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

Minakova, S.

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)

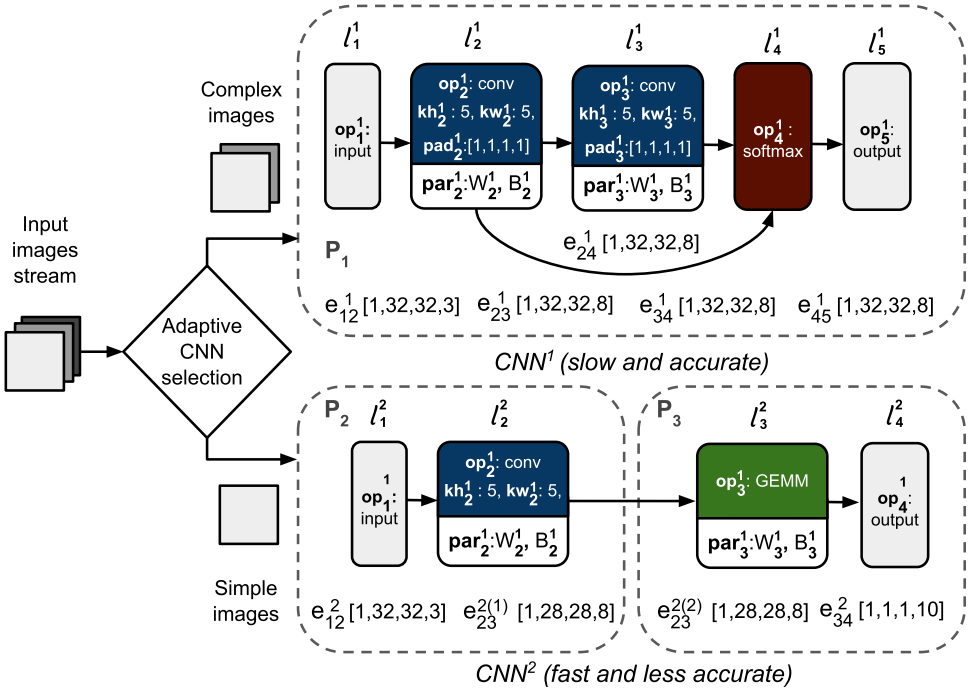

Figure 6.1: *Example CNN-based application APP*

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.

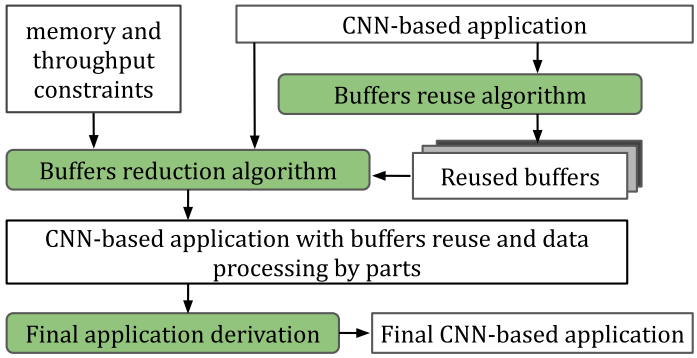

Figure 6.2: *Our methodology design flow*

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),

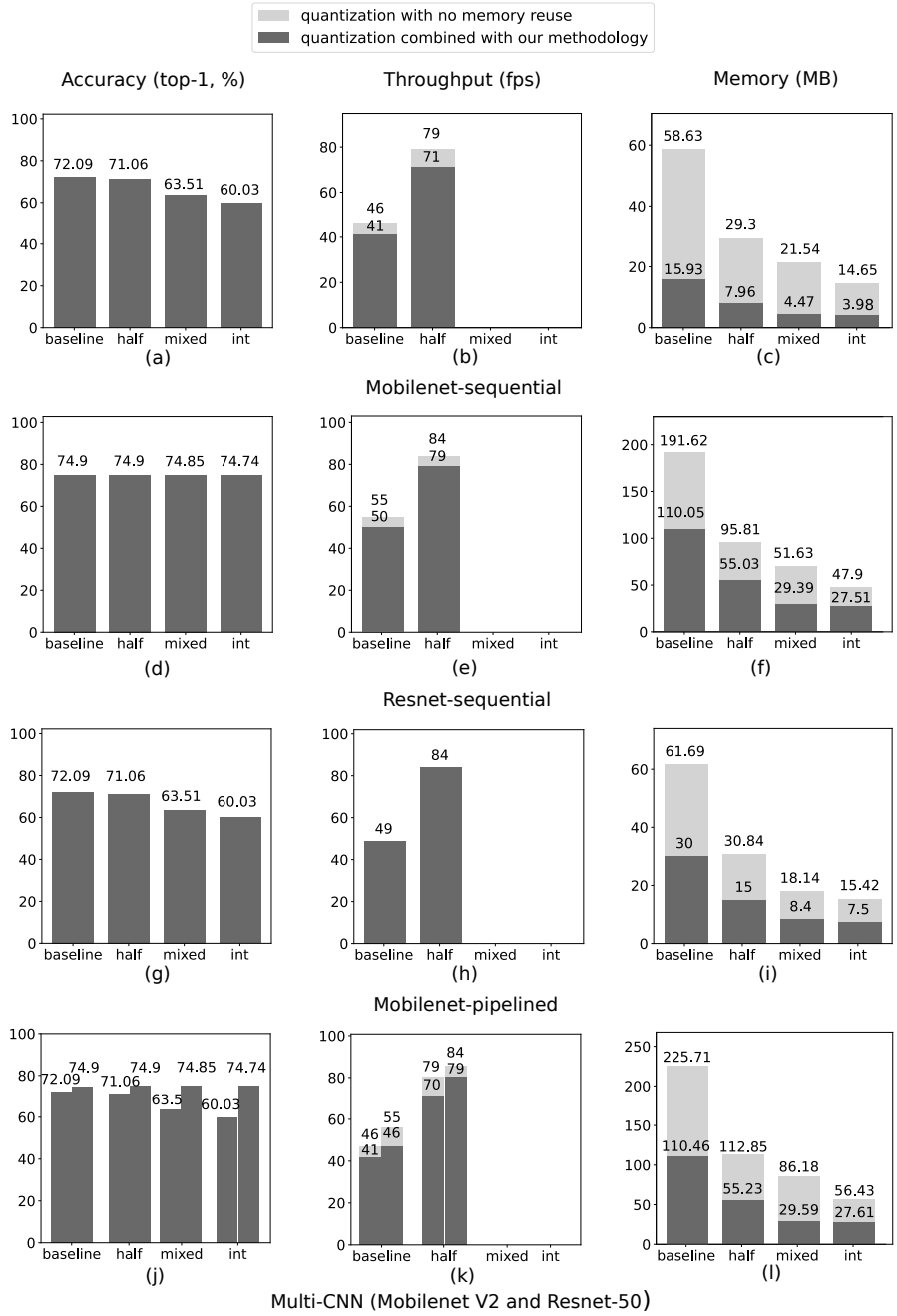

Figure 6.3: *Experimental results*

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.