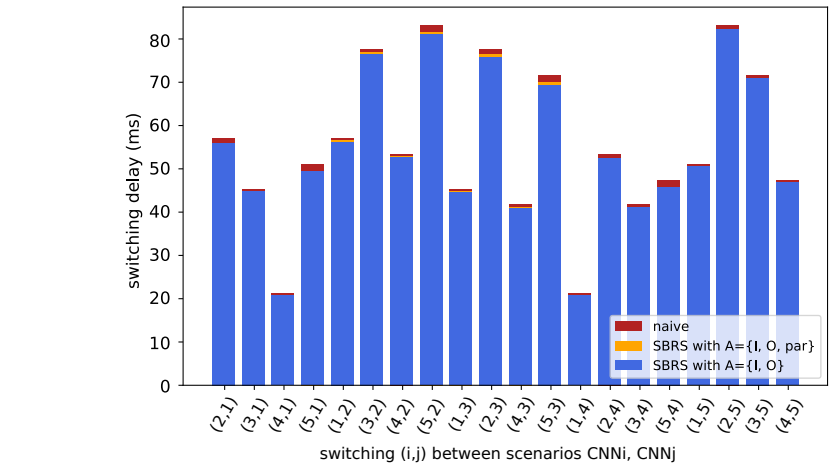
PAMAP2. Thus, in CIFAR-10, the SBRS MoC can reuse less components than in Pascal VOC, but more components than in PAMAP2.

5.10.3 SBRS-TP efficiency

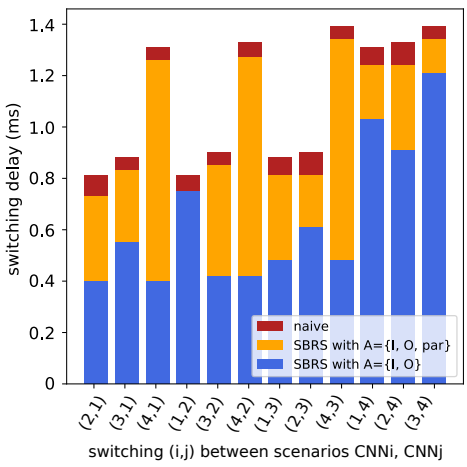
In this experiment, for every CNN-based application, explained in Section 5.10.1, and represented as two functionally equivalent SBRS MoCs with sets of adaptive attributes $A = \{I, O\}$ and $A = \{I, O, par\}$, respectively, we measure and compare the application responsiveness during the scenarios switching, when the switching is performed using: 1) the naive switching mechanism; 2) the SBRS-TP transition protocol. The results of this experiment for Pascal VOC, PAMAP2 and CIFAR-10 are shown as bar charts in Figure 5.7, subplots (a), (b), and (c), respectively. Every pair (o, n), shown along the horizontal axis in the subplots denotes switching between a pair (CNN^o, CNN^n), $o \neq n$ of the application scenarios, performed upon arrival of a Scenarios Switch Request (SSR) at the first step of the old scenario ($step^o_{SSR}$=1). For example, pair (2, 1) shown in Figure 5.7(b), denotes switching between scenarios CNN^2 and CNN^1 of PAMAP2, performed at the fist step of scenario CNN^2. Every such switching is associated with 3 bars, showing the switching delay Δ (in milliseconds), when switching is performed: 1) using the naive switching mechanism [2]; 2) using the SBRS-TP for an SBRS MoC with $A = \{I, O, par\}$; 3) using the SBRS-TP for an SBRS MoC with $A = \{I, O\}$. The higher the corresponding bar is (i.e., the larger response delay Δ is), the less efficient the switching is. For example, switching (2, 1), shown in Figure 5.7(b), is associated with 1) a bar of height 0.8; 2) a bar of height 0.7; 3) a bar of height 0.4. The bar of height 0.8, showing delay Δ of the naive switching, is the highest among the bars. Thus, the switching between scenarios CNN^2 and CNN^1 of PAMAP2 is the least efficient, when performed using the naive switching mechanism. The difference in height of bars, corresponding to one switching, shows the relative efficiency of different switching methods expressed via these bars. For example, the switching (2, 1), shown in Figure 5.7(b), is 0.8 - 0.4 = 0.4 ms less efficient when performed using naive switching (bar of height 0.8) than when performed using SBRS-TP for an SBRS with $A = \{I, O\}$ (bar of height 0.4).

As shown in Figure 5.7: 1) the switching delay Δ is typically lower when the switching is performed using the SBRS-TP, compared to the switching performed using the naive switching mechanism. Thus, the SBRS-TP is, in general, more efficient than the naive switching mechanism; 2) When the switching

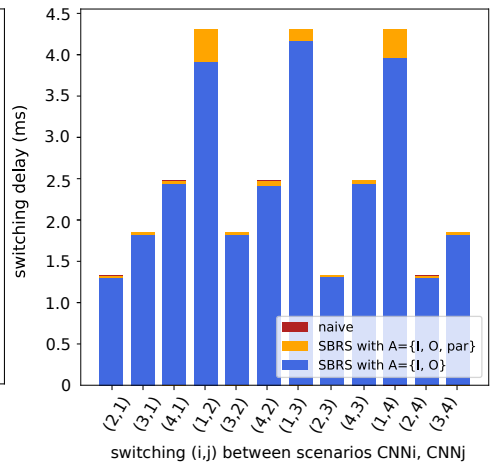
[2] One bar is sufficient to show the delay of the naive switching for SBRS MoCs with $A = \{I, O\}$ and $A = \{I, O, par\}$, respectively, because, as explained in Section 5.9, the naive switching is not affected by the application components reuse, determined by the set A



(a) *Pascal VOC*



(b) *PAMAP2*



(c) *CIFAR-10*

Figure 5.7: *SBRS-TP efficiency evaluation*

is performed under the SBRS-TP, the switching delay Δ is typically lower for an SBRS MoC with $A = \{I, O\}$ than for a functionally equivalent SBRS MoC with $A = \{I, O, par\}$. The difference occurs because among these SBRS MoCs, the one with $A = \{I, O, par\}$ typically reuses more CNN components than the one with $A = \{I, O\}$ (see Section 5.7). As explained in Section 5.9, reuse of the application components can cause an increase in switching delays, when the switching is performed under the SBRS-TP. Thus, the switching performed under the SBRS-TP is more efficient when performed in an SBRS MoC with $A = \{I, O\}$ than in a functionally equivalent SBRS MoC with $A = \{I, O, par\}$. Analogously, the relative efficiency of the SBRS-TP compared to the naive

switching is lower for Pascal VOC than for PAMAP2 or CIFAR-10 because, as explained in Section 5.10.2, Pascal VOC exploits more components reuse than PAMAP2 or CIFAR-10.

5.10.4 Comparative study

In this section, we compare our SBRS methodology to the most relevant related work called the MSDNet adaptive CNN methodology [39]. As explained in Section 5.10, the MSDNet methodology and our SBRS methodology can be compared via: 1) the run-time adaptive trade-offs between the application accuracy and resources utilization; 2) the memory efficiency.

In this section, we perform such a comparison for an example CNN-based application. The example application performs classification on the CIFAR-10 dataset, and is affected by the application environment at run-time. The comparison is performed in three steps. In Step 1, we construct the MSDNet CNN and the SBRS MoC for the example application. In Step 2, we compare the accuracy-throughput trade-off offered by the MSDNet methodology and our SBRS methodology, using the applications obtained at Step 1. Finally, in Step 3, we compare the memory efficiency of the MSDNet methodology and our SBRS methodology, using the applications obtained at Step 1.

The MSDNet CNN is constructed according to the design and training parameters specified for the CIFAR-10 dataset in the original MSDNet work [39]. It has six exits, characterized with different accuracy and throughput. During the application run-time, the MSDNet CNN can yield data from different exits, thereby offering various trade-offs between the application accuracy and throughput. We evaluate these trade-offs by executing the MSDNet CNN with an *anytime prediction* setting [39]. This setting allows the MSDNet CNN to switch among its subgraphs (exits), thereby adapting the MSDNet CNN to changes in the application environment. We note that in the original work [39] the switching among the MSDNet CNN exits is driven by a resource budget given in FLOPs, not by a throughput requirement. However, conceptually, it is possible to extend the MSDNet CNN with a throughput-driven adaptive mechanism. In this experiment, we emulate execution of the MSDNet CNN with such a mechanism in order to enable direct comparison of the MSDNet CNN with our SBRS MoC.

The SBRS MoC is obtained by using our methodology, presented in Section 5.5. As input, our methodology accepts a ResNet-18 [36] baseline CNN and three sets of application requirements. In the first set $r_1 = \{0.1, 0.9, 0, 0\}$, the application prioritizes high throughput over high accuracy. In the second set $r_2 = \{0.5, 0.5, 0, 0\}$, high throughput and high accuracy are equally impor-

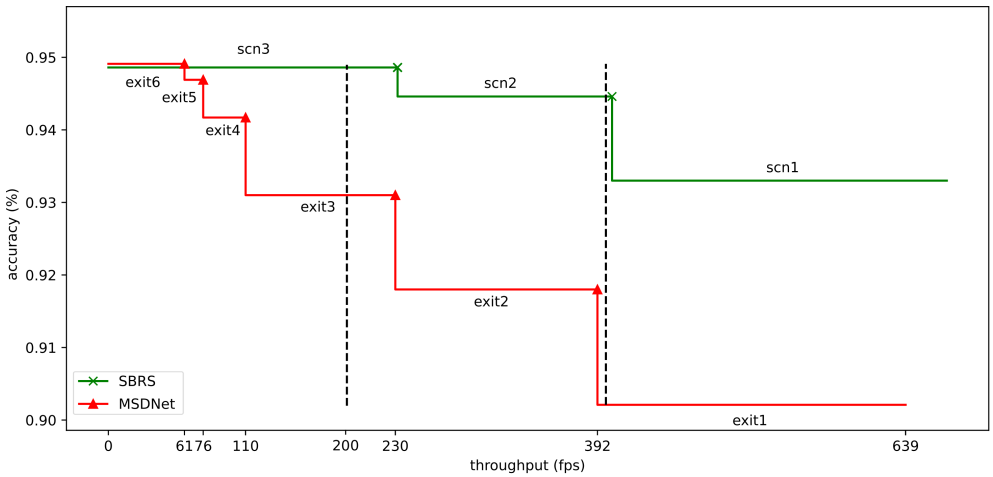


Figure 5.8: *Comparison between SBRS MoC and MSDNet CNN [39], performing classification on the CIFAR-10 dataset with throughput-driven adaptive mechanism*

tant for the application. In the third set $r_3 = \{0.9, 0.1, 0, 0\}$, the application prioritizes high accuracy over high throughput. The obtained SBRS MoC has three scenarios corresponding to the three sets of requirements r_1, r_2, and r_3. During the application run-time the SBRS MoC can switch among its scenarios, thereby offering various trade-offs between the application accuracy and throughput, and adapting the application to changes in the application environment at run-time.

The comparison, in terms of accuracy and throughput characteristics of the aforementioned MSDNet CNN and the SBRS MoC, is visualized in Figure 5.8. The horizontal axis shows the throughput (in fps). The vertical axis shows the accuracy (in %). The two step-wise curves in Figure 5.8 represent the relationships between the accuracy and the throughput, exhibited by the MSDNet CNN and SBRS MoC. Each flat segment of the step-wise curves represents a scenario in the SBRS MoC or an exit in MSDNet CNN. For example, the flat segment of the MSDNet curve, characterized with throughput between 231 and 392 fps and accuracy of 0.918%, represents exit 2 of the MSDNet CNN. Each cross marker or triangle marker represents a switching point between SBRS MoC scenarios or MSDNet CNN exits, respectively. As explained above, run-time switching among the scenarios or exits occurs when the application is affected by changes in its environment at run time. Figure 5.8 illustrates such changes in the application environment as the two vertical dashed lines, representing demands of minimum throughput, imposed on the application by the environment at run time. For example, at the start of the application execution, the environment demands that the application must

have throughput of no less than 200 fps with as high as possible accuracy. In this case, the MSDNet CNN yields data from exit 3, demonstrating 0.931% accuracy, and the SBRS MoC executes in scenario 3, demonstrating 0.949% accuracy. Later, the application environment changes and demands that the application must have throughput of no less than 394 fps. Thus, the MSDNet CNN starts to yield data from exit 1, demonstrating 0.902% accuracy, and the SBRS MoC switches to scenario 2, demonstrating 0.946% accuracy.

As shown in Figure 5.8, our SBRS MoC exhibits higher accuracy than the MSDNet CNN for any throughput requirement, except when the application has to exhibit throughput lower or equal to 61 fps. In the latter case, the accuracy of our SBRS MoC is comparable (0.05% lower) to the accuracy of the MSDNet CNN. We believe that the difference in accuracy between our SBRS MoC and the the MSDNet CNN occurs because the scenarios in the SBRS MoC are optimized for both high accuracy and high throughput, whereas the exits of MSDNet are only optimized for high CNN accuracy. Optimization for the platform-aware requirements performed during the SBRS MoC design enables for more efficient utilization of the platform resources, and therefore for more efficient execution of the application when high throughput is required.

Finally, we compare the memory efficiency between our SBRS methodology and the MSDNet methodology. To do so, we compare the memory cost of the MSDNet CNN and the SBRS MoC, designed to perform classification on the CIFAR-10 dataset. The memory cost of our final application equals 77.68 MB when the application is designed with adaptive parameters $A = \{I, O, PAR\}$, and 97.6 MB when the application is designed with adaptive parameters $A = \{I, O\}$. The memory cost of the MSDNet CNN, designed for the CIFAR-10 dataset, is equal to 103.76 MB. Thus, for the CIFAR-10 dataset, the memory efficiency of our methodology is higher than the one of MSDNet. The difference occurs because unlike the MSDNet methodology, our methodology reuses memory allocated to store intermediate computational results within every CNN as well as among different CNNs. It is fair to note that, since our methodology does not enable for reuse of CNN parameters, it may prove less efficient than MSDNet for applications that use CNNs characterized with large sizes of weights. However, such applications are not typical for execution at the Edge.

5.11 Conclusion

We have proposed a novel methodology, which provides run-time adaptation for CNN-based applications executed at the Edge to changes in the application

environment. We evaluated our proposed methodology by designing three real-world run-time adaptive applications in the domains of Human Activity Recognition (HAR) and image classification, and executing these applications on the NVIDIA Jetson TX2 edge device. The experimental results show that for real-world applications our methodology: 1) Enables to adapt a CNN-based application to changes in the application environment during run-time and therefore ensures that the application needs are served at every moment in time; 2) Achieves a high (up to 78%) degree of platform memory reuse for CNN-based applications that execute CNNs with large amounts of similar components; 3) Enables for efficient switching between the application scenarios, using the novel SBRS-TP transition protocol proposed in our methodology. Additionally, we compared our methodology to the run-time adaptive MSDNet CNN methodology, which is the most relevant to our methodology among the related work. The comparison is performed by CNNs designed for the CIFAR-10 dataset and executed on the Jetson TX2 edge device. The comparison illustrates that the application designed using our methodology outperforms the MSDNet CNN when executed under tight platform-aware requirements, and demonstrates comparable accuracy against the MSDNet CNN when the platform-aware requirements are relaxed. The difference can be attributed to the fact that unlike the MSDNet CNN, our methodology optimizes the application in terms of both high accuracy and platform-aware characteristics.

Chapter 6

Methodology for joint memory optimization of multiple CNNs

Svetlana Minakova and Todor Stefanov. "Memory-Throughput Trade-off for CNN-based Applications at the Edge". *Accepted for publication in ACM Transactions on Design Automation of Electronic Systems (TODAES), March 2022.*

IN this chapter, we present our methodology for joint memory optimization of multiple CNNs, which corresponds to the fourth research contribution of this thesis summarized in Section 1.5.4. The proposed methodology is a part of the post-selection optimization component, introduced in Section 1.5, and is an extension of our methodology for low-memory CNN inference at the Edge, presented in Chapter 4. The reminder of this chapter is organized as follows. Section 6.1 introduces, in more details, the problem addressed by our novel methodology. Section 6.2 summarizes the novel research contributions, presented in this chapter. An overview of the related work is given in Section 6.3. Section 6.4 presents a formal definition of a CNN-based application, used in this chapter. Section 6.5 presents our proposed methodology. Section 6.6 presents the experimental study performed by using the proposed methodology. Finally, Section 6.7 ends the chapter with conclusions.

6.1 Problem statement

As mentioned in Chapter 4 (see Section 4.1), the memory footprint of an application using a single CNN, let alone multiple CNNs, often has to be reduced to fit the application into the limited memory of an edge device. Typically,

the memory footprint of a CNN-based application is reduced using methodologies such as pruning and quantization [11,17,31,98], briefly introduced in Section 1.3 as a part of the CNN optimization engine. These methodologies reduce the number or/and precision of parameters (weights and biases) of a CNN, thereby reducing the memory footprint of a CNN-based application. However, at high memory reduction rates, these methodologies may decrease the CNN accuracy, while as mentioned in Section 1.2, high CNN accuracy is very important for many CNN-based applications.

To achieve high CNN memory reduction and avoid substantial decrease of the CNN accuracy, the CNN pruning and quantization methodologies can be combined with CNN memory reuse methodologies such as the methodologies in [28,47,65,76]. Orthogonal to the pruning and quantization methodologies, the CNN memory reuse methodologies reuse the platform memory allocated to store intermediate computational results, exchanged between the layers of a CNN. Thus, these methodologies further reduce the application memory cost without decreasing the CNN accuracy. However, the methodologies in [28,47,65,76] reuse platform memory within a CNN, but not among multiple CNNs, thereby missing opportunities for inter-CNN memory reuse. As a result, these methodologies are inefficient for multi-CNN applications (i.e., applications that use multiple CNNs to perform their functionality) such as the applications demonstrated in [70,84,97,104]. Moreover, due to Limitation 1, explained in Section 1.4.1, the methodologies in [28,47,65,76] do not account for non-sequential manners of CNN execution, introduced in Section 2.4. Consequently, these methodologies are also unfit for CNN-based applications that execute CNNs in a non-sequential manner, such as the applications in [65, 67,101]. To address the two issues, mentioned above, we propose our novel methodology for joint memory optimization of multiple CNNs.

6.2 Contributions

In this chapter, we propose a methodology for joint memory optimization of multiple CNNs. Our methodology offers memory reduction for CNN-based applications that use multiple CNNs or/and execute CNNs in a non-sequential manner. To this aim, our methodology significantly extends and combines two existing CNN memory reduction methodologies: the CNN buffers reuse methodology proposed in [76] and our methodology for low-memory CNN inference, presented in Chapter 4 and based on our publication [65]. Our methodology presented in Section 6.5 is the main novel contribution of this chapter. Other important novel contributions are:

- A schedule-aware CNN buffers reuse algorithm (see Section 6.5.1). This algorithm extends the CNN buffers reuse methodology proposed in [76] with consideration of various manners of CNN execution, including the most common sequential execution manner briefly introduced in Section 2.2, and alternative manners of CNN execution, explored by the system-level optimization engine, introduced in Section 1.5. Furthermore, unlike the methodology in [76], our novel CNN buffers reuse algorithm reuses memory among different CNNs as well as within a CNN. Therefore, our schedule-aware CNN buffers reuse algorithm offers memory reduction for applications that use multiple CNNs to perform their tasks or/and execute CNNs in a non-sequential manner.

- A CNN buffers size reduction algorithm (see Section 6.5.2). This algorithm combines the buffers reuse, offered by the schedule-aware CNN buffer reuse algorithm proposed in Section 6.5.1, with data processing by parts proposed in our methodology for low-memory CNN inference in Chapter 4. Additionally, our CNN buffers size reduction algorithm extends the methodology presented in Chapter 4 with memory-throughput trade-off balancing, thus avoiding unnecessarily reducing the throughput of the CNN. Therefore, our CNN buffers size reduction algorithm offers further reduction of the memory of a CNN-based application at the cost of possible CNN throughput decrease.

- up to 5.9 times memory reduction compared to deployment of CNN-based applications with no memory reduction and 7% to 30% memory reduction compared to other CNN memory reuse methodologies (see Section 6.6.1);

Additionally, in Section 6.6.2 we demonstrate that our methodology can be efficiently combined with orthogonal memory reduction methodologies such as CNN quantization.

6.3 Related Work

The most common CNN memory reduction methodologies, namely pruning and quantization, reviewed in surveys [11, 17, 31, 98], reduce the memory cost of CNN-based applications by reducing the number or size of CNN parameters (weights and biases) [4]. However, at high CNN memory reduction rates these methodologies decrease the CNN accuracy, whereas high accuracy is very important for many CNN-based applications [4]. In contrast, our memory

reduction methodology does not change the CNN model parameters and therefore does not decrease the CNN accuracy.

The knowledge distillation methodologies, reviewed in surveys [17,98], try to replace an initial CNN in a CNN-based application by an alternative CNN with the same functionality but smaller size. However, these methodologies involve CNN training from scratch and do not guarantee that the accuracy of the initial CNN can be preserved. In contrast, our memory reduction methodology is a general systematic methodology which always guarantees preservation of the CNN accuracy.

The CNN buffers reuse methodologies, such as the methodology proposed in [76], and the methodologies reviewed in [47], reduce the required CNN memory by reusing platform memory, allocated for storage of intermediate CNN computational results. These methodologies can significantly reduce the CNN memory cost without decreasing the CNN throughput or accuracy. However, these methodologies do not support reuse of the platform memory among multiple CNNs. Reusing the memory among CNNs as well as within every CNN is vital for deployment of multi-CNN applications, such as [84,86,95]. Thus, the methodologies in [47,76] are not suitable for multi-CNN applications. Moreover, these methodologies do not account for parallel execution of CNN layers. Therefore, they are not applicable to CNN-based applications, exploiting task-level (pipeline) parallelism [67,101], available within the CNNs. In contrast to these methodologies, our methodology is applicable to the CNN-based applications, exploiting pipeline parallelism, and multi-CNN applications.

The CNN buffers reduction methodology proposed in [65] and presented in Chapter 4 of this thesis allows to significantly reduce the CNN-based application memory cost at the expense of CNN throughput decrease. In this methodology, CNN layers process their input data by parts and the device memory is reused to store different parts of the layers input data. However, this methodology always tries to achieve a very low CNN memory cost at the expense of large CNN throughput decrease. In practice, partial reduction of the CNN memory cost is often sufficient to fit a CNN-based application into a device with a given memory constraint. In contrast to the methodology proposed in [65], our proposed methodology involves a balanced memory-throughput trade-off in a CNN-based application, and therefore does not involve unnecessary decrease of the CNN throughput.

The CNN layers fusion methodologies, such as the methodologies in [5,73] and the methodologies adopted by Deep Learning (DL) frameworks, such as the TensorRT DL framework [72] or the PyTorch DL framework [75], enable

to reduce the CNN memory cost by transforming the network into a simpler form but preserving the same overall behavior. Being a part of the CNN model definition, the CNN layer fusion methodologies are orthogonal to our proposed methodology and can be combined with our methodology for further CNN memory optimizations. In our experimental study (Section 6.6) we implicitly use the CNN layers fusion by implementing the CNNs with the TensorRT DL framework [72], which has built-in CNN layers fusion.

6.4 CNN-based application

A CNN-based application is an application which requires execution of one or multiple CNNs to perform its functionality. In this section, we give an example and a formal definition of a CNN-based application. Our example application *APP* is shown in Figure 6.1 and is inspired by the real-world CNN-based application for adaptive images classification proposed in [95]. For simplicity, in our application *APP* we use small made-up CNNs instead of the real-world state-of-the-art CNNs used in [95]. Also, unlike the original application in [95], our application *APP* utilizes alternative (non-sequential)

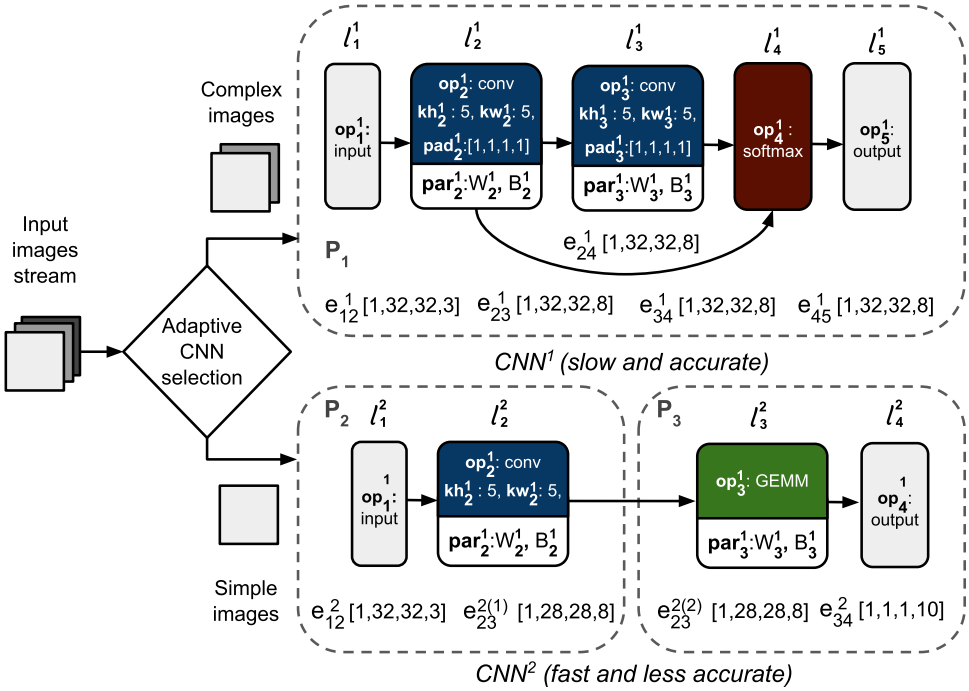


Figure 6.1: *Example CNN-based application APP*

manners of CNN execution.

To perform its functionality, application APP uses two CNNs, CNN^1 and CNN^2, designed to perform image classification on the same dataset, but characterized with different accuracy and platform-aware characteristics. CNN^1 is a large and complex CNN, characterized with high accuracy, i.e., CNN^1 performs the images classification very well. CNN^2 is a small and simple CNN. It is characterized with smaller accuracy than CNN^2, but has higher throughput, i.e., it is able to process images very fast. During its execution, application APP accepts a stream of images, also called frames, analyses these images, and adaptively selects one of its CNNs (CNN^1 or CNN^2) to perform the image classification of the input frame. The complex images are sent for processing to CNN^1, while the simple images are sent for processing to CNN^2. By using CNN^1 and CNN^2 interchangeably, application APP achieves higher classification accuracy and higher throughput, than by using only CNN^1 or only CNN^2 [95].

As mentioned in Section 2.2, when deployed on a target edge platform, a CNN-based application utilizes the platform memory and computational resources to execute the CNNs. The memory of the edge device is used to store parameters (weights and biases) and intermediate computational results of the CNNs. The intermediate computational results are typically stored in CNN buffers, briefly introduced in Section 2.2. Recall that a CNN buffer is an area of platform memory, which stores intermediate computational results (data) associated with one or multiple CNN edges and is characterized with size, specifying the maximum number of data elements, that can be stored in the buffer. To store data associated with every edge e^n_{ij} of CNN^1 and CNN^2, our example application APP uses a set of buffers B^{naive} , where every edge e^n_{ij} has its own buffer B_k of size $|e^n_{ij}.data|$. Hereinafter, we refer to such buffers allocation as naive buffers allocation. In total, application APP uses $|B^{naive}| = 9$ CNN buffers. These buffers are shown in Table 6.1, where Row 1 lists the buffers; Row 2 lists the edges using the CNN buffers to store associated data; Row 3 lists the sizes of the CNN buffers expressed in number of data elements.

The computational resources of the edge device are utilized to perform the functionality of the CNNs. Typically the CNNs are executed layer-by-layer,

Table 6.1: *Naive CNN buffers allocation*

B	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9
edges	e^1_{12}	e^1_{23}	e^1_{24}	e^1_{34}	e^1_{45}	e^2_{12}	$e^{2(1)}_{23}$	$e^{2(2)}_{23}$	e^2_{34}
size	3072	8192	8192	8192	8192	3072	6272	6272	10

i.e. at every moment in time only one CNN layer is executed on the edge platform. However, as explained in Section 2.4, a CNN-based application executed on a multi-processor platform may split CNNs into partitions (sub-networks) executed in a parallel pipelined fashion on different processors of the platform. Our example application APP shown in Figure 6.1 exploits pipeline parallelism available in CNN^2 by splitting CNN^2 into two partitions (P_2 and P_3) and executing these partitions in parallel pipelined fashion.

To enable for representation of pipeline parallelism in a CNN-based application, we: 1) represent CNNs used by the application as a set of CNN partitions P. For application APP, $P = \{P_1, P_2, P_3\}$, where P_1 is a single partition of CNN^1 (i.e., $P_1 = CNN^1$), P_2 and P_3 are partitions of CNN^2; 2) use set J, which explicitly defines the exploitation of pipeline parallelism among CNN partitions P. Every element $J_i \in J$ contains one or several CNN partitions. If two CNN partitions P_m and $P_x, m \neq x$ belong to the element $J_i \in J$, the CNN-based application exploits task-level (pipeline) parallelism among these partitions. For application APP, set $J = \{\{P_2, P_3\}\}$ specifies that partitions P_2 and P_3 of the application are executed in parallel pipelined fashion.

The execution order of CNN layers within every CNN partition $P_i, i \in [1, |P|]$ used by a CNN-based application is specified using sequence $schedule_i$ of computational steps. At every step, represented as an element of $schedule_i$, a layer of partition P_i is executed. For example, $schedule_1 = \{\{l_1^1\}, \{l_2^1\}, \{l_3^1\}, \{l_4^1\}, \{l_5^1\}\}$ specifies that the layers within partition P_1 of application APP are executed in 5 steps, and at j-th step, $j \in [1, 5]$, layer l_j^1 is executed.

Based on the discussion above , we formally define a CNN-based application as a tuple $(\{CNN^1, ..., CNN^N\}, B, P, J, \{schedule_1, ..., schedule_{|P|}\})$, where $\{CNN^1, ..., CNN^N\}$ are the CNNs utilized by the application; B is the set of CNN buffers, utilized by the application; P is the set of CNN partitions; J is the set which explicitly defines exploitation of task-level (pipeline) parallelism by the application; $schedule_i, i \in [1, |P|]$ is a schedule of partition P_i which determines the execution order of the layers within partition P_i. The example application shown in Figure 6.1 and explained above is formally defined as a tuple $APP = (\{CNN^1, CNN^2\}, B^{naive}, \{P_1, P_2, P_3\}, \{\{P_2, P_3\}\}, \{\{\{l_1^1\}, \{l_2^1\}, \{l_3^1\}, \{l_4^1\}, \{l_5^1\}\}, \{\{l_1^2\}, \{l_2^2\}\}, \{\{l_3^2\}, \{l_4^2\}\}\})$, where buffers $B^{naive} = \{B_1, ..., B_9\}$ are given in Table 6.1.

6.5 Methodology

In this section, we present our methodology for joint memory optimization of multiple CNNs. The design flow of our methodology is shown in Figure 6.2.

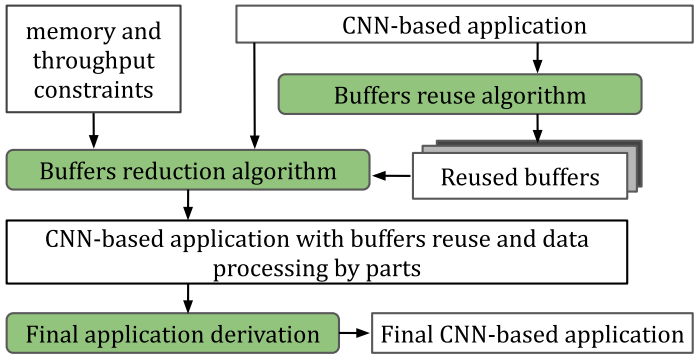


Figure 6.2: *Our methodology design flow*

Our methodology accepts as inputs a CNN-based application, as formally defined in Section 6.4, a memory constraint (in Megabytes) and an optional throughput constraint (in frames per second) posed on the CNN-based application. As an output, our methodology produces a final CNN-based application that is functionally equivalent to the input CNN-based application, but characterized with reduced memory cost and possibly decreased throughput. Our methodology consists of three main steps.

At Step 1, we introduce CNN buffer reuse into the CNN-based application, thereby reducing the application memory cost. This step is performed automatically using our buffers reuse algorithm proposed in Section 6.5.1. As an output, this step provides a set of CNN buffers to be reused among the CNNs and within the CNNs of the CNN-based application.

If the memory reduction introduced by Step 1 is insufficient to fit a CNN-based application within the given memory constraint, at Step 2, we try to further reduce the the memory cost of the CNN-based application at the expense of application throughput decrease. To do so, we introduce data processing by parts (as proposed in Chapter 4) and the buffers reuse (as proposed in Section 6.5.1) to the CNN-based application. We note that unlike the methodology in [65], where the data processing by parts has been originally proposed, Step 2 of our methodology does not introduce data processing by parts into every layer of every CNN used by the application. Instead, Step 2 searches for a subset of layers such that data processing by parts in these layers combined with buffers reuse introduces a balanced memory-throughput trade-off to the CNN-based application. This step is performed automatically using our buffers reduction algorithm proposed in Section 6.5.2. As explained in Section 4.4 in Chapter 4 , the introduction of data processing by parts in a CNN requires the layers of the CNN to be executed in a specific order. Therefore, our buffers reduction algorithm also finds and enforces in the CNNs used by

the application a specific schedule, which explicitly specifies the execution order of layers and phases in the CNNs. As an output, Step 2 provides a CNN-based application with buffers reuse and data processing by parts.

At Step 3, we use the CNN-based application, obtained at Step 2, to derive the final CNN-based application provided as the output by our methodology. This step is described in Section 6.5.3.

6.5.1 Buffers Reuse Algorithm

In this section, we present our buffers reuse algorithm, Algorithm 8, which is a greedy algorithm. It visits, one-by-one, every edge in every CNN of a CNN-based application and allocates a CNN buffer to this edge. When possible, Algorithm 8 reuses CNN buffers among the visited edges, thereby introducing memory reuse into the CNN-based application and reducing the application memory cost. Algorithm 8 accepts as an input a CNN-based application with naive buffers allocation, explained in Section 6.4. As an output Algorithm 8 produces a set of buffers B, reused among all the CNNs of the CNN-based application. An example of buffers B generated by Algorithm 8 for the example CNN-based application APP, explained in Section 6.4, is given in Table 6.2.

Unlike the naive CNN buffers allocation given in Table 6.1, the buffers in Table 6.2 are reused among CNNs and within the CNNs of application APP. For example, as shown in Column 2 in Table 6.2, CNN buffer B_1, generated by Algorithm 8, is reused among edges e^1_{12} and e^1_{34} of CNN^1 and edge e^2_{12} of CNN^2. We note that according to Equation 2.7, explained in Section 2.2, the reused buffers B, produced by Algorithm 8, occupy 24586* *token_size* bytes of memory, while the initial, non-reuse buffers, given in Table 6.1 in Section 6.4, occupy 51446* *token_size* bytes of memory.

In Line 1, Algorithm 8 sets the CNN buffers B to an empty set. In Lines 4 to 35, Algorithm 8 visits every edge e^n_{ij} of every partition $P_m \in P$ of the CNN-based application. In Line 4, Algorithm 8 creates an empty list B_{reuse} of existing CNN buffers that can be assigned to edge e^n_{ij}. In Lines 5 to 18, Algorithm 8 checks every buffer $B_k \in B$, and determines if buffer B_k can be assigned to edge e^n_{ij}. Buffer B_k cannot be assigned to edge e^n_{ij} if it is already assigned to another edge e^r_{zq} used by the CNN-based application simultane-

Table 6.2: *Reused CNN buffers*

B	B_1	B_2	B_3	B_4
edges	$e^1_{12}, e^1_{34}, e^2_{12}$	$e^1_{23}, e^1_{45}, e^{2(1)}_{23}$	$e^1_{24}, e^{2(2)}_{23}$	e^2_{34}
size	8192	8192	8192	10

Algorithm 8: Buffers reuse

Input: $APP^{in} = (\{CNN^1, ..., CNN^N, B^{naive}, P, J, \{schedule_1, ..., schedule_{|P|}\}\})$
Result: B

$B \leftarrow \emptyset$;
for $P_m \in P$ **do**
  for $e^n_{ij} \in P_m.E$ **do**
    $B_{reuse} \leftarrow \emptyset$;
    for $B_k \in B$ **do**
      $suits = true$;
      for $e^r_{zq} \in B_k.edges$ **do**
        find $P_x : e^r_{zq} \in P_x$;
        if $m \neq x$ **then**
          if $\exists J_r \in J : \{P_m, P_x\} \in J_r$ **then**
            $suits = false$;
        else
          $start_z \leftarrow$ find in $schedule_m$ first step of l^r_z;
          $end_q \leftarrow$ find in $schedule_m$ last step of l^r_q;
          $start_i \leftarrow$ find in $schedule_m$ first step of l^n_i;
          $end_j \leftarrow$ find in $schedule_m$ last step of l^n_j;
          if $[start_i, end_j] \cap [start_z, end_q] \neq \emptyset$ **then**
            $suits = false$;
      if $suits = true$ **then**
        $B_{reuse} \leftarrow B_{reuse} + B_k$;
    if $B_{reuse} = \emptyset$ **then**
      $edges \leftarrow \emptyset$; $edges \leftarrow edges + e^n_{ij}$;
      find B_z in B^{naive} such that $e^n_{ij} \in B_z.edges$;
      B_{best} = new shared buffer $(edges, B_z.size)$;
      $B \leftarrow B + B_{best}$;
    else
      $cost_{min} = inf$;
      for $B_k \in B_{reuse}$ **do**
        find B_z in B^{naive} such that $e^n_{ij} \in B_z.edges$;
        $cost = max(B_z.size - B_k.size, 0)$;
        if $cost < cost_{min}$ **then**
          $B_{best} = B_k$;
          $cost_{min} = cost$;
      $B_{best}.edges \leftarrow B_{best}.edges + e^n_{ij}$;
      $B_{best}.size = B_{best}.size + cost_{min}$;
return B

ously with edge e^n_{ij}, i.e., if: 1) edges e^r_{zq} and e^n_{ij} belong to different partitions

and the CNN-based application exploits parallelism between these partitions (conditions in Line 9 and Line 10 are met). For example, buffer B_1 of application APP, assigned to edge e^2_{12} of partition P_2 cannot be also assigned to edge e^2_{34} of partition P_3 because the application APP exploits pipeline parallelism between partitions P_2 and P_3; 2) edges e^r_{zq} and e^n_{ij}, belong to one and the same partition (condition in Line 9 is not met) and simultaneously use the platform memory. To determine whether edges e^r_{zq} and e^n_{ij} use the platform memory simultaneously, in Lines 13 to 16 Algorithm 8 takes the schedule of partition P_m, i.e, $schedule_m$, and finds in this schedule intervals (in steps) when the platform memory is used by edges e^r_{zq} and e^n_{ij}. Edge e^r_{zq} starts to use the platform memory when layer l^r_z is first executed, i.e., when layer l^r_z first writes data associated with edge e^r_{zq} to the platform memory. Edge e^r_{zq} stops using the platform memory when layer l^r_q is last executed, i.e., when layer l^r_q reads the (last part of) data associated with edge e^r_{zq} from the platform memory. Analogously, edge e^n_{ij} starts to use the platform memory when layer l^n_i is first executed and stops using the platform memory when layer l^n_j is last executed. Thus, edges e^r_{zq} and e^n_{ij} use the platform memory simultaneously if the steps interval of memory usage of e^r_{zq} overlaps with the interval of e^n_{ij}, i.e., if the condition in Line 17 is met. For example, buffer B_2 of the example application APP, assigned to edge e^1_{23} of partition P_1 cannot be also assigned to edge e^1_{24} of partition P_1. The layers within partition P_1 are executed according to $schedule_1 = \{\{l^1_1\}, \{l^1_2\}, \{l^1_3\}, \{l^1_4\}, \{l^1_5\}\}$, explained in Section 6.4. According to $schedule_1$, edge e^1_{23} uses the platform memory in steps interval [2,3], and edge e^1_{24} uses the platform memory in steps interval [2,4]. Intervals [2,3] and [2,4] overlap, which means that edges e^1_{23} and e^1_{24} use the platform memory simultaneously and cannot be assigned to one buffer. If neither of conditions 1) and 2) mentioned above is met, buffer B_k can be reused for storage of data associated with edge e^n_{ij} and is added to the list B_{reuse} in Line 20.

In Lines 21 to 35 Algorithm 8 finds a reuse buffer B_{best}, which is best suited to store the data associated with edge e^n_{ij}. If list B_{reuse}, created in Lines 4 to 20, is empty (the condition in Line 21 is met), in Lines 21 to 25, Algorithm 8 defines B_{best} as a new buffer and allocates this buffer to edge e^n_{ij}. The size of buffer B_{best} is computed as the size of buffer $B_z \in B^{naive}$ allocated to edge e^n_{ij} in the naive buffers allocation.

Otherwise, in Lines 27 to 35, Algorithm 8 selects B_{best} from the list B_{reuse}. Buffer B_{best} is selected such that the increase in memory cost, computed in Line 30, and introduced by reusing of buffer B_{best} to store data associated with edge e^n_{ij} is minimal. In Lines 34 to 35, Algorithm 8 assigns buffer B_{best} to edge

e_{ij}^n and increases the size of buffer B_{best} by the memory cost $cost_{min}$, introduced into the CNN-based application by reuse of buffer B_{best} for storage of data associated with edge e_{ij}^n. Finally, in Line 36, Algorithm 8 returns the CNN buffers B.

6.5.2 Buffers Reduction Algorithm

In this section, we present our buffers sizes reduction algorithm, Algorithm 9. This algorithm introduces data processing by parts (as proposed in Chapter 4) and buffers reuse (as proposed in Section 6.5.1) to a CNN-based application. To enable a balanced memory-throughput trade-off in the application, data processing by parts is introduced only in a subset of layers used by the application. To find this subset, Algorithm 9 uses a multi-objective Genetic Algorithm (GA) [83]: a well-known heuristic approach, which basic concepts and parameters are introduced in Section 2.6.

Algorithm 9 accepts as inputs: 1) a CNN-based application with naive buffers allocation, explained in Section 6.4; 2) a list of reused buffers B obtained using Algorithm 8, presented in Section 6.5.1; 3) Constraints M^c and T^c posed on the application. The memory constraint M^c specifies the maximum amount of memory (in MegaBytes) that can be occupied by the CNN-based application. The throughput constraint T^c is defined as a set $\{T_1^c, ..., T_N^c\}$, where $T_n^c, n \in [1, N]$ specifies the minimum throughput (in fps) which has to be demonstrated by CNN^n used by the application; 4) A set GA_par of standard user-defined GA parameters, briefly introduced in Section 2.6. As outputs, Algorithm 9 provides: 1) a CNN-based application functionally equivalent to the input application but utilizing data processing by parts and buffers reuse as explained above. Compared to the input application, the output application is characterized with reduced memory cost and possibly decreased throughput. Also, due to the utilization of data processing by parts, the output application may execute CNN layers in a different order than the input application; 2) a set of phases Φ which specifies the number of phases in every layer of every CNN used by the application. These two outputs are required to generate the final application as proposed in Section 6.5.3.

As an example, taking CNN-based application $APP = (\{CNN^1, CNN^2\}, B^{naive}, \{P_1, P_2, P_3\}, \{\{P_2, P_3\}\}, \{\{\{l_1^1\}, \{l_2^1\}, \{l_3^1\}, \{l_4^1\}, \{l_5^1\}\}, \{\{l_1^2\}, \{l_2^2\}\}, \{\{l_3^2\}, \{l_4^2\}\}\})$ introduced in Section 6.4, reused buffers B shown in Table 6.2, constraints $M^c = 0.02$ MegaBytes (20000 bytes), $T^c = \{0, 0\}$, and standard GA parameters GA_par proposed in work [83], Algorithm 9 produces as an output application $APP' = (\{CNN^1, CNN^2\}, B^{reduced}, \{P_1, P_2, P_3\}, \{\{P_2, P_3\}\}, \{\{l_1^1\}, \{l_2^1\}, [\{l_3^1\}, \{l_4^1\}, \{l_5^1\}]\ \text{x}\ 32\}, \{\{l_1^2\}, \{l_2^2\}\}, \{\{l_3^2\}, \{l_4^2\}\}\})$ and a set

Algorithm 9: Buffers reduction

Input: APP^{in}=($\{CNN^1, ..., CNN^N\}, B^{naive}, P, J, \{schedule_1, ..., schedule_{|P|}\}$), $B, Constraints = (M^c, T^c), GA\_par$
Result: APP^{out}=($\{CNN^1, ..., CNN^N\}, B^{reduced}, P, J, \{schedule'_1, ..., schedule'_{|P|}\}$), Φ

$APP^{out} \leftarrow (\{CNN^1, ..., CNN^N\}, B, P, J, \{schedule_1, ..., schedule_{|P|}\})$;
$M =$ compute memory cost of APP^{out}, using Equation 2.5;
if $M \leq M^c$ **then**
    $\Phi \leftarrow \{(l_i^n, 1)\}, n \in [1, N], i \in [1, |L^n|]$;
    return (APP^{out}, Φ);
$X \leftarrow$ binary string of length $\sum_{n=1}^{N} |L^n|$;
$fitness = minimize(EvalMemory(APP^{in}, X),$
$-EvalThroughput(APP^{in}, X, 1), ..., -EvalThroughput(APP^{in}, X, N))$;
$pareto \leftarrow GA(X, GA\_par, fitness)$;
$S \leftarrow \emptyset$;
for $X \in pareto$ **do**
    if $M = EvalMemory(APP^{in}, X) \leq M^c \wedge T_n =$
    $EvalThroughput(APP^{in}, X, n) \geq T_n^c \in T^c, n \in [1, N]$ **then**
        $S \leftarrow S \cup X$;
if $S \neq \emptyset$ **then**
    X^{best} = select from S chromosome X with minimal memory footprint
    $M = EvalMemory(APP^{in}, X)$;
else
    X^{best} = select from *pareto* chromosome X with minimal memory footprint
    $M = EvalMemory(APP^{in}, X)$;
$(APP^{out}, \Phi) \leftarrow Algorithm\ 10(APP^{in}, X^{best})$;
return (APP^{out}, Φ);

Function $EvalMemory\ (APP^{in}, X)$**:**
    $(APP^X, \Phi) \leftarrow Algorithm\ 10(APP^{in}, X)$;
    $M =$ compute memory cost of APP^X, using Equation 2.5;
    return M;

Function $EvalThroughput\ (APP^{in}, X, n)$**:**
    $(APP^X, \Phi) \leftarrow Algorithm\ 10(APP^{in}, X)$;
    $T_n =$ evaluate throughput of CNN^n used by APP^X and executed with phases Φ;
    return T_n;

of phases $\Phi = \{1, 1, 32, 32, 32, 1, 1, 1, 1\}$. Elements of set Φ specify the number of phases performed by layers l_1^1, l_2^1, l_3^1, l_4^1, l_5^1, l_1^2, l_2^2, l_3^2, and l_4^2, respectively. Application APP' uses buffers $B^{reduced}$, produced by Algorithm 9 and shown

Table 6.3: *reduced CNN buffers*

B	B_1	B_2	B_3	B_4
edges	e^1_{12}, e^1_{34},e^2_{12}	e^1_{23}, $e^{2(1)}_{23}$	e^1_{24}, $e^{2(2)}_{23}$	e^1_{45}, e^2_{34}
size	3072	8192	8192	256

in Table 6.3. We note that according to Equation 2.7, the reduced CNN buffers produced by Algorithm 9 occupy 19712* *token_size* bytes of memory (see Table 6.3), while the CNN buffers obtained by only using buffers reuse occupy 24586* *token_size* bytes of memory (see Table 6.2). The difference occurs because, besides buffers reuse, Algorithm 9 introduces data processing by parts to layers l^1_3, l^1_4, and l^1_5 of CNN^1. To enable for buffers reduction with data processing by parts, Algorithm 9 enforces a specific execution order for the layers of CNN^1 which processes data by parts. This is expressed in APP' through $schedule'_1 = \{\{l^1_1\}, \{l^1_2\}, [\{l^1_3\}, \{l^1_4\}, \{l^1_5\}]\text{x}32\}$. In $schedule'_1$, the square brackets enclose a repetitive sub-sequence of steps. At every step, a phase of a CNN layer is executed. During the first 2 steps, layers l^1_1 and l^1_2, respectively, execute their single phase. Then, phases 1-32 of layers l^1_3, l^1_4, and l^1_5 are executed in an alternating manner, where a phase of layer l^1_3 is followed by a phase of layer l^1_4, and a phase of layer l^1_5. The set Φ specifies that each of layers l^1_3, l^1_4, and l^1_5 in CNN^1 performs 32 phases (processes its input data by 32 parts), while layers l^1_1, l^1_2 of CNN^1 and all layers of CNN^2 perform one phase (do not process data by parts).

In Lines 1 to 3, Algorithm 9 checks if utilization of only buffers reuse is sufficient to meet the memory constraint. To perform the check, in Line 1, Algorithm 9 generates an application that employs only buffers reuse (uses buffers B, obtained using Algorithm 8). In Lines 2 and 3, Algorithm 9 checks whether this application meets the memory constraint. If so (the condition in Line 3 is met), in Line 5, Algorithm 9 performs an early exit. It returns as an output the application, generated in Line 1. It also returns the set of phases Φ generated in Line 4 specifying that every layer in every CNN in the application performs one phase, i.e., does not process data by parts.

Otherwise, Algorithm 9 performs a GA-based search to find a set of layers that have to process data by parts. To this end, Algorithm 9 uses a standard GA with two-parent crossover and a single-gene mutation as presented in Section 2.6, and two problem-specific GA attributes: a chromosome and a fitness function. Recall that the chromosome is a genetic representation of a GA solution. In Algorithm 9, a chromosome X specifies data processing by parts in a CNN-based application. It is defined in Line 6 as a string of length $\sum_{n=1}^{N} |L^n|$, where N is number of CNNs used by the application, $|L^n|$

Table 6.4: *Chromosome*

l_1^1	l_2^1	l_3^1	l_4^1	l_5^1	l_1^2	l_2^2	l_3^2	l_4^2
0	0	1	1	1	0	0	0	0

is the total number of layers in the n-th CNN used by the application. Every gene of the chromosome takes value 0 or 1 and specifies whether a layer processes data by parts (gene=1) or not (gene=0). Table 6.4 gives an example of a chromosome, which specifies data processing by parts as in the example application APP', mentioned above.

The fitness-function, briefly introduced in Section 2.6, evaluates the quality of GA solutions, represented as chromosomes, and guides the GA-based search. The fitness function used by Algorithm 9 is defined in Line 7. It specifies that during the GA-based search Algorithm 9 tries to: 1) minimize the application memory cost M; 2) maximize (minimize the negative) throughput T_n of every CNN used by the application. To evaluate a chromosome in terms of memory and throughput, Algorithm 9 uses function *EvalMemory* and function *EvalThroughput*, explained in the *Memory and throughput evaluation* section below.

In Line 8, Algorithm 9 performs the GA-based search, which delivers a set of pareto-optimal solutions (chromosomes) called a pareto-front [83]. From this pareto-front, in Lines 9 to 16, Algorithm 9 selects the best chromosome, i.e., a chromosome which ensures that the CNN-based application has minimum memory footprint, while, if possible, meets the memory and throughput constraints posed on the application. In Lines 9 to 12, Algorithm 9 defines subset S of the pareto-front. All chromosomes in subset S enable the CNN-based application to meet the memory and throughput constraints. If such a subset exists (the condition in Line 13 is met), in Line 14, Algorithm 9 selects the best chromosome from this subset. Otherwise, in Line 16, Algorithm 9 selects the best chromosome from the pareto-front.

In Line 17, Algorithm 9 uses the input application APP^{in} and the best chromosome X^{best} selected in Lines 9 to 16, to generate the output application APP^{out} and a set of phases Φ performed by layers of application APP^{out}. The output application uses both data processing by parts and buffers reuse, and is characterized with reduced memory cost and possibly decreased throughput compared to the input application. The generation of application APP^{out} and set Φ from the input application APP^{in} and the best chromosome X^{best} is performed using Algorithm 10, explained in the *Derivation of a CNN-based application with data processing by parts and buffers reuse* section below. Finally, in Line 18, Algorithm 9 returns application APP^{out} and set Φ.

Derivation of a CNN-based application with data processing by parts and buffers reuse

To generate an application, functionally equivalent to the input application APP^{in} but using the data processing by parts as specified in chromosome X and buffers reuse as proposed in Section 6.5.1, Algorithm 9 uses the derivation of a CNN-based application with data processing by parts and buffers reuse - see Algorithm 10. In Line 1, Algorithm 10 defines an empty set B^{min} of buffers with minimum size and no reuse, and an empty set of phases Φ. In Lines 2 to 7, Algorithm 10 visits every partition P_p in the input application APP^{in}. In Line 3, Algorithm 10 uses chromosome X and Equation 6.1 to compute the number of phases Φ_n^1 performed by every layer l_i^n in partition P_p. If gene $X.l_i^n$ of chromosome X specifies that layer l_i^n processes data by parts (i.e., $X.l_i^n = 1$), the number of phases Φ_i^n for this layer is determined using Algorithm 3, explained in Section 4.5.1 in Chapter 4. Otherwise, the number of phases Φ_i^n for layer l_i^n is set to 1, which means that layer l_i^n does not process data by parts.

$$\Phi_i^n(x) = \begin{cases} \textit{determine using Algorithm 3} & \textit{if } x = 1 \\ 1 & \textit{otherwise} \end{cases} \tag{6.1}$$

In Line 4 to 5, Algorithm 10 obtains a set of buffers B_p^{min} for partition P_p,

Algorithm 10: Derivation of a CNN-based application with data processing by parts and buffers reuse

Input: APP^{in}=$(\{CNN^1, ..., CNN^N\}, B^{naive}, P, J, \{schedule_1, ..., schedule_{|P|}\}), X$
Result: $APP^{reduced}, \Phi$

1 $B^{min} \leftarrow \emptyset$; $\Phi \leftarrow \emptyset$;
2 **for** $P_p \in APP^{in}$ **do**
3 $\quad \Phi_p \leftarrow \{(l_i^n$, Equation 6.1 $(X.l_i^n))\}, l_i^n \in P_p.L$;
4 $\quad G^p(A^p, C^p) \leftarrow$ CNN-to-CSDF (P_p, Φ_p) // Algorithm 4 in Section 4.5.2;
5 $\quad B_p^{min}, schedule'_p \leftarrow$ use SDF3 [91] to derive minimum-sized buffers and a schedule that enables execution of partition P_p represented as CSDF model G^p with these buffers;
6 $\quad B^{min} \leftarrow B^{min} \cup B_p^{min}$;
7 $\quad \Phi \leftarrow \Phi \cup \Phi_p$;
8 $APP^{parts} \leftarrow (\{CNN^1, ..., CNN^N\}, B^{min}, P, J, \{schedule'_1, ..., schedule'_{|P|}\})$;
9 $B^{reduced} \leftarrow$ Algorithm 8 (APP^{parts});
10 $APP^{reduced} = (\{CNN^1, ..., CNN^N\}, B^{reduced}, P, J, \{schedule'_1, ..., schedule'_{|P|}\})$
11 **return** $(APP^{reduced}, \Phi)$;

where every buffer $B_k \in B_p^{min}$ is allocated to an edge in partition P_p, and is characterized with minimum size. Together with buffers B_p^{min}, Algorithm 9 obtains specific schedule $schedule'_p$, which enables to correctly execute partition P_p with buffers B_p^{min}. To do so, Algorithm 10 converts every CNN partition into a functionally equivalent CSDF model (Line 4) using the CNN-to-CSDF conversion procedure - see Algorithm 4 in Section 4.5.2, and feeds the obtained CSDF models to the SDF3 embedded systems design and analysis tool [91]. In Lines 6 and 7, Algorithm 10 accumulates the minimum sized buffers and phases obtained in Lines 3 to 5 in sets B^{min} and Φ, respectively. In Line 8, Algorithm 10 generates application APP^{parts} which processes data by parts as specified in chromosome X without buffers reuse. In Lines 9 to 10, Algorithm 10 introduces buffers reuse into application APP^{parts}, thereby obtaining application $APP^{reduced}$. Finally, in Line 11, Algorithm 10 returns application $APP^{reduced}$ together with phases Φ.

Memory and throughput evaluation

The memory and throughput of a GA solution, i.e., a chromosome, are evaluated using function *EvalMemory* defined in Lines 19 to 22 of Algorithm 9 and function *EvalThroughput* defined in Lines 23 to 24 of Algorithm 9. Both functions accept as inputs the CNN-based application APP^{in} and chromosome X. From the application APP^{in} and chromosome X, functions *EvalMemory* and *EvalThroughput* generate application APP^X as explained in the *Derivation of a CNN-based application with data processing by parts and buffers reuse* section above. Function *EvalMemory* computes the memory cost of application APP^X using Equation 2.5. Function *EvalThroughput* evaluates the throughput of CNN^n used by application APP^X. The throughout of CNN^n is estimated using measurements on the platform or a third-party throughput evaluation tool.

6.5.3 Final application derivation

In this section, we show how we perform the last step of our methodology, where we derive the final CNN-based application with reduced memory cost and possibly decreased throughput from the CNN-based application with data processing by parts and buffers reuse obtained using Algorithm 9, explained in Section 6.5.2. To derive the final CNN-based application, we use a DL framework, such as TensorRT [72], and custom extensions. The DL framework is used to implement and execute the CNNs and the CNN buffers within the application. The custom extensions are used to enable alternative (different

from layer-by-layer) execution order within every CNN partition and among CNN partitions. The alternative execution order is required for processing data by parts and exploiting pipeline parallelism in the CNN-based application.

6.6 Experimental Results

In this section, we evaluate the efficiency of our methodology. The experiments are performed in two steps. First, in Section 6.6.1, we compare our proposed methodology to the existing memory reuse methodologies proposed in [76] and [65]. Then, in Section 6.6.2, we further study the impact of our proposed methodology on real-world applications and demonstrate how our methodology can be used jointly with orthogonal memory reduction methodologies such as CNN quantization. The applications considered in our experiments belong to three categories: 1) applications utilizing one CNN which is executed in a commonly adopted sequential fashion (layer-by-layer); 2) applications utilizing one CNN and exploiting pipeline parallelism available among layers of the CNN as explained in Section 2.4; 3) multi-CNN applications. By performing the experiments on the applications from these common categories, we study the efficiency of our methodology for a wide range of CNN-based applications.

6.6.1 Comparison to existing memory reuse methodologies

In this section, we evaluate the efficiency of our methodology in comparison with the existing memory reuse methodologies proposed in [76] and [65]. The comparison between our methodology and the methodologies in [76] and [65] in terms of memory reduction principles is summarized in Table 6.5.

To evaluate the efficiency of our methodology and study the impact of the memory reuse principles and features summarized in Table 6.5 on CNN-based applications, we apply our methodology and the methodologies in [76] and [65] to six real-world CNN-based applications from the three common categories, introduced in Section 6.6. The applications are listed in Column 1 in Table 6.6. To perform their functionality, the CNN-based applications utilize the state-of-the-art CNNs listed in Column 2.

We measure and compare the applications memory cost, when it is: 1) reduced using our methodology; 2) not reduced, i.e. every CNN edge has its own CNN buffer allocated, similar to the example CNN-based application, explained in Section 6.4; 3) reduced using the methodology in [76]; 4) reduced using the methodology in [65].

Table 6.5: *Comparison of the memory reduction principles and features associated with the memory reuse methodologies in [76], [65], and our proposed methodology*

memory reuse principle or feature	[65]	[76]	our methodology
buffers reuse, i.e. reuse of platform memory, allocated to store output data of different CNN layers	no	yes	yes
data processing by parts, i.e. reuse of platform memory, allocated to store partitions of input data of CNN layers	yes	no	yes
pipeline parallelism awareness	no	no	yes
reuse of platform memory among multiple CNNs	no	no	yes
memory-throughput trade-off	yes, unbalanced	no	yes, balanced

Taking into account that both the related work in [65] and our methodology can decrease the throughput of CNNs, we also measure and compare the throughput of every CNN utilized by the CNN-based applications. To measure the applications memory cost and the CNNs throughput, we execute the CNNs on the NVIDIA Jetson TX2 embedded platform [71]. Every CNN is implemented using the Tensorrt DL framework [72], the best-known and state-of-the-art for CNNs execution on the Jetson TX2, and is executed with batch size = 1, typical for CNNs execution at the Edge and native floating-point 32 data precision.

The results of our experiments are given in Columns 3 to 11 of Table 6.6, where Column 3 lists memory constraints (in MegaBytes) posed on the CNN-based applications; Columns 4 to 7 show the applications memory cost; Columns 8 to 11 show the throughput (in frames per second) of the CNNs utilized by the applications.

As shown in Columns 4 to 7, when compared to the applications deployed without memory reduction, our methodology demonstrates 2.3 to 5.9 times memory reduction, with the minimum of $(380/162) \approx 2.3$ times memory reduction achieved for application 5 and the maximum of $(161.33/27.30) \approx 5.9$ times memory reduction achieved for application 2. Analogously, when compared to the most relevant related work (the methodologies in [76] and [65]), our methodology achieves 7% to 30% memory reduction with minimum and maximum memory reduction achieved for application 5 and application 2, respectively. As shown in Columns 4 to 7, for every CNN-based application our methodology enables for more memory reduction than the methodologies in [76] and [65]. For example, the memory cost of application 1 can be

Table 6.6: *Experimental Results*

Application			Memory (MB)				Throughput (fps)			
No	CNN(s)	Memory constraint (MB)	no reduction	[76]	[65]	ours	no reduction	[76]	[65]	ours
CNN-based applications with one CNN and no exploitation of task-level (pipeline) parallelism										
1	MobileNet V2 1.0	25 15 min	58.63	20.32	16.2	20.32 14.98 14.90	46	46	40	46 41 40.5
2	EfficientNet B0	150 40 min	161.33	39.14	42.97	39.14 39.14 27.30	168.35	168.35	98	168.35 168.35 128.5
CNN-based applications, exploiting pipeline parallelism, as proposed in [67]										
3	MobileNet V2 1.0	30 15 min	61.69	20.32	17.38	30 15.92 15.92	49	46	43	49 43.65 43.65
4	EfficientNet B0	150 50 min	163.65	39.14	44.18	45 45 31.34	170.3	168.35	98.8	170.3 170.3 124.24
Multi-CNN applications										
5	Inception V2 Mobilenet V1 0.25 ResNet V1 50	200	380	175	226	175	94 432 55	94 432 55	67 183 46	94 432 55
	Inception V2 Mobilenet V1 0.25 ResNet 50	min				162	94 432 55	94 432 55	67 183 46	75 244 47
6	DenseNet121 Mobilenet V1 1.0 Resnet v1 50	500	625	291	184	161	52 59 55	52 59 55	37 50 46	52 59 55
	DenseNet121 Mobilenet V1 1.0 Resnet v1 50	min				155	52 59 55	52 59 55	37 50 46	41 54 49

reduced to 14.90 MB by our methodology and to 20.32 MB and 16.2 MB by the methodologies in [76] and [65], respectively. The difference occurs because our methodology combines the strength of both methodologies and extends the memory reuse among multiple CNNs.

Columns 8, 10 and 11 show that the reduction of the applications memory cost by the methodology in [65] and our methodology may decrease the throughput of CNNs utilized by a CNN-based application. For example, as shown in Row 4, the throughput of Mobilenet V2 CNN is: 1) decreased to 40 fps by the methodology in [65]; 2) may be decreased to 41 or 40.5 fps by our methodology. However, our methodology: 1) does not decrease the CNN throughput when the memory constraint is 25 MB; 2) decreases the CNN throughput by $46 - 41 = 5$ fps when the memory constraint is 15 MB; 3) decreases the CNN throughput by $46 - 40.5 = 5.5$ fps when the memory constraint is 0, whereas the methodology in [65] always decreases the throughput of Mobilenet V2 CNN by $46 - 40 = 6$ fps. The difference occurs because, unlike the methodology in [65], our methodology searches for an optimal (balanced) memory-throughput trade-off (see Algorithm 9).

Columns 8 to 9 show that the methodology in [76] does not introduce throughput decrease into the CNN-based applications exploiting no task-level parallelism and multi-CNN applications. However, [76] can decrease the throughput of CNNs in the CNN-based applications that exploit pipeline parallelism. For example, it decreases the throughput of EfficientNet B0 CNN, shown in Row 8. The throughput decrease occurs because the methodology in [76] reuses CNN buffers which may be simultaneously accessed by different partitions of a CNN-based application, and thus prevents exploitation of pipeline parallelism in the CNN-based application. Unlike the methodology in [76], our proposed methodology does not reuse such buffers and thus enables for exploitation of pipeline parallelism.

Columns 4 to 7, Rows 10 to 13 show that for multi-CNN applications our methodology enables more memory reduction than the methodology in [76] and the methodology in [65]. For example, our methodology is able to reduce the memory of multi-CNN application 6, shown in Rows 12 to 13 in Table 6.6 to 155 MB. This is ≈ 2 times more memory reduction than offered by the methodology in [76] and $\approx 15\%$ more memory reduction than offered by the methodology in [65]. The difference occurs because: 1) our methodology combines memory reuse principles offered by the methodologies in [76] and [65]; 2) Unlike the methodologies in [76] and [65], our methodology reuses memory among different CNNs as well as within the CNNs.

As demonstrated in this section, our methodology enables for up to 5.9

times memory reduction compared to deployment of CNN-based applications without memory reduction and 7% to 30% memory reduction compared to other memory reduction methodologies that reduce the CNN memory cost without CNN accuracy decrease.

6.6.2 Joint use of quantization and our proposed methodology

In this section, we further study the impact of our proposed methodology on real-world applications and demonstrate how our methodology can be used jointly with orthogonal memory reduction methodologies such as CNN quantization. We apply the quantization methodology offered by the TensorFlow DL framework [1] together with our proposed methodology to four CNN-based applications, executed on the NVIDIA Jetson TX2 edge platform [71]. The applications are summarized in Table 6.7 and explained in details in the *Experimental setup* section below. To study the impact of joint use of our methodology and the quantization methodology, we measure and compare the accuracy, memory cost, and throughput of the CNNs used by the applications after the applications' memory cost is decreased using: 1) quantization and no memory reuse; 2) our methodology combined with quantization. The measurements are presented in the *Experimental results* section below. The comparison of the measurements along with analysis and conclusions are presented in the *Analysis and conclusions* section below.

Experimental setup

The applications that we use to study the effectiveness of our methodology when used jointly with CNN quantization, are summarized in Table 6.7. Column 1 lists the applications' names. Column 2 lists the CNNs used by the applications. All the CNNs perform image classification on the ImageNet dataset [21], composed of RGB images with 224 pixels height and width. The baseline topology and weights of every CNN are taken from the applications

Table 6.7: *Applications*

application	CNN(s)	requirements	
		T (fps)	M (MB)
Mobilenet-sequential	Mobilenet V2	75	8
Resnet-sequential	Resnet-50	75	26
Mobilenet-pipelined	Mobilenet V2	80	30
multi-CNN	Mobilenet V2	32	30
	Resnet-50	32	

Table 6.8: *Quantization in the TensorFlow DL framework [1]*

name	No (baseline)	Half	Mixed	Int
data precision	fp32	fp16	fp16	int
parameters precision	fp32	fp16	int	int

library of the TensorFlow DL framework [1], which is well-known and widely used for CNNs design and training. For execution at the Edge, the CNNs are implemented using the Tensorrt DL framework [72], which is the best-known DL framework for CNNs execution on the NVIDIA Jetson TX2 edge platform. Columns 3 and 4 specify requirements, posed on the CNNs by the applications, and passed as inputs to our proposed methodology. Column 3 specifies the minimum throughput (in frames per second) which the CNNs are expected to demonstrate during their inference on the NVIDIA Jetson TX2 platform. Column 4 specifies the maximum amount of memory (in MegaBytes) which the CNNs can occupy.

To every application listed in Table 6.7, we apply our methodology and the quantization methodology offered by the TensorFlow DL framework [1]. The quantization methodology in [1] offers several types of quantization, summarized in Table 6.8. Each type of quantization suggests a specific target precision used to store CNN parameters and weights. The available precision includes 32-bit floating-point (fp32) precision, 16-bit floating-point (fp16) precision and a 8-bit integer (int) precision. For example, the half-quantization, shown in Column 3 in Table 6.8, suggests that the CNN parameters and data are stored in fp16 precision.

Experimental results

The experimental results for the four CNN-based applications, summarized in Table 6.7, are shown in Figure 6.3. They are shown as bar plots that compare the characteristics of the CNNs used by the applications when the applications' memory cost is reduced using: 1) quantization with no memory reuse (the light-grey bars); our methodology combined with quantization (the dark-grey bars). Every plot shows a comparison for the CNNs with a certain type of quantization offered by the TensorFlow DL framework (see Table 6.8 explained in the *Experimental setup* section above), as well as for the baseline CNNs with no quantization and the original 32-bit floating-point weights and data precision.

The bar plots in Figure 6.3 are organized in a matrix. Every row corresponds to a CNN-based application. Every column corresponds to a characteristic of the CNNs used by the application: the CNN accuracy (the first column),

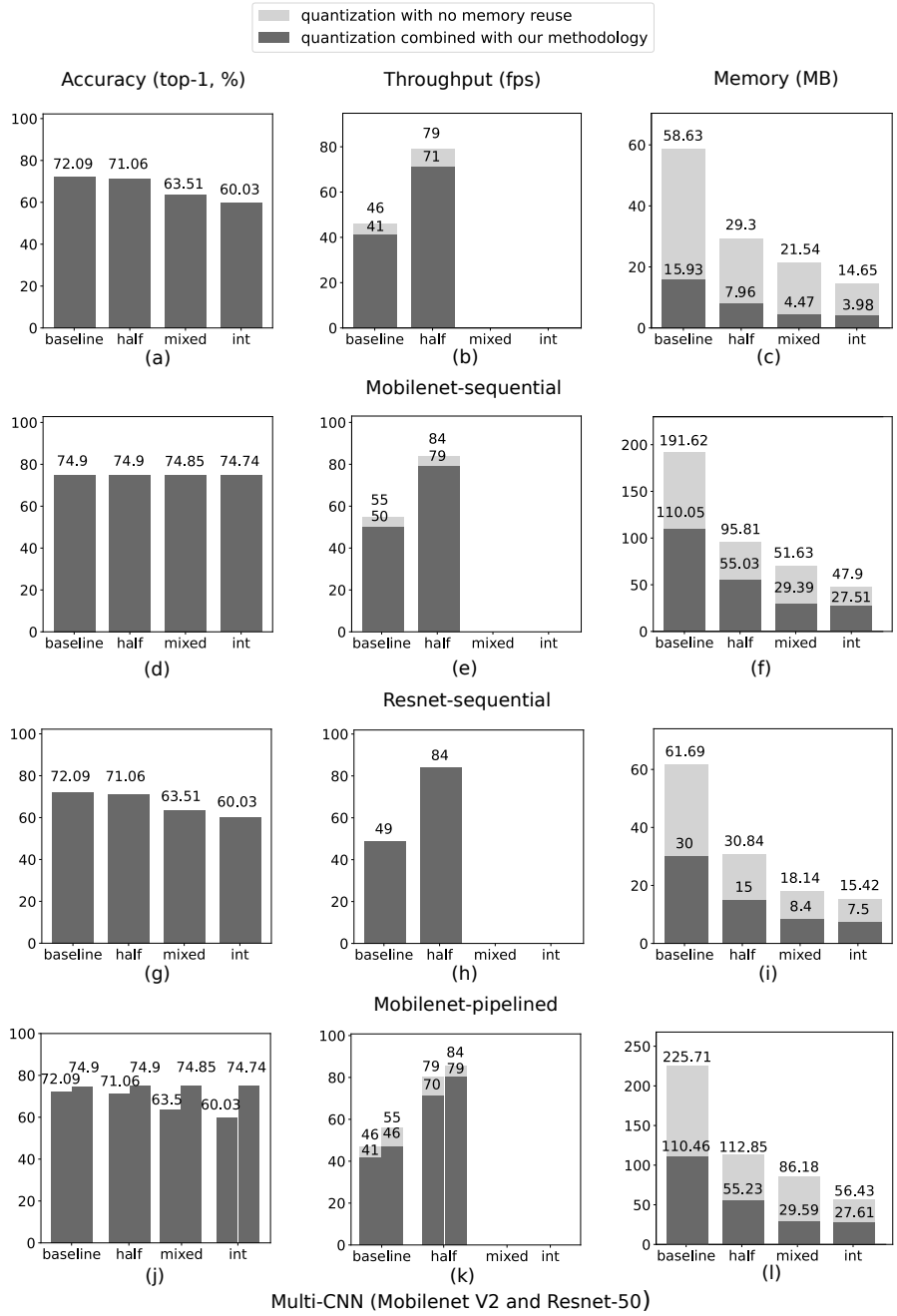


Figure 6.3: *Experimental results*

the CNN throughput (the second column)[1], and the CNN memory cost (the third column). For example, the bar plot in Figure 6.3(b), located in the first row and second column, shows the throughput of the Mobilenet V2 CNN, used by the Mobilenet-sequential application. Every bar is annotated with the value of the respective characteristic. For example, Figure 6.3(b) shows that the Mobilenet V2 CNN with half-quantization demonstrates 79 fps throughput after the quantization and no memory reuse. The difference in height between the light-grey bars and the dark-grey bars demonstrates the reduction (decrease) of the respective characteristics. For example, Figure 6.3(b) shows that our methodology decreases the throughput of the Mobilenet V2 CNN with half-quantization by $79 - 71 = 8$ fps.

Analysis and conclusions

In this section, we compare and analyse the experimental results, presented in the the *Experimental results* section above.

First, we compare the CNNs accuracy. To do that, we analyse the plots shown in the first column in Figure 6.3. We note that the accuracy of the CNNs after quantization with no memory reuse matches the CNNs accuracy after quantization combined with our methodology. In other words, our methodology does not reduce the CNNs accuracy. This is because our methodology does not change the number and precision of CNN weights.

Second, we compare the throughput of the CNNs. To do that, we analyse the plots shown in the second column in Figure 6.3. So, we see that our methodology may decrease the CNNs throughput. For example, Figure 6.3(b) shows that our methodology decreases the throughput of the Mobilenet V2 CNN with half-quantization by $79 - 71 = 8$ fps. As explained in Section 6.5, the throughput decrease occurs due to the processing data by parts, utilized by our methodology. However, the throughput decrease introduced by our methodology is small and is compensated by the throughput increase, introduced by the quantization. For example, Figure 6.3(b) shows that the throughput of the Mobilenet V2 CNN with half-quantization combined with our methodology is increased by $71 - 46 = 25$ fps, compared to the CNN with no quantization and no memory reuse (the latter CNN is represented as the light-grey 'baseline' bar).

Finally, we compare the memory cost of the CNNs. To do that, we analyse the plots shown in the third column in Figure 6.3. The plots show that our

[1]The CNN throughput is not shown for the CNNs with int- and mixed-quantization because the Jetson TX2 platform does not support integer computations.

methodology enables to further reduce the memory cost of the quantized CNNs. For example, Figure 6.3(c) shows that our methodology reduces 3.7 times the memory cost of Mobilenet V2 CNN with half-quantization. Analogously, Figure 6.3(i) shows that our methodology reduces 2.1 times the memory cost of Mobilenet V2 CNN with half-quantization and pipelined execution. This means, that our methodology can be efficiently combined with the orthogonal quantization methodology to achieve high rates of CNN memory reduction. The effectiveness of the methodologies joint use is explained by the orthogonality of the methodologies. The quantization methodology changes the precision of the CNN data and weights, thereby reducing the CNN memory cost, i.e., the amount of platform memory required to deploy and execute the CNN. Our methodology, orthogonal to the quantization, efficiently reuses the platform memory allocated for the CNN deployment, thereby further reducing the CNN memory cost.

Based on the analysis presented above, we conclude that *our methodology can be efficiently combined with the orthogonal methodologies such as quantization. The joint use of our methodology and quantization enables to achieve high rates of CNN memory reduction. Moreover, when our methodology is combined with quantization, the decrease of the CNN throughput, introduced by our methodology is easily compensated by the CNN throughput increase, introduced by the quantization.*

6.7 Conclusions

We propose a methodology for joint memory optimization of multiple CNNs. Our proposed methodology significantly extends and combines two existing memory reuse methodologies. In addition to the reuse of platform memory offered by the existing methodologies, our methodology offers support of alternative (non-sequential) manners of CNN execution, reuse of memory among different CNNs, and a memory-throughput trade-off balancing mechanism. Thus, our methodology offers efficient memory reduction for CNN-based applications that use multiple CNNs or/and execute CNNs in a non-sequential manner. The evaluation results show that our methodology: 1) enables for up to 5.9 times memory reduction compared to deployment of CNN-based applications with no memory reduction, and 7% to 30% memory reduction compared to other memory reduction methodologies that reduce the CNN memory cost without CNN accuracy decrease; 2) can be efficiently combined with orthogonal memory reduction methodologies such as quantization to achieve high rates of CNN memory reduction.

Chapter 7

Summary and concluding remarks

CONVOLUTIONAL Neural Networks (CNNs) are biologically inspired computational models that are extremely effective at processing multi-dimensional data and solving tasks such as images classification, objects detection and others [4]. Nowadays, to ensure high responsiveness, low energy cost and data privacy, many CNN-based applications execute CNNs on edge (mobile and embedded) platforms. However, while execution of CNNs at the Edge is desirable and beneficial, it is also challenging due to numerous requirements, posed on the CNNs by the application and the target edge platform. Among these requirements, the most common are high CNN accuracy, high CNN throughput, low CNN latency, low CNN memory cost, and low CNN energy cost. The aforementioned requirements make the design of a CNN executed at the Edge a complex task. Typically, this task is performed by means of the state-of-the-art (SOTA) design flow, shown in Figure 1.3. The heart of the SOTA design flow are platform-aware NAS and the CNN optimization methodologies. These methodologies explore CNNs with different architectures and parameters (weights and biases) and try to find a CNN which adheres to all the requirements posed on it. This CNN is then implemented by means of existing DL frameworks such as Keras [19], Tensorflow [1], TensorRT [72] and others [74], and deployed on an edge platform.

The SOTA design flow, however, has limitations that negatively affect the design of CNN-based applications executed at the Edge. First of all, it restricts the execution of a CNN to a so called sequential (layer-by-layer) manner. The sequential manner of CNN execution is widespread due to its simplicity but it cannot always guarantee efficient CNN execution (i.e.,

efficient utilization of resources, available on the target edge platform by a CNN). As a dataflow kind of model, characterized with large amount of parallelism available within and among its layers, a CNN can be executed in alternative (non-sequential) manners, that ensure efficient utilization of the target edge platform resources, and thus significantly improve the platform-aware characteristics of the CNN [14,101]. However, this is not explored in the SOTA design flow. Secondly, the SOTA design flow assumes that a CNN-based application only uses one CNN to perform its task. As a result, the SOTA design flow lacks means for inter-CNN optimizations and run-time adaptivity, important to some CNN-based applications. In this thesis, we try to relax the two limitations, mentioned above, and reduce their negative impact on the design of CNN-based applications executed at the Edge. To this end, we have extended the SOTA design flow as shown in Figure 1.5 and explained in Section 1.5. To implement the extended design flow shown in Figure 1.5, we have proposed four novel methodologies, presented in Chapters 4 to 6 in this thesis. Below, we summarize the proposed methodologies and give some concluding remarks.

To relax the first limitation, mentioned above, we have proposed the methodologies, presented in Chapter 3 and Chapter 4. These methodologies explore and exploit alternative (non-sequential) manners of CNN execution, thereby improving platform-aware characteristics of a CNN.

The methodology presented in Chapter 3 ensures high CNN inference throughput, required by many CNN-based applications executed at the Edge [14]. To ensure high CNN inference throughput, our methodology efficiently distributes (maps) the computations within a CNN to the computational resources of a target edge platform. The mapping, found by our methodology, features combined exploitation of two types of parallelism, namely task-level parallelism and data-level parallelism, available within a CNN. This feature distinguishes our methodology from other existing methodologies because these methodologies utilize only task-level or only data-level parallelism, when mapping a CNN onto an edge platform. Based on the experimental results, we conclude that for CNNs executed on the Jetson TX2 edge platform [71], our methodology offers a 1.36% to 42% higher inference throughput, compared to the mapping methodology employed by the best-known and state-of-the-art TensorRT DL framework for the Jetson TX2 edge platform.

The methodology presented in Chapter 4 ensures low-memory CNN inference at the Edge, required for CNN-based applications executed on edge platforms with extremely small memory resources, such as embedded Internet-of-Things (IoT) devices [17]. To ensure low CNN memory footprint, our

methodology splits the data exchanged between layers of a CNN into parts, processed in a specific order, and efficiently reuses the platform memory among the data parts. However, as the data processing by parts may cause CNN execution time overheads (e.g., CNN layers may require time to switch among the data parts), our methodology may decrease the CNN throughput. The evaluation results show that, compared to the memory reduction, achieved by the most relevant CNN buffers reuse methodology, employed by the TensorRT DL framework for efficient CNN execution at the Edge, our memory reduction methodology allows to reduce the CNN memory footprint by 2.8% to 38% at the cost of 2% to 23% decrease of the CNN throughput.

The methodologies proposed in Chapter 5 and Chapter 6 of this thesis, are aimed at relaxation of the second limitation, mentioned above.

The scenario-based run-time switching (SBRS) methodology in Chapter 5 introduces the use of multiple CNNs and run-time adaptive switching between these CNNs to a CNN-based application. Every CNN in the application corresponds to a scenario and is designed to adhere to a specific set of requirements, posed on the CNN. During the application execution, the application environment can trigger the application to switch between the scenarios, thereby adapting the characteristics of a CNN-based application to changes in the application environment (such as a change in device battery level or a change of the throughput of the input data stream). Thus, our methodology ensures efficient execution of an application which needs are affected by the application environment at run-time. To the best of our knowledge, our proposed methodology is the first methodology, able to design an adaptive CNN-based application, which considers platform-aware requirements and constraints that are specifically affected by environment changes at run-time. The most relevant to our methodology, the MSDNet methodology, does not consider real-world platform-aware requirements and constraints and does not offer means for automated switching among the application scenarios, based on the changes in the application environment. Based on the experimental results, we conclude that: 1) by introducing run-time adaptivity into CNN-based applications, affected by changes in the application environment at run-time, our methodology significantly improves the applications' characteristics; 2) our methodology outperforms the most relevant MSDNet methodology.

The methodology proposed in Chapter 6 extends our low-memory CNN inference methodology, proposed in Chapter 4, with support for pipeline parallelism and inter-CNN memory reuse. Thus, our methodology offers efficient memory reduction to a wide spectrum of applications, designed using both the SOTA design flow and our extended design flow. Based on the

evaluation results, we conclude that our methodology: 1) enables for up to 5.9 times memory reduction compared to deployment of CNN-based applications with no memory reduction, and 7% to 30% memory reduction compared to other memory reduction methodologies that reduce the CNN memory cost without CNN accuracy decrease; 2) can be efficiently combined with well-known pruning and quantization methodologies (that are orthogonal to our methodology) in order to offer high rates of CNN memory compression without significant decrease of the application accuracy and throughput.

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: Large-scale machine learning on heterogeneous systems. https://www.tensorflow.org/, 2015.

[2] M. Abadi, M. Isard, and D. G. Murray. A computational model for tensorflow: An introduction. In *1st ACM SIGPLAN International Workshop on Machine Learning and Programming (MAPL)*, MAPL 2017, page 1–7, New York, NY, USA, 2017. Association for Computing Machinery.

[3] M. S. Abdelfattah, L. 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 *Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[4] M. Z. Alom, T. M. Taha, C. Yakopcic, S. Westberg, P. Sidike, M. S. Nasrin, B. C. V. Essen, A. A. S. Awwal, and V. K. Asari. The history began from alexnet: A comprehensive survey on deep learning approaches. *CoRR*, abs/1803.01164, 2018.

[5] M. Alwani, H. Chen, M. Ferdman, and P. Milder. Fused-layer cnn accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[6] Y. Ando, S. Seiya, S. Honda, H. Tomiyama, and H. Takada. Automated identification of performance bottleneck on embedded systems

for design space exploration. *Embedded Systems: Design, Analysis and Verification*, pages 171–180, 2013.

[7] J. Bai, F. Lu, and K. Zhang. Open neural network exchange format (onnx) models zoo. https://github.com/onnx/models.

[8] A. Barbier and others at https://github.com/ARM software/ComputeLibrary/graphs/contributors. Arm compute library. https://github.com/ARM-software/ComputeLibrary.

[9] H. Benmeziane, K. E. Maghraoui, H. Ouarnoughi, S. Niar, M. Wistuba, and N. Wang. A comprehensive survey on hardware-aware neural architecture search. *CoRR*, abs/2101.09336, 2021.

[10] G. Bilsen, M. Engels, and R. Lauwereins. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, 1996.

[11] D. Blalock, J. Ortiz, J. Frankle, and J. Guttag. What is the state of neural network pruning? In *Proceedings of Machine Learning and Systems (MLSys)*, 2020.

[12] T. Bolukbasi, J. Wang, O. Dekel, and V. Saligrama. Adaptive neural networks for efficient inference. In *International Conference on Machine Learning (ICML)*, page 527–536, 2017.

[13] R. Bonna, D. S. Loubach, G. Ungureanu, and I. Sander. Modeling and simulation of dynamic applications using scenario-aware dataflow. *ACM Transactions on Design Automation of Electronic Systems*, 24(5), 2019.

[14] S. Branco, A. G. Ferreira, and J. Cabral. Machine learning in resource-scarce embedded systems, fpgas, and end-devices: A survey. *Electronics*, 8(11), 2019.

[15] J. M. Carroll. *Scenario-based design: envisioning work and technology in system development*. John Wiley and Sons Inc, 1995.

[16] A. Cheng, J. Dong, C. Hsu, S. Chang, M. Sun, S. Chang, J. Pan, Y. Chen, W. Wei, and D. Juan. Searching toward pareto-optimal device-aware neural architectures. In *International Conference On Computer-Aided Design (ICCAD)*. Association for Computing Machinery, 2018.

[17] Y. Cheng, D. Wang, P. Zhou, and T. Zhang. A survey of model compression and acceleration for deep neural networks. *IEEE Signal Processing Magazine*, 2018.

[18] A. Chinchuluun et al. *Pareto Optimality, Game Theory And Equilibria*, volume 17. Springer, 01 2008.

[19] F. Chollet. Keras. https://github.com/fchollet/keras, 2015.

[20] B. Dayma, S. Patil, P. Cuenca, K. Saifullah, T. Abraham, P. Le Khac, L. Melas, and R. Ghosh. Dall·e mini. https://github.com/borisdayma/dalle-mini, 2021.

[21] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, pages 248–255. IEEE, 2009.

[22] M. Dhouibi, A. K. B. Salem, A. Saidi, and S. B. Saoud. Accelerating deep neural networks implementation: A survey. *IET Computers and Digital Techniques*, 2021.

[23] A. Diamant, A. Chatterjee, M. Vallieres, G. Shenouda, and J. Seuntjens. Deep learning in head and neck cancer outcome prediction. *Scientific Reports*, 9:27–64, 2019.

[24] T.-D. Do, M.-T. Duong, Q.-V. Dang, and M.-H. Le. Real-time self-driving car navigation using deep neural network. In *2018 4th International Conference on Green Technology and Sustainable Development (GTSD)*, pages 7–12, 2018.

[25] T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20:1–21, 2019.

[26] S. Even. *Graph Algorithms*. Cambridge University Press, 2 edition, 2011.

[27] M. Everingham, L. van Gool, C. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results. http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html, 2012.

[28] J. Farley and A. Gerstlauer. Memory-aware fusing and tiling of neural networks for accelerated edge inference. *CoRR*, abs/2107.06960, 2021.

[29] M. Garza-Fabre, G. T. Pulido, and C. A. Coello. Ranking methods for many-objective optimization. In *MICAI 2009: Advances in Artificial Intelligence*, pages 633–645, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[30] W. Gay. *Raspberry Pi Hardware Reference*. Apress, USA, 1st edition, 2014.

[31] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer. A survey of quantization methods for efficient neural network inference. *ArXiv*, abs/2103.13630, 2021.

[32] D. Gizopoulos, G. Papadimitriou, A. Chatzidimitriou, V. J. Reddi, B. Salami, O. S. Unsal, A. C. Kestelman, and J. Leng. Modern hardware margins: Cpus, gpus, fpgas recent system-level studies. In *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 129–134, 2019.

[33] I. J. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, Cambridge MA USA, 2016.

[34] A. Gordon, E. Eban, O. Nachum, B. Chen, T. Yang, and E. Choi. Morphnet: Fast & simple resource-constrained structure learning of deep networks. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1586–1595. IEEE Computer Society, 2018.

[35] J. Hanhirova, T. Kämäräinen, S. Seppälä, M. Siekkinen, V. Hirvisalo, and A. Ylä-Jääski. Latency and throughput characterization of convolutional neural networks for mobile computer vision. MMSys '18, page 204–215, New York, NY, USA, 2018. Association for Computing Machinery.

[36] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[37] S. Heath. *Embedded Systems Design 2nd Edition*. Newnes, 2002.

[38] C. Hsu, S. Chang, D. Juan, J. Pan, Y. Chen, W. Wei, and S. Chang. MONAS: multi-objective neural architecture search using reinforcement learning. *CoRR*, abs/1806.10332, 2018.

[39] G. Huang, D. Chen, T. Li, F. Wu, L. van der Maaten, and K. Q. Weinberger. Multi-scale dense networks for resource efficient image classification. International Conference on Learning Representations (ICLR), 2018.

[40] G. Huang, Z. Liu, and K. Q. Weinberger. Densely connected convolutional networks. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[41] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

[42] L. N. Huynh, R. Balan, and Y. Lee. Deepsense: A gpu-based deep convolutional neural network framework on commodity mobile devices. In *Workshop on Wearable Systems and ApplicationsJune (WearSys'16)*, pages 25–30, 2016.

[43] L. N. Huynh, Y. Lee, and R. K. Balan. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2017.

[44] E. Jeong, J. Kim, and S. Ha. Tensorrt-based framework and optimization methodology for deep learning inference on jetson boards. *ACM Trans. Embed. Comput. Syst.*, dec 2022. Just Accepted.

[45] E. Jeong, J. Kim, S. Tan, J. Lee, and S. Ha. Deep learning inference parallelization on heterogeneous processors with tensorrt. *IEEE Embedded Systems Letters*, 14(1):15–18, 2022.

[46] W. Jiang, X. Zhang, E. H. Sha, L. Yang, Q. Zhuge, Y. Shi, and J. Hu. Accuracy vs. efficiency: Achieving both through fpga-implementation aware neural architecture search. In *Design Automation Conference (DAC)*, pages 1–6, 2019.

[47] H. Jin, B. Liu, W. Jiang, Y. Ma, X. Shi, B. He, and S. Zhao. Layer-centric memory reuse and data migration for extreme-scale deep learning on many-core architectures. *ACM Transactions on Architecture and Code Optimization*, 15(3), 2018.

[48] D. Kang, D. Kang, J. Kang, S. Yoo, and S. Ha. Joint optimization of speed, accuracy, and energy for embedded image recognition systems. In *2018 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 715–720, 2018.

[49] D. Kang, E. Kim, I. Bae, B. Egger, and S. Ha. C-good: C-code generation framework for optimized on-device deep learning. In *International Conference On Computer-Aided Design (ICCAD)*, 2018.

[50] D. Kang, J. Oh, J. Choi, Y. Yi, and S. Ha. Scheduling of deep learning applications onto heterogeneous processors in an embedded device. *IEEE Access*, 8:43980–43991, 2020.

[51] A. Krizhevsky, V. Nair, and G. Hinton. Cifar-10 (canadian institute for advanced research). http://www.cs.toronto.edu/ kriz/cifar.html, 2013.

[52] S. Kukkonen and J. Lampinen. Ranking-dominance and many-objective optimization. In *2007 IEEE Congress on Evolutionary Computation*, pages 3983–3990, 2007.

[53] C. Kyrkou, G. Plastiras, T. Theocharides, S. I. Venieris, and C.-S. Bouganis. Dronet: Efficient convolutional neural network detector for real-time uav applications. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 967–972, 2018.

[54] L. Lai, N. Suda, and V. Chandra. Not all ops are created equal! In *SysML*, 2018.

[55] M. N. U. Laskar, L. G. S. Giraldo, and O. Schwartz. Correspondence of deep neural networks and the brain for visual textures. *ArXiv*, abs/1806.02888, 2018.

[56] Y. Lecun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989.

[57] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[58] T. Lee, S. Mckeever, and J. Courtney. Flying free: A research overview of deep learning in drone navigation autonomy. *Drones*, 5(2), 2021.

[59] S. Lipschutz and M. Lipson. *Linear Algebra (Schaum's Outlines)*. McGraw Hill, 4 edition, 2009.

[60] D. Liu, H. Kong, X. Luo, W. Liu, and R. Subramaniam. Bringing AI To Edge: From deep learning's perspective. *Neurocomputing*, 485:297–320, 2022.

[61] L. Liu and J. Deng. Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution. In *AAAI*, pages 3675–3682. AAAI Press, 2018.

[62] L. Lu, Y. Zheng, G. Carneiro, and L. Yang. *Deep Learning and Convolutional Neural Networks for Medical Image Computing*. Springer, 2017.

[63] G. Martin. Overview of the mpsoc design challenge. In *Design Automation Conference (DAC)*, DAC '06, page 274–279, New York, NY, USA, 2006. Association for Computing Machinery.

[64] S. Minakova, D. Sapra, T. Stefanov, and A. D. Pimentel. Scenario based run-time switching for adaptive cnn-based applications at the edge. *ACM Transactions on Embedded Computing Systems (TECS)*, 21(2), 2022.

[65] S. Minakova and T. Stefanov. Buffer sizes reduction for memory-efficient cnn inference on mobile and embedded devices. In *Euromicro Conference on Digital System Design (DSD)*, pages 133–140. IEEE Xplore, 2020.

[66] S. Minakova and T. Stefanov. Memory-throughput trade-off for cnn-based applications at the edge. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 1(1), 2022.

[67] S. Minakova, E. Tang, and T. Stefanov. Combining task- and data-level parallelism for high-throughput cnn inference on embedded cpus-gpus mpsocs. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 18–35, Cham, 2020. Springer International Publishing.

[68] O. Moreira. *Temporal analysis and scheduling of hard real-time radios running on a multi-processor*. PhD thesis, Technical University Eindhoven, 2012.

[69] F. Moya Rueda, R. Grzeszick, G. A. Fink, S. Feldhorst, and M. Ten Hompel. Convolutional neural networks for human activity recognition using body-worn sensors. *Informatics*, 5(2), 2018.

[70] L. Nanni, S. Ghidoni, and S. Brahnam. Ensemble of convolutional neural networks for bioimage classification. *Applied Computing and Informatics*, 17, 2021.

[71] NVIDIA. Jetson embedded platform. //https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2.

[72] NVIDIA. Tensorrt framework. https://developer.nvidia.com/tensorrt, 2016.

[73] M. Olyaiy, C. Ng, and M. Lis. Accelerating dnns inference with predictive layer fusion. In *ICS*, page 291–303. Association for Computing Machinery, 2021.

[74] A. Parvat, J. Chavan, S. Kadam, S. Dev, and V. Pathak. A survey of deep-learning frameworks. In *International Conference on Inventive Systems and Control (ICISC)*, pages 1–7, 2017.

[75] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[76] Y. Pisarchyk and J. Lee. Efficient memory management for deep neural net inference. In *MLSys 2020 Workshop on Resource-Constrained Machine Learning (ReCoML 2020)*, 2020.

[77] B. Reagen, U. Gupta, R. Adolf, M. M. Mitzenmacher, A. M. Rush, G.-Y. Wei, and D. Brooks. Weightless: Lossy weight encoding for deep neural network compression. In *International Conference on Learning Representations (ICLR)*, 2018.

[78] P. Refaeilzadeh, L. Tang, and H. Liu. *Encyclopedia of Database Systems*, chapter Cross-Validation, pages 532–538. Springer US, 2009.

[79] A. Reiss. https://archive.ics.uci.edu/ml/datasets/PAMAP2 Physical Activity Monitoring, 2012.

[80] M. Richards and N. Ford. *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media, Incorporated, 2019.

[81] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4510–4520, 2018.

[82] D. Sapra and A. D. Pimentel. Constrained evolutionary piecemeal training to design convolutional neural networks. In *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*. Springer, 2020.

[83] K. Sastry, D. Goldberg, and G. Kendall. *Genetic Algorithms*, pages 97–125. Springer US, Boston, MA, 2005.

[84] B. Savelli, A. Bria, M. Molinara, C. Marrocco, and F. Tortorella. A multi-context cnn ensemble for small lesion detection. *Artificial Intelligence in Medicine*, 103:101749, 2020.

[85] L. M. Schmitt. Theory of genetic algorithms. *Theoretical Computer Science*, 259(1):1–61, 2001.

[86] M. Seeland and P. Mader. Multi-view classification with convolutional neural networks. *PLoS ONE*, 2021.

[87] A. Shvets, A. Rakhlin, A. A. Kalinin, and V. Iglovikov. Automatic instrument segmentation in robot-assisted surgery using deep learning. In *IEEE International Conference on Machine Learning and Applications (IEEE ICMLA)*, pages 624–628, 2018.

[88] A. K. Singh, A. Prakash, K. R. Basireddy, G. V. Merrett, and B. M. Al-Hashimi. Energy-efficient run-time mapping and thread partitioning of concurrent opencl applications on cpu-gpu mpsocs. *ACM Trans. Embed. Comput. Syst.*, 16(5s), 2017.

[89] H. Sofaer, J. Hoeting, and C. Jarnevich. The area under the precision-recall curve as a performance metric for rare binary events. *Methods in Ecology and Evolution*, 10, 12 2018.

[90] L. Song, J. Mao, Y. Zhuo, X. Qian, H. Li, and Y. Chen. Hypar: Towards hybrid parallelism for deep learning accelerator array. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 56–68, 2019.

[91] S. Stuijk, T. Basten, and M. Geilen. Sdf3: Sdf for free. In *Sixth International Conference on Application of Concurrency to System Design*, pages 276–278, Los Alamitos, CA, USA, jun 2006. IEEE Computer Society.

[92] M. Tan, B. Chen, R. Pang, V. Vasudevan, and Q. V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *IEEE / CVF Computer Vision and Pattern Recognition Conference (CVPR)*, 2019.

[93] E. Tang, S. Minakova, and T. Stefanov. Energy-efficient and high-throughput cnn inference on embedded cpus-gpus mpsocs. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. Springer, 2021.

[94] L. Tang, Y. Wang, T. L. Willke, and K. Li. Scheduling computation graphs of deep learning models on manycore cpus. *CoRR*, abs/1807.09667, 2018.

[95] B. Taylor, V. S. Marco, W. Wolff, Y. Elkhatib, and Z. Wang. Adaptive selection of deep learning models on embedded systems. In *ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, page 31–43. Association for Computing Machinery, 2018.

[96] I. Theodorakopoulos, V. K. Pothos, D. Kastaniotis, and N. Fragoulis. Parsimonious inference on convolutional neural networks: Learning and applying on-line kernel activation rules. *CoRR*, abs/1701.05221, 2017.

[97] J. Venugopalan, L. Tong, H. R. Hassanzadeh, and M. Wang. Multimodal deep learning models for early detection of alzheimer's disease stage. *Scientific Reports*, 11, 2021.

[98] M. P. Vestias. A survey of convolutional neural networks on edge with reconfigurable computing. *Algorithms*, 12(8), 2019.

[99] J. Vinu et al. A programmable approach to neural network compression. *IEEE Micro*, 40(5):17–25, 2020.

[100] C.-C. Wang, Y.-C. Liao, M.-C. Kao, W.-Y. Liang, and S.-H. Hung. Perfnet: Platform-aware performance modeling for deep neural networks. RACS '20, page 90–95, New York, NY, USA, 2020. Association for Computing Machinery.

[101] S. Wang, G. Ananthanarayanan, Y. Zeng, N. Goel, A. Pathania, and T. Mitra. High-throughput cnn inference on embedded arm big.little multicore processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2254–2267, 2020.

[102] Y. Wang, J. Shen, T.-K. Hu, P. Xu, T. Nguyen, R. Baraniuk, Z. Wang, and Y. Lin. Dual dynamic inference: Enabling more efficient, adaptive and controllable deep inference. *IEEE Journal of Selected Topics in Signal Processing*, 14:623–633, 2020.

[103] 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 *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10734–10742. Computer Vision Foundation / IEEE, 2019.

[104] Y. Xiao, F. Codevilla, A. Gurram, O. Urfalioglu, and A. López. Multimodal end-to-end autonomous driving. *IEEE Transactions on Intelligent Transportation Systems*, PP:1–11, 2020.

[105] Y. Xu, L. Xie, X. Zhang, X. Chen, B. Shi, Q. Tian, and H. Xiong. Latency-aware differentiable neural architecture search. *arXiv preprint arXiv:2001.06392*, 2020.

[106] T.-J. Yang, Y.-H. Chen, and V. Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6071–6079, 2017.

[107] J. Yu, L. Yang, N. Xu, J. Yang, and T. Huang. Slimmable neural networks. In *International Conference on Learning Representations (ICLR)*, 2019.

[108] J. T. Zhai, S. Niknam, and T. Stefanov. Modeling, analysis, and hard real-time scheduling of adaptive streaming applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2636–2648, 2018.

[109] Y. Zhao, W. Wang, Y. Li, C. Colman Meixner, M. Tornatore, and J. Zhang. Edge computing and networking: A survey on infrastructures and applications. *IEEE Access*, pages 1–1, 07 2019.

[110] Z. Zhao, K. M. Barijough, and A. Gerstlauer. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2348–2359, 2018.

Summary

Convolutional Neural Networks (CNNs) are biologically inspired computational models, characterized with the ability to handle large, unstructured data. Due to this ability, CNNs excel at tasks such as image classification, image segmentation, natural language processing, and are widely used to perform these tasks in applications such as navigation, facial recognition, medical images analysis, and others. Nowadays, many CNN-based applications are executed on edge platforms: mobile phones, tablets, cameras, etc. This stands in contrast to the more common practice in which CNN-based applications are executed on data centers (in the cloud). Unlike execution in the cloud, execution at the Edge does not require transmission of the collected data (e.g., images from a CCTV camera) over the Internet, and thus guarantees higher responsiveness and security.

However, execution of CNN-based applications at the Edge is challenging due to requirements posed on the CNNs by the application and the target edge platform. Among these requirements, the most common are high accuracy, high throughput, low latency, low memory cost, and low energy cost. These requirements make the design of a CNN executed at the Edge a complex task. Typically, this task is performed using the state-of-the-art (SOTA) design flow. The SOTA design flow explores CNNs with different architectures and parameters and tries to find a CNN which adheres to all the requirements posed on it.

While taking a good care of *what* is executed at the Edge (i.e., which architecture and which parameters does a CNN have), the SOTA design flow does not explore *how* a CNN or CNN-based application is executed (i.e., how it utilizes computational, memory, and energy resources available on the edge platform). Instead, the SOTA design flow adopts limitations that negatively affect the design of CNNs and CNN-based applications executed at the Edge. The first limitation is that a CNN is always executed layer-by-layer. This sequential manner of CNN execution is widespread due to its simplicity, but cannot guarantee efficient utilization of the resources available on the edge

platform. Consequently, a CNN designed using the SOTA design flow may utilize the limited resources of an edge platform inefficiently. The second limitation is that a CNN-based application only uses one CNN to perform its task. Due to this limitation, the SOTA design flow lacks the means for inter-CNN optimizations and run-time adaptivity, which are important to some CNN-based applications.

In this thesis, we aim to relax the two aforementioned limitations and reduce their negative impact on the design of CNN-based applications executed at the Edge. To this end, we extend the SOTA design flow and propose four novel methodologies within the extended design flow.

The first two methodologies focus on relaxing the first limitation. These methodologies find and enforce a non-sequential manner of CNN execution to ensure efficient utilization of the platform resources by a CNN. The first methodology efficiently distributes (maps) the computations within a CNN to the computational resources of a target edge platform and thereby increases the CNN throughput. The second methodology splits the data exchanged between CNN layers into parts and reuses the platform memory among the data parts, thus reducing the memory footprint of the CNN.

The last two methodologies focus on relaxing the second limitation. These methodologies optimize CNN-based application beyond optimizing the individual CNNs. The third methodology introduces run-time adaptivity into a CNN-based application. This enables for the design and efficient execution of an application which needs can change at run-time. The fourth methodology performs joint memory optimization of multi-CNN applications (applications that use multiple CNNs to perform their task). Thus, the methodology offers high rates of memory compression to fit multi-CNN applications into the limited memory resources of an edge platform.

Samenvatting

Convolutionele Neurale Netwerken (CNN's) zijn biologisch geïnspireerde modellen die in staat zijn grote hoeveelheden ongestructureerde data te verwerken. Door dit vermogen blinken CNN's uit in taken zoals beeldclassificatie, beeldsegmentatie en natuurlijke taalverwerking, en worden ze veel gebruikt om dergelijke taken uit te voeren in toepassingen zoals navigatie, gezichtsherkenning, medische beeldanalyse en meer. Tegenwoordig worden veel op CNN's gebaseerde applicaties uitgevoerd op zogeheten edge-platformen: mobiele telefoons, tablets, camera's, enz. Dit staat in contrast met de meer gebruikelijke aanpak waarbij de applicaties worden uitgevoerd op datacenters (in de cloud). In tegenstelling tot uitvoering in de cloud, vereist uitvoering op edge-platformen geen overdracht van de verzamelde gegevens (bv. beelden van een beveiligingscamera) via het internet, en garandeert zo een betere reactiesnelheid en veiligheid.

De uitvoering van op CNN's gebaseerde applicaties op edge-platformen is echter uitdagend vanwege de vereisten die aan de netwerken worden gesteld door de applicatie en het beoogde edge-platform. Van deze vereisten zijn de meest voorkomende: hoge nauwkeurigheid, hoge doorvoer, lage latentie, lage geheugenkosten en lage energiekosten. Deze vereisten maken het ontwerp van een CNN uitgevoerd op een edge-platform een complexe taak. Meestal wordt deze taak uitgevoerd met behulp van het state-of-the-art (SOTA) ontwerpprocess. Het SOTA-ontwerpprocess verkent CNN's met verschillende architecturen en parameters en probeert een CNN te vinden dat voldoet aan alle eisen die eraan worden gesteld.

Hoewel het SOTA-ontwerpprocess goed uitzoekt *wat* moet worden uitgevoerd op het edge-platform (d.w.z. welke architectuur en welke parameters een CNN moet hebben), onderzoekt het niet *hoe* CNN's of applicaties gebaseerd op CNN's moeten worden uitgevoerd (d.w.z. hoe het de rekenkracht, geheugen en energie gebruikt die beschikbaar zijn op een edge-platform). In plaats daarvan past het SOTA-ontwerpprocess principes toe die een negatief effect hebben op het ontwerp van CNN's en op CNN's gebaseerde applicaties uit-

gevoerd op edge-platformen. De eerste beperking is dat een CNN altijd laag-voor-laag wordt uitgevoerd. Deze sequentiële manier van CNN-uitvoering is wijdverbreid vanwege zijn eenvoud, maar kan het efficiënt gebruik van de beschikbare middelen op het edge-platform niet garanderen. Daardoor kan een CNN dat is ontworpen met behulp van het SOTA-ontwerpprocess de beperkte middelen van een edge-platform inefficiënt gebruiken. De tweede beperking is dat een CNN-gebaseerde applicatie slechts één CNN gebruikt om zijn taak uit te voeren. Hierdoor mist het SOTA-ontwerpprocess de middelen voor inter-CNN-optimalisaties en aanpassingsvermogen tijdens operatie (run-time adaptivity), welke belangrijk zijn voor bepaalde op CNN's gebaseerde applicaties.

In dit proefschrift richten we ons op het versoepelen van de twee bovengenoemde beperkingen, en het verminderen van de negatieve impact daarvan op het ontwerp van op CNN's gebaseerde applicaties uitgevoerd op edge-platformen. Hiervoor breiden we het SOTA-ontwerpprocess uit en stellen we vier nieuwe methodologieën voor binnen het uitgebreide ontwerpprocess.

De eerste twee methodieken richten zich op het versoepelen van de eerste beperking. Deze methodologieën vinden en handhaven een niet-sequentiële manier van CNN-uitvoering om efficiënt gebruik van middelen op een beoogd platform te garanderen. De eerste methodologie verdeelt de berekeningen van een CNN efficiënt onder de rekenkernen van een beoogd edge-plaform en verhoogt daardoor de doorvoer. De tweede methode splitst de gegevens die tussen CNN-lagen worden uitgewisseld in delen en hergebruikt het platformgeheugen tussen deze delen, waardoor het benodigde geheugen voor een CNN wordt verminderd.

De laatste twee methodieken richten zich op het versoepelen van de tweede beperking. Deze methodologieën optimaliseren applicaties gebaseerd op CNN's, buiten het optimaliseren van individuele CNN's om. De derde methode introduceert aanpassingsvermogen tijdens operatie in een CNN applicatie. Dit maakt het mogelijk een applicatie te ontwerpen en efficiënte uit te voeren, dat zich kan aanpassen aan de noden van het moment. De vierde methodologie voert gezamenlijke geheugenoptimalisatie uit over een multi-CNN-applicatie (een applicatie die meerdere CNN's gebruikt om een taak uit te voeren). De methodologie maakt hoge geheugencompressie mogelijk om zo de multi-CNN-applicatie in de beperkte geheugencapaciteiten van een edge-platform te passen.

List of Publications

Journal Articles

- **Svetlana Minakova** and Todor Stefanov. "Memory-Throughput Trade-off for CNN-based Applications at the Edge". *Accepted for publication in ACM Transactions on Design Automation of Electronic Systems (TODAES), March 2022.*

- **Svetlana Minakova**, Dolly Sapra, Todor Stefanov, Andy Pimentel. "Scenario Based Run-time Switching for Adaptive CNN-based Applications at the Edge". *In ACM Transactions on Embedded Computing Systems (TECS), vol. 21, Iss. 2, Article 14, March 2022.*

- Paola Busia, **Svetlana Minakova**, Todor Stefanov, Luigi Raffo, Paolo Meloni. "ALOHA: A Unified Platform-Aware Evaluation Method for CNNs Execution on Heterogeneous Systems at the Edge". *In IEEE Access, vol. 9, September 2021.*

Peer-Reviewed Conference Proceedings

- Erqian Tang, **Svetlana Minakova**, Todor Stefanov. "Energy-efficient and High-throughput CNN Inference on Embedded CPUs-GPUs MPSoCs", *In Proceedings of the 21th International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS'21)*, Virtual Conference, July 04-08, 2021.

- **Svetlana Minakova**, Erqian Tang, Todor Stefanov. "Combining Task- and Data-level Parallelism for High-Throughput CNN Inference on Embedded CPUs-GPUs MPSoCs". *In Proceedings of the 20th International Conference on Embedded Computer Systems: Architectures, Modeling and*

Simulation (SAMOS), pp. 18-35, Pythagoreio, Samos Island, Greece, July 05-09, 2020.

- **Svetlana Minakova** and Todor Stefanov. "Buffer Sizes Reduction for Memory-efficient CNN Inference on Mobile and Embedded Devices". *In Proceedings of 23rd Euromicro Conference on Digital System Design (DSD'20)*, pp. 133-140, Portoroz, Slovenia, August 26-28, 2020.

- Paolo Meloni, Daniela Loi, Paola Busia, Gianfranco Deriu, Andy D. Pimentel, Dolly Sapra, Todor Stefanov, **Svetlana Minakova**, Francesco Conti, Luca Benini, Maura Pintor, Battista Biggio, Bernhard Moser, Natalia Shepeleva, Nikos Fragoulis, Ilias Theodorakopoulos, Michael Masin, and Francesca Palumbo. "Optimization and deployment of CNNs at the edge: the ALOHA experience ". *In Proceedings of the ACM International Conference on Computing Frontiers 2019 (CF'19)*, pp. 326-332, Alghero, Italy, Apr. 30 - May 2, 2019.

- Paolo Meloni, Daniela Loi, Gianfranco Deriu, Andy D. Pimentel, Dolly Sapra, Bernhard Moser, Natalia Shepeleva, Francesco Conti, Luca Benini, Francesca Palumbo, Michael Masin, Oscar Ripolles, David Solans, Maura Pintor, Battista Biggio, Todor Stefanov, **Svetlana Minakova**, Nikos Fragoulis, and Ilias Theodorakopoulos. "Architecture-aware design and implementation of CNN algorithms for embedded inference: the ALOHA project". *In Proceedings of the 30th International Conference on Microelectronics (ICM'18)*, pp. 52-55, Sousse, Tunisia, Dec. 16-19, 2018.

- Paolo Meloni, Daniela Loi, Gianfranco Deriu, Andy D. Pimentel, Dolly Sapra, Bernhard Moser, Natalia Shepeleva, Francesco Conti, Luca Benini, Francesca Palumbo, Michael Masin, Oscar Ripolles, David Solans, Maura Pintor, Battista Biggio, Todor Stefanov, **Svetlana Minakova**, Nikos Fragoulis, and Ilias Theodorakopoulos. "ALOHA: an architectural-aware framework for deep learning at the edge". *In Proceedings of INTelligent Embedded Systems Architectures and Applications (INTESA'18)*, Turin, Italy, Oct. 4, 2018.

Curriculum Vitae

Svetlana Minakova was born on January 31, 1993 in Ryazan, Russian Federation. She obtained her B.Sc. degree in informatics and computer engineering from Bauman Moscow State Technical University, Moscow, Russian Federation, in 2015 and the M.Sc. degree in informatics and computer engineering from Bauman Moscow State Technical University, Moscow, Russian Federation, in 2017. In January 2018 she joined the Leiden Embedded Research Center (LERC), part of the Leiden Institute of Advanced Computer Science (LIACS) at Leiden University, as a Ph.D. candidate. Her research work, which resulted in this thesis, has received funding from the European Unions Horizon 2020 Research and Innovation project under grant agreement No. 780788. Besides her work as a researcher, she has been teaching assistant for courses such as Digital Systems Design and Embedded Systems and Software. Since September 2022 she has been working as an Applied AI/ML Scientist at Signify, The Netherlands.

Acknowledgments

My PhD study comes to an end, and what a journey it's been! I have learned much, visited many places I've never been before and met many amazing people. In this section, I would like to express my gratitude to everyone who was beside me throughout my PhD journey.

First of all, I would like to thank my supervisor, **Todor Stefanov**, who gave me an opportunity to join LERC group at Leiden University as a PhD student. Thank you, **Todor** for this opportunity as well as for you guidance and patience and all nice and fruitful discussions we had.

I was fortunate to perform my PhD studies as a part of the large **ALOHA consortium**. While I am grateful to every member of the consortium, I would like to express my special thanks to my colleagues from University of Amsterdam, **Andy** and **Dolly** (or should I say **professor Pimentel** and **doctor Sapra**), as well as to my colleagues from the University of Cagliari, **Paolo Meloni** and **Paola Busia**. I enjoyed the time we spent together discussing research ideas, travelling and just chatting.

I am also grateful that I got to spend time with my colleagues from Leiden University **Sobhan, Erqian, Peng, Xiaotian**, and **Faezeh**. **Sobhan**, special thanks to you for your tips and help in my job search following my formal graduation.

To my friends from Russia, **Alexander Maltsev, Vladimir Vysochansky, Ivan Chernenky, Valeria Zharova, and Alexey Leontiev**: thank you for keeping my spirits up. I am happy we stay in touch even while being scattered all over the globe. To my new friends here in The Netherlands, **Remco, Daila, Milan, Ilse-Marie, Erik, Jonas, Oskar, Bas, Jorke, Arthur, Niek, Bart, Wouter** and **Elea**: thank you for the fun and board games we've had together.

I would like to thank my family. To my parents, **Anna** and **Yuri**: thanks you for supporting me in my move to Europe, and for sending me care packages (with delicious mushrooms, chocolates and teas) and love over such long distances. And to the parents of my partner as well, **Chris** and **Wim**: thank you for supporting me during the difficult times, and also for the most wonderful

meals and wines I have ever had.

To my cousin **Radmir Gaynutdinov**. Thank you **Radmir**, for believing in me and for sharing your expertise in working in a scientific environment.

A special place in my acknowledgements goes to my grandmother, **Kulumbetova Liya Nigmatovna**. A physicist, a bright person with a curious never-resting mind. She was a great inspiration to me. Not only was she an example to me, but she also taught me how to approach complex problems. Furthermore, she greatly supported me in my decision to pursue my studies and later a career in the field of computer science, where I found my happiness and purpose.

The final and biggest thanks goes to my dear **Siebren**. I cannot quite express all my gratitude for all the love, support and kindness you have surrounded me with during my PhD journey. For all the feedback you have provided me after reading the first drafts of my papers, and for all extra commas you have removed from the last drafts. For your willingness to share every moment of my journey, no matter whether it was a happy or a sad moment.