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

Minakova, S.

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,

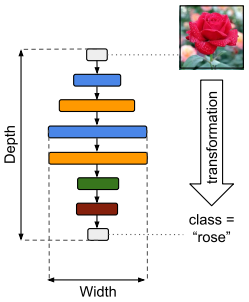

Figure 1.1: *CNN*

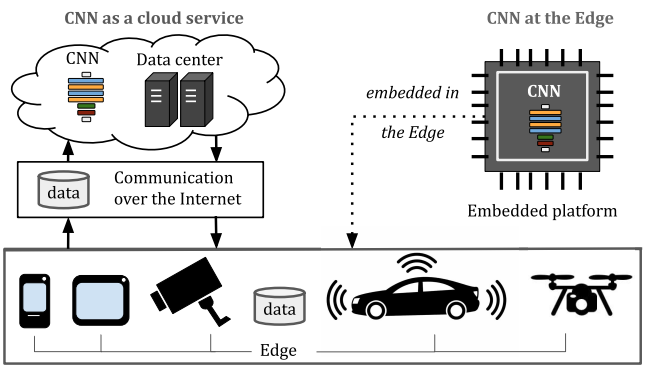

Figure 1.2: *Execution of CNNs as cloud services and at the Edge*

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

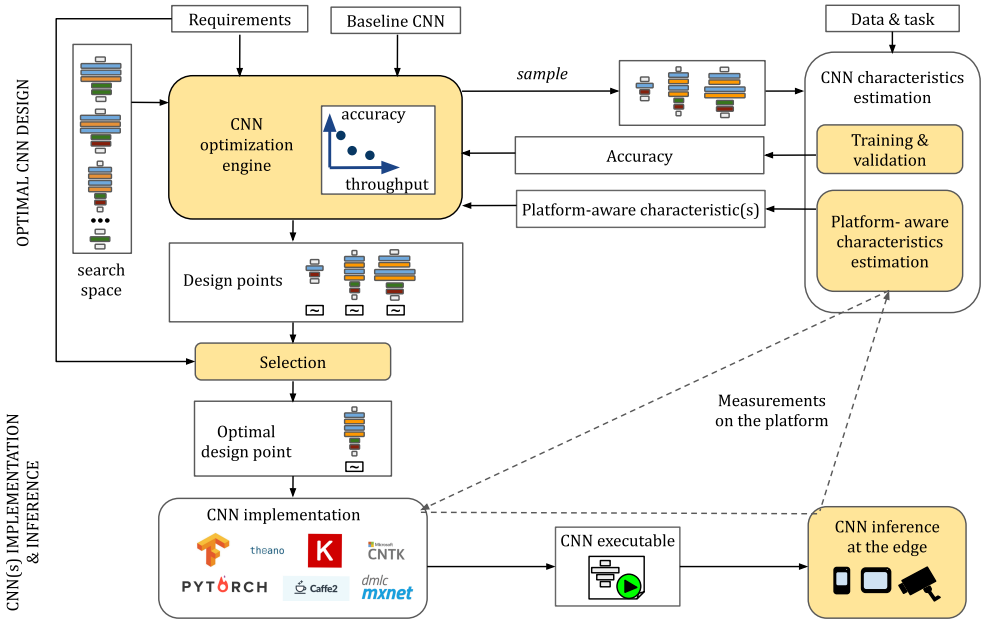

Figure 1.3: *Current trends in the design and inference of CNN-based applications executed at the Edge*

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

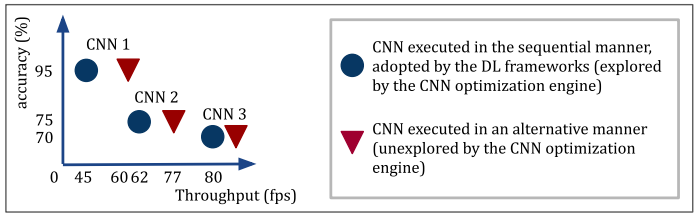

Figure 1.4: *CNNs associated with alternative manners of execution*

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

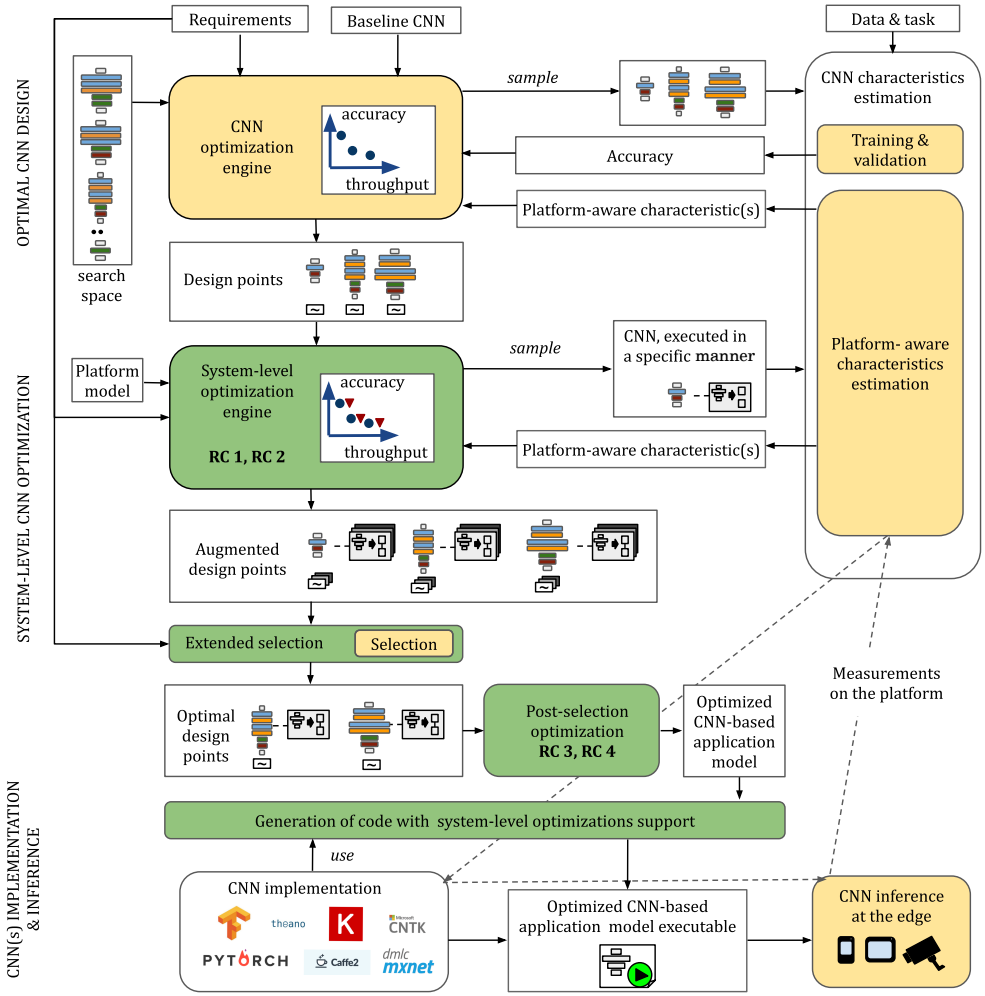

Figure 1.5: *Extended CNN design flow*

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.