Memory-Throughput Trade-off for CNN-based Applications at the Edge

SVETLANA MINAKOVA, Leiden University, The Netherlands
TODOR STEFANOV, Leiden University, The Netherlands

Many modern applications require execution of Convolutional Neural Networks (CNNs) on edge devices, such as mobile phones or embedded platforms. This can be challenging as the state-of-the art CNNs are memory-costly, whereas the memory budget of edge devices is highly limited. To address this challenge, a variety of CNN memory reduction methodologies have been proposed. Typically, the memory of a CNN is reduced using methodologies such as pruning and quantization. These methodologies reduce the number or precision of CNN parameters, thereby reducing the CNN memory cost. When more aggressive CNN memory reduction is required, the pruning and quantization methodologies can be combined with CNN memory reuse methodologies. The latter methodologies reuse device memory allocated for storage of CNN intermediate computational results, thereby further reducing the CNN memory cost. However, the existing memory reuse methodologies are unfit for CNN-based applications that exploit pipeline parallelism available within the CNNs or use multiple CNNs to perform their functionality. In this paper, we therefore propose a novel CNN memory reuse methodology. In our methodology, we significantly extend and combine two existing CNN memory reuse methodologies to offer efficient memory reuse for a wide range of CNN-based applications.

CCS Concepts: • **Computing methodologies → Neural networks**; • **Computer systems organization → Embedded software**; • **Hardware → Emerging tools and methodologies**.

Additional Key Words and Phrases: convolutional neural networks, AI at the edge, memory reduction, trade-off

ACM Reference Format:
Svetlana Minakova and Todor Stefanov. 2022. Memory-Throughput Trade-off for CNN-based Applications at the Edge. *ACM Trans. Des. Autom. Electron. Syst.* 1, 1, Article 1 (January 2022), 25 pages. https://doi.org/10.1145/3527457

1 INTRODUCTION

Many modern applications are based on Convolutional Neural Networks (CNNs): biologically inspired computational models that are extremely effective at processing multi-dimensional data and solving tasks such as images classification, objects detection and others [3]. With recent trends in the fields of Deep Learning (DL) and Edge Computing, more and more CNN-based applications are executed on edge devices such as mobile and embedded platforms [16]. Typical reasons for deployment of CNN-based applications at the edge are privacy (some applications require local storage of their data), high responsiveness (embedded platforms can guarantee real-time response) and energy efficiency (embedded platforms consume much less energy than high-performance cloud-based servers) [16].

The deployment of state-of-the-art CNN-based applications often involves hosting one or more memory-costly CNNs on a target edge device, whereas the memory budget of edge devices is very limited. Thus, it may occur that a CNN-based application does not fit into the limited memory budget of a target edge device. To tackle this problem CNN

Authors' addresses: Svetlana Minakova, s.minakova@liacs.leidenuniv.nl, Leiden University, Niels Bohrweg 1, Leiden, South Holland, The Netherlands, 2333 CA; Todor Stefanov, t.p.stefanov@liacs.leidenuniv.nl, Leiden University, Niels Bohrweg 1, Leiden, South Holland, The Netherlands, 2333 CA.

memory reduction methodologies [7, 15, 17, 23, 30] have been proposed. These methodologies reduce the memory cost of CNNs significantly and thus enable for fitting of a memory-costly CNN-based application into the limited memory of an edge device.

The most common of these methodologies are pruning and quantization methodologies such as the methodologies reviewed in surveys [6, 7, 11, 30]. These methodologies reduce the number or precision of CNN parameters, thereby reducing the CNN memory cost and increasing the CNN throughput. However, at high CNN memory reduction rates these methodologies may decrease the CNN accuracy.

Orthogonal to the pruning and quantization methodologies, the methodologies in [15, 17, 23] reuse platform memory allocated to store intermediate CNN computational results produced by the CNN layers. These methodologies do not change the CNN parameters and therefore enable to further reduce the CNN memory cost without decreasing the CNN accuracy. To achieve high CNN memory reduction and avoid substantial decrease of the CNN accuracy, one can combine the CNN pruning and quantization methodologies with the CNN memory reuse methodologies. Existing CNN memory reuse methodologies are however unfit for some CNN-based applications.

For example, CNN buffer reuse methodologies such as the methodologies in [15, 23] reuse platform memory allocated to store intermediate CNN computational results produced by different CNN layers. Thus, these methodologies reduce the CNN memory cost at no expense. However, these methodologies are not suitable for applications that utilize several CNNs, e.g. [26, 27, 29], or CNN-based applications exploiting task-level (pipeline) parallelism [18, 31] available within the CNNs. Moreover, these methodologies are not very efficient for CNNs with residual connections, such as ResNets [12] and DenseNets [14], that have to simultaneously store large amounts of intermediate CNN computational results.

In addition, the CNN memory reuse methodology proposed in [17] reuses platform memory allocated for different partitions of input data processed by the CNN layers. This methodology does not reduce the CNN accuracy. Instead, it involves CNN memory-throughput trade-off caused by synchronization among the CNN input data partitions. As noted in [17], the rapidly growing computational power of edge devices, allowing for high CNN throughput, makes the memory-throughput trade-off preferred over the memory-accuracy trade-off for many state-of-the-art CNN-based applications. However, the trade-off offered by this methodology is unbalanced: it often involves more throughput decrease than necessary to fit a CNN-based application into the memory of a target edge device. Thus, this methodology involves unnecessary CNN throughput decrease, undesired for many CNN-based applications [8, 10].

Based on the discussion above, we argue that existing work still lacks a CNN memory reuse methodology which:

(1) does not introduce accuracy decrease into CNN-based applications;
(2) is suitable for a wide range of CNN-based applications including multi-CNN applications (applications that utilize several CNNs), CNN-based applications exploiting task-level (pipeline) parallelism and CNN-based applications utilizing CNNs with residual connections;
(3) does not introduce unnecessary throughput reduction to a CNN-based application.

In this paper, we propose a methodology fitting the criteria mentioned above. Our methodology significantly extends and combines the existing CNN memory reduction methodologies proposed in [23] and [17] to enable for efficient trade-off between CNN memory and CNN throughput for a wide range of CNN-based applications. Our methodology consists of three main steps. At Step 1 (Section 5), we introduce CNN buffer reuse into the CNN-based application, thereby reducing the application memory cost. To perform this step we propose and utilize a buffer reuse algorithm. Unlike other CNN buffer reuse algorithms [15, 23], our proposed algorithm is suitable for multi-CNN applications and

CNN-based applications exploiting task-level (pipeline) parallelism. As mentioned above, the reuse of CNN buffers does not affect the throughput and accuracy of a CNN-based application. However, it might be insufficient to fit the application into the limited memory of an edge device, especially if the application utilizes CNNs with residual connections. In such cases, we perform Step 2 (Section 6), where we further reduce the memory cost of the CNN-based application at the expense of CNN-based application throughput decrease. At Step 2 we propose and utilize a buffers size reduction algorithm. This algorithm introduces data processing by parts, initially proposed in [17], and buffers reuse proposed in Section 5 to a CNN-based application. Unlike the methodology in [17], our buffers reduction algorithm does not introduce data processing by parts into every layer of every CNN used by the application. Instead, it searches for a subset of layers that have to process data by parts to fit the application into a predefined memory constraint. The data processing by parts employed by these layers, in combination with the buffers reuse, introduces a balanced memory-throughput trade-off in a CNN-based application. Finally, at Step 3 (Section 7), we derive a final CNN-based application with reduced memory cost.

Paper contributions

In this paper, we propose a novel methodology for balanced trade-off between the memory cost and throughput of CNN-based applications. Our main contribution is our methodology presented in Section 4. Other important novel contributions are:

- A CNN buffer reuse algorithm, suitable for multi-CNN applications and CNN-based applications, using task-level (pipeline) parallelism (Section 5);
- A CNN buffers size reduction algorithm (Section 6) , which combines data processing by parts with buffers reuse and introduces a balanced memory-throughput trade-off to a CNN-based application;
- up to 5.9 times memory reduction compared to deployment of CNN-based applications with no memory reduction (Section 8.1);
- 7% to 30% memory reduction compared to other CNN memory reuse methodologies (Section 8.1).

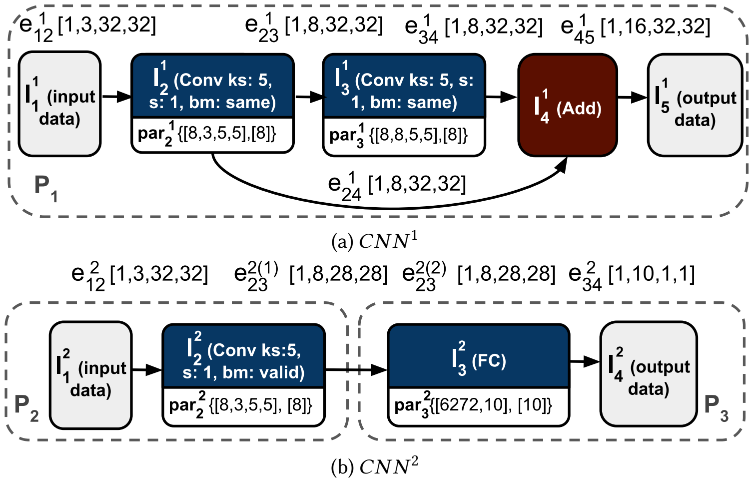
Additionally, in Section 8.2 we demonstrate that our methodology can be efficiently combined with orthogonal memory reduction methodologies such as CNN quantization.

2 BACKGROUND

In this section we provide a brief description of: 1) the CNN computational model (Section 2.1); 2) the parallelism available within a CNN (Section 2.2); 3) a CNN-based application (Section 2.3); 4) estimation of the memory cost of a CNN-based application (Section 2.4); 5) the data processing by parts in the CNN layers (Section 2.5). This section is essential for understanding the proposed methodology.

2.1 CNN computational model

A convolutional neural network (CNN) is a computational model [2], commonly represented as a directed acyclic computational graph $CNN(L, E)$ with a set of nodes L, also called layers, and a set of edges E. An example of a CNN model with $|L| = 5$ layers and $|E| = 5$ edges is given in Figure 1(a). Every layer $l_i \in L$ represents part of the CNN functionality. It performs operator op_i (such as Convolution, Pooling, etc.), parameterized with hyper-parameters hyp_i (such as kernel size, stride, borders processing mode etc.) and learnable parameters par_i (such as weights and biases). We define a layer as a tuple $l_i = (op_i, hyp_i, par_i)$, where op_i is the operator of l_i; hyp_i are the hyper-parameters of l_i; par_i is



(a) CNN^1

(b) CNN^2

Fig. 1. CNN computational model

a list of multi-dimensional arrays, called tensors [2], where every tensor $par_{ik} \in par_i$ stores a set of learnable parameters (weights or biases) or layer l_i. An example of a CNN layer $l_2^1 = (Conv, \{ks : 5, s : 1, bm : same\}, \{[8, 3, 5, 5], [8]\}$ is shown in Figure 1(a). Layer l_2^1 performs Convolutional operator $op_2^1 = Conv$, parameterized with three hyper-parameters (kernel size $ks = 5$, stride $s = 1$ and borders processing mode $bm = same$) and parameters $par_2^1 = \{[8, 3, 5, 5], [8]\}$, where $[8, 3, 5, 5]$ is a four-dimensional tensor of the layer weights and $[8]$ is one-dimensional tensor of the layer biases.

Every edge $e_{ij} \in E$ specifies a data dependency between layers l_i and l_j, such that data produced by layer l_i is accepted as an input by layer l_j. We define an edge as a tuple $(i, j, data)$, where i and j are the indexes of the layers connected by edge e_{ij}; $data$ is the data exchanged between layers l_i and l_j and stored in a tensor of shape $[batch, Ch, H, W]$, where $batch, Ch, H, W$ are the tensor batch size [2], the number of channels, the height and the width, respectively. An example of edge $e_{12}^1 = (1, 2,$ [1,3,32,32] $)$ is shown in Figure 1(a). Edge e_{12}^1 represents a data dependency between layers l_1^1 and l_2^1, where layer l_1^1 produces a data tensor [1,3,32,32] with batch size = 1, number of channels = 3, height and width = 32, accepted as input by layer l_2^1.

2.2 Parallelism, available within a CNN

As a computational model the CNN is characterized with large amount of available parallelism. This parallelism can be exploited to speed-up the CNN inference and to efficiently utilize the computational resources of a platform where the CNN is deployed. The most well-known and widely exploited type of parallelism available within the CNNs is *data-level parallelism*. This type of parallelism involves the same computation, e.g. Convolution, performed by a CNN layer over the CNN layer input data partitions. It allows to speed-up CNN inference by accelerating the execution of individual CNN layers on parallel processors such as Graphics Processing Units (GPUs) or Field Programmable Gate Arrays (FPGAs). The data-level parallelism available within the CNNs is exploited by most of the existing Deep Learning frameworks, such as TensorFlow [1] or PyTorch [22].

Another type of parallelism available within a CNN is known as *task-level parallelism* or *pipeline parallelism* [18, 31] among the CNN layers. This type of parallelism is related to the streaming nature of CNN-based applications, where the application accepts different input frames (images) from an input data stream. When a CNN is executed on a platform with multiple processors, the frames from the input data stream can be processed in a pipelined fashion by different

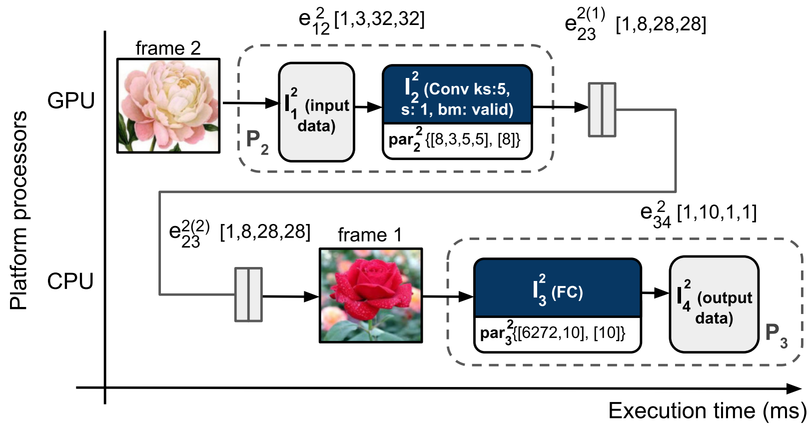


Fig. 2. Execution of CNN^2 as a pipeline

layers of the CNN deployed on different processors. Figure 2 shows an example where CNN^2, introduced and explained in Section 2.1, is executed in a pipelined fashion on a platform with two processors: a Central Processing Unit (CPU) and a GPU.

The layers of CNN^2, representing computations within the CNN, are distributed over the platform processors: layers l_1^2 and l_2^2 are executed on the GPU, while layers l_3^2 and l_4^2 are executed on the CPU. These layers form two CNN sub-graphs also referred as *partitions* [18, 31], annotated as P_2 and P_3. Partition P_2 accepts frames from the application input data stream, processes these frames as specified by layers l_1^2 and l_2^2 and stores the results into a buffer associated with edge e_{23}^2. Partition P_3 accepts the frames processed by partition P_2 from edge e_{23}^2 further processes these frames and produces the output data of CNN^2. Partitions P_2 and P_3 are executed on different processors in the platform and do not compete for the platform computational resources. Thus, when applied to different data (i.e. different frames), the partitions can be executed in parallel. In Figure 2, partitions P_2 and P_3 process frames *frame 2* and *frame 1* in parallel. This leads to overlapping execution of layers belonging to different partitions and enables for faster inference of CNN^2 compared to conventional layer-by-layer execution. However, pipelined CNN execution involves memory overheads. As shown in Figure 2, edge e_{23}^2 of CNN^2 is duplicated between the partitions P_2 and P_3 (see edges $e_{23}^{2(1)}$ and $e_{23}^{2(2)}$ and the corresponding buffers). Such duplication, called the double-buffering [13], is necessary for execution of the CNN as a pipeline. It prevents competition between the partitions when accessing data associated with edge e_{23}^2. If the double buffering is not enabled the CNN partitions compete for access to edge e_{23}^2, creating stalls in the pipeline and reducing the CNN throughput.

2.3 CNN-based application

A CNN-based application is an application which requires execution of one or multiple CNNs to perform its functionality. When deployed on a target edge device, a CNN-based application utilizes memory and computational resources of the device to execute the CNNs.

The memory of the edge device is used to store parameters (weights and biases) and intermediate computational results. The platform memory allocated to store the CNNs intermediate computational results is typically defined as a set of CNN buffers [23], where every CNN buffer stores 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.

Table 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	16384	3072	6272	6272	10

The computational resources of the edge device are utilized to perform the functionality of the CNNs. Typically the CNNs are executed layer-by-layer, i.e. at every moment in time only one CNN layer is executed on the edge platform. However, as explained in Section 2.2, some of the applications execute CNNs in a pipelined fashion.

In this paper, 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. 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; $schedule_i, i \in [1, |P|]$ is a schedule of partition P_i which determines the execution order of the layers within partition P_i. Formally, we define $schedule_i$ as a set of steps, where at each step one or several layers of partition P_i are executed.

To illustrate a CNN-based application as defined above we give an example of a CNN-based application $APP = (\{CNN^1, CNN^2\}, \{B_1, ..., B_9\}, \{P_1, P_2, P_3\}, \{\{P_1\}, \{P_2, P_3\}\}, \{\{\{l^1_1\}, \{l^1_2\}, \{l^1_3\}, \{l^1_4\}, \{l^1_5\}\}, \{\{l^2_1\}, \{l^2_2\}\}, \{\{l^2_3\}, \{l^2_4\}\}\})$, inspired by the real-world CNN-based application for adaptive images classification proposed in [29]. To perform its functionality application APP uses $N = 2$ CNNs: CNN^1 and CNN^2, shown in Figure 1(a) and Figure 1(b), respectively. During its execution, application APP accepts a stream of images, also called frames, and adaptively selects one of its CNNs (CNN^1 or CNN^2) to perform the image classification of the input frame. CNN^1 consists of one partition P_1. CNN^2 consists of two partitions, P_2 and P_3, executed in a pipelined fashion, as shown in Figure 2 and explained in Section 2.2. The layers within every partition $P_i, i \in [1, 3]$ of application APP are executed sequentially (one-by-one). This is expressed through $schedule_1$, $schedule_2$, and $schedule_3$ of application APP. For example, $schedule_1 = \{\{l^1_1\}, \{l^1_2\}, \{l^1_3\}, \{l^1_4\}, \{l^1_5\}\}$ 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^1_j is executed.

To store intermediate computational results associated with every edge e^n_{ij} of CNN^1 and CNN^2, application APP uses a set of buffers B, 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 to naive buffers allocation. In total, application APP uses $|B| = 9$ CNN buffers. These buffers are shown in Table 1, where Row 1 lists the layers within every CNN buffer; 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.

2.4 CNN-based application memory cost

The memory cost M of a CNN-based application, explained in Section 2.3, is estimated as:

$$M = M^{par} + M^{buf} \tag{1}$$

where M^{par} is the amount of memory allocated to store CNN parameters (weights and biases); M^{buf} is the amount of memory allocated to the CNN buffers. M^{par} and M^{buf} are computed in Equation 2 and Equation 3, respectively.

$$M^{par} = \sum_{n=1}^{N} \sum_{i=1}^{|L|} \sum_{k=1}^{|par_i|} |par_{ik}^n| * par\_bytes \tag{2}$$

In Equation 2, N is the total number of CNNs of a CNN-based application; par_i^n is the list of parameter tensors $par_{ik}^n, k \in [1, |par_i|]$ of layer l_i^n of CNN^n; par_bytes is the size of one CNN parameter in bytes.

$$M^{buf} = \sum_{B_k \in B} B_k.size * data\_bytes \tag{3}$$

In Equation 3, $B = \{B_1, ..., B_K\}$ are the CNN buffers; $data\_bytes$ is the size of one data element in bytes; $B_k.size$ is the size of CNN buffer in tokens, computed as:

$$B_k.size = \max_{e_{ij}^n \in B_k.edges} |e_{ij}^n.data| \tag{4}$$

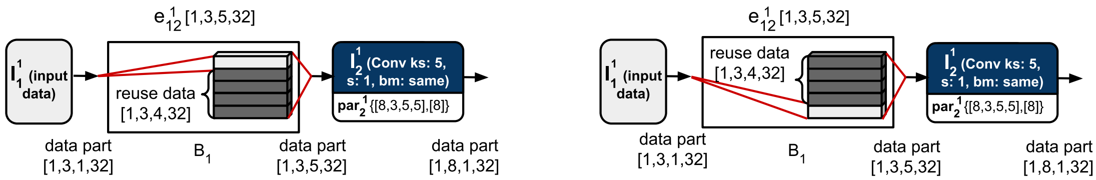
In Equation 4, $|e_{ij}^n.data|$ is the total number of elements in data tensor $e_{ij}^n.data$ associated with edge e_{ij}^n and stored in buffer B_k.

2.5 Data processing by parts in the CNN layers

Many CNN operators are characterized with the ability to process data by parts [2]. Formally, such ability can be expressed as follows: applying a CNN operator op to a data tensor $data$ can be represented as a sequence of Φ phases, where at every phase operator op is applied to a part $data'$ of the tensor $data$. For example, applying CNN operator $conv$ to data tensor [1,3,32,32] associated with edge e_{12}^1 (shown in Figure 1(a) and explained in Section 2.1) can be represented as a sequence of 32 phases, where at each phase operator $conv$ is applied to a part [1,3,5,32] of data [1,3,32,32]. The CNN memory reduction methodology proposed in [17] exploits such data processing by parts to reduce the CNNs memory cost. In this methodology every layer l_i of a CNN processes data in Φ_i phases. At each phase layer l_i accepts a part of the input data, applies operator $l_i.op$ to this part of data and produces the corresponding part of the output data. Each part of input and output data of layer l_i is characterized with minimum height. The minimum height of the data parts as well as the number of phases Φ_i are determined by operator $l_i.op$ performed by layer l_i, hyperparameters hyp_i of layer l_i and the data tensors associated with the input and output edges of layer l_i. Table 2 shows how the minimum input and output data height and corresponding number of phases are computed for layers performing the most common CNN operators. In Table 2, Column 1 lists the most common CNN operators $l_i.op$ performed by the CNN layers; Columns 2 and 3 show the minimum height of input and output data of layer l_i; Column 4 shows the number of phases Φ_i performed by layer l_i. For example, Row 2 in Table 2 shows that layer l_i performing operator $conv$ or operator $pool$ can process data in Φ_i phases, where Φ_i is computed as the height of data tensor $e_{ij}.data$ produced by layer l_i. At every phase, layer l_i accepts and processes a data part of minimum height H_{min}^{in} equal to the layer kernel size $hyp_i.ks$ and produces an output data part of height H_{min}^{out} =1.

Table 2. Data processing by parts in CNN layers

$l_i.op$	H_{min}^{in}	H_{min}^{out}	Φ_i
conv, pool	$hyp_i.ks$	1	$e_{ij}.data.H$
activation	1	1	
FC, loss	$e_{ji}.data.H$	$e_{ij}.data.H$	1



(a) Phase 1 of layer l_2^1 (b) Phase 2 of layer l_2^1

Fig. 3. Execution of layers l_1^1 and l_2^1 of CNN^1 with data processing by parts

When two layers l_i and l_j process data by parts, only the part of data exchanged between these layers, $e_{ij}.data'$, has to be stored in the memory of a target edge device at every moment in time [17]. The size of the minimum data part $e_{ij}.data'$ exchanged between layers l_i and l_j is computed as $e_{ij}.data'$=[$e_{ij}.batch, e_{ij}.Ch, H', e_{ij}.W$], where $e_{ij}.batch, e_{ij}.Ch$ and $e_{ij}.W$ are the batch size, number of channels and width of data e_{ij}; $H' \leq e_{ij}.H$ is computed as:

$$H' = \max(H^{out}_{min}(l_i), H^{in}_{min}(l_j)) \quad (5)$$

where $H^{out}_{min}(l_i)$ is the minimum height of data produced by layer l_i; $H^{in}_{min}(l_j)$ is minimum height of data accepted as input by layer l_j; $H^{out}_{min}(l_i)$ and $H^{in}_{min}(l_j)$ are determined using Table 2.

To illustrate how data processing by parts reduces the memory cost of a CNN-based application, we show an example where layers l_1^1 and l_2^1 of CNN^1, shown in Figure 1(a) and used by application *APP* explained in Section 2.3, process data by parts. The example is illustrated in Figure 3, where layer l_1^1 has 32 phases and layer l_2^1 has 32 phases. Execution of the phases of layer l_1^1 and layer l_2^1 is performed in a specific order. We formally define this order as a schedule shortly written as $\{[\{l_1^1\}]$x5, $\{l_2^1\}$, $[\{l_1^1\}, \{l_2^1\}]$x27, $[l_2^1]$x4$\}$. In the defined schedule, the square brackets enclose the repetitive (sub-sequences of) steps. At every step, a phase of a CNN layer is executed. During the first 5 steps, the first 5 phases of layer l_1^1 are executed, which is expressed at $[\{l_1^1\}]$x5 in the aforementioned schedule. At every phase, layer l_1^1 produces data part of shape $[1, 3, 1, 32]$ in buffer B_1, used to store the data exchanged between layers l_1^1 and l_2^1 as specified in Table 1 in Section 2.3. After the first 5 steps, data part of shape $[1, 3, 5, 32]$ is accumulated in buffer B_1. This part is sufficient to execute the first phase of layer l_2^1. Thus, at step 6 of the schedule, the first phase of layer l_2^1 is executed (see Figure 3 (a)). To execute the second phase of layer l_2^1 (see Figure 3 (b)), data of shape $[1, 3, 5, 32]$ should be accumulated in B_1. However, some of this data is already in B_1 because the data between subsequent execution steps of layer l_2^1 is overlapping. When the overlapping part is stored in buffer B_1, only new (non-overlapping) data should be produced in B_1 to enable the execution of the second phase of layer l_2^1. This new data can be produced by execution of one phase of layer l_1^1. Thus, phases 6-32 of layer l_1^1 and phases 2-28 of layer l_2^1 are executed in an alternating manner, where a phase of layer l_1^1 is followed by a phase of layer l_2^1, and this pattern repeats, until all phases of l_1^1 are executed. This is expressed as $[\{l_1^1\}, \{l_2^1\}]$x27 in the aforementioned schedule. Finally, the last 4 phases of layer l_2^1 are executed. The maximum amount of data, stored between layers l_1^1 and l_2^2 at any time of layers execution corresponds to data part of shape $[1, 3, 5, 32]$, accumulated in B_1. Thus, when layers l_1^1 and l_2^1 of CNN^1 process data by parts, the size of buffer B_1 is 1 * 3 * 5 * 32 = 480 data elements, which is 3072/480 ≈ 6.4 times less, compared to the size of buffer B_1 given in Table 1 in Section 2.3. Thus, by introducing data processing by parts into the CNN layers, the methodology in [17] reduces the memory cost of a CNN. 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, processing data by parts involves a trade-off between the CNN memory cost and the CNN throughput. In this paper, similarly to the methodology in [17], we exploit this trade-off to reduce the CNNs memory cost.

It is important to note that the reduction of application buffer sizes from data processing by parts requires the layers of a CNN to be executed in a specific order formally expressed as a schedule. For example, as explained above, layers l_1^1 and l_2^1 are executed in phases where the execution order of the phases is defined by the following schedule: $\{[\{l_1^1\}]x5, \{l_2^1\}, [\{l_1^1\}, \{l_2^1\}]x27, [l_2^1]x4\}$. To find a proper schedule, i.e., execution order of phases in a CNN, similarly to the methodology in [17] we: 1) perform conversion of the CNN into a functionally equivalent Cyclo-Static Dataflow (CSDF) model of computation [5], accepted as an input by many embedded systems analysis and design tools. For the description of a CNN represented as a CSDF model and details of the CNN-to-CSDF model conversion, we refer the reader to the methodology proposed in [17]; 2) use the *SDF*3 embedded systems analysis and design tool [28] to automatically derive the execution order (schedule) of the phases within a CNN. We also use the *SDF*3 tool to automatically compute the sizes of CNN buffers, when the CNN is executed with phases.

3 MOTIVATIONAL EXAMPLE

In this section we motivate the necessity of devising a new memory reduction methodology for deployment of CNN-based applications at the edge. We show an example where we design a CNN-based application executed on the NVIDIA Jetson TX2 edge platform [20]. To perform its functionality the application requires execution of the Mobilenet V2 CNN [24]. The CNN performs image classification on the ImageNet dataset [9] composed of RGB images with 224 pixels height and width. The application poses requirements on the Mobilenet V2 CNN: it requires the CNN to utilize less than 8 MegaBytes (MB) of memory, demonstrate more than 70% accuracy, and no less than 71 frames per second (fps) throughput.

As the baseline implementation of the Mobilenet V2 CNN, we take a pre-trained CNN from the applications library of the well-known and widely used TensorFlow DL framework [1]. The baseline CNN is trained and inferred with the original 32-bit floating-point (fp32) weights and data precision. When executed on the NVIDIA Jetson TX2 platform, the baseline CNN occupies 58.63 MB of memory, demonstrates 72.09 % accuracy and 46 fps throughput. Thus, the baseline CNN meets the accuracy requirement but does not meet the memory and throughput requirements.

To reduce the CNN memory cost and increase the CNN throughput we use the quantization methodology offered by the TensorFlow DL framework. The quantization methodology reduces the precision of the CNN parameters and data from the original 32-bit floating-point (fp32) precision to a lower precision such as a 16-bit floating-point (fp16) precision or a 8-bit integer (int) precision, thereby reducing the CNN memory cost and increasing the CNN throughput. The TensorFlow framework offers several types of quantization, varying in terms of target precision used to store CNN parameters and weights. The quantization types and their respective target precision are shown in Table 3. For example, the half-quantization, shown in Row 4, reduces the precision of CNN parameters and data to fp16 precision.

The characteristics of the baseline CNN after quantization, executed on the Jetson TX2 platform, are shown in Table 4. Column 1 lists the types of quantization. Column 2 shows the top-1 images classification accuracy (in %). Column 3 shows the CNN throughput (in fps). The CNN throughput is not shown for the CNNs with int- and mixed-quantization

Table 3. Quantization in the TensorFlow DL framework [1]

Quantization		
name	data	par
No (baseline)	fp32	fp32
Half	fp16	fp16
Mixed	fp16	int
Int	int	int

Table 4. MobileNet v2 CNN after quantization

Quantization	A (%)	T (fps)	M (MB)
No (baseline)	72.09	46	58.63
Half	71.06	79	29.3
Mixed	63.51	-	21.54
Int	60.03	-	14.65

Table 5. MobileNet v2 CNN after quantization and buffers reuse proposed in [23]

Quantization	A (%)	T (fps)	M (MB)
No (baseline)	72.09	46	20.32
Half	71.06	79	10.16
Mixed	63.51	-	7.46
Int	60.03	-	5.08

Table 6. MobileNet v2 CNN after quantization and data processing by parts proposed in [17]

Quantization	A (%)	T (fps)	M (MB)
No (baseline)	72.09	40	16.2
Half	71.06	68	8.1
Mixed	63.51	-	4.61
Int	60.03	-	4.04

Quantization	A (%)	T (fps)	M (MB)
No (baseline)	72.09	41	15.93
Half	71.06	71	7.96
Mixed	63.51	-	4.47
Int	60.03	-	3.98

Table 7. MobileNet v2 CNN after quantization and our methodology

because the Jetson TX2 platform does not support integer computations. Column 4 shows the CNN memory cost (in MB).

Table 4 shows that the CNN quantization leads to significant reduction of the CNN memory cost as well as increase of the CNN throughput. For example, the CNN with half-quantization, shown in Row 3 in Table 4, has 2 times smaller memory cost and $\approx$ 1.72 times higher throughput, compared to the baseline CNN, shown in Row 2. However, the memory reduction achieved by applying any type of quantization, shown in Table 4, is insufficient to meet the 8 MB memory requirement. Moreover, both int-quantization and mixed quantization significantly reduce the CNN accuracy, dropping it below the requirement of 70% accuracy.

To further reduce the memory cost of the quantized CNNs, shown in Table 4, we apply existing CNN memory reuse methodologies proposed in [23] and [17]. Table 5 shows the characteristics of the MobileNet v2 CNN after the quantization and the buffers reuse methodology proposed in [23]. A comparison between Table 4 and Table 5 shows that the methodology in [23] enables to further reduce the CNN memory cost without decreasing the CNN accuracy or throughput. For example, the CNN with half-quantization and buffers reuse has $\approx$ 3 times smaller memory cost but equal accuracy and throughput compared to the CNN with half-quantization and no buffers reuse (see Row 3 in Table 4 and in Table 5). However, none of the CNNs shown in Table 5 meets all three requirements posed on the CNN.

Analogously, Table 6 shows the characteristics of the MobileNet v2 CNN after the quantization and the methodology proposed in [17]. A comparison between Table 4 and Table 6 shows that the methodology in [17] enables to further reduce the CNN memory cost without decreasing the CNN accuracy. However, the methodology in [17] significantly reduces the CNN throughput. For example, the throughput of the CNN with half-quantization and the memory reuse proposed in [17] is reduced by 11 fps compared to the CNN with half-quantization and no memory reuse (see Row 3, Column 3 in Table 4 and in Table 6). Among the CNNs shown in Table 6, none of the CNNs meets all three requirements, posed by the application.

Table 7 shows the characteristics of the MobileNet v2 CNN after quantization combined with our novel methodology. As shown in Row 3 in Table 7, after the half-quantization combined with our methodology, the Mobilenet V2 CNN occupies 7.96 MB of memory and demonstrates 71.06 % accuracy and 71 fps throughput. This means that our methodology enables the Mobilenet V2 CNN to meet all three requirements posed on the CNN.

Based on the above motivational example and analysis, we conclude that *some CNN-based applications cannot meet their respective requirements by utilizing existing memory reduction methodologies but can meet these requirements by utilizing our proposed methodology in combination with quantization.*

4 METHODOLOGY

In this section we present our memory-throughput trade-off methodology for CNN-based applications at the edge. The design flow of our methodology is shown in Figure 4. Our methodology accepts as inputs a CNN-based application, described in Section 2.3, 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.

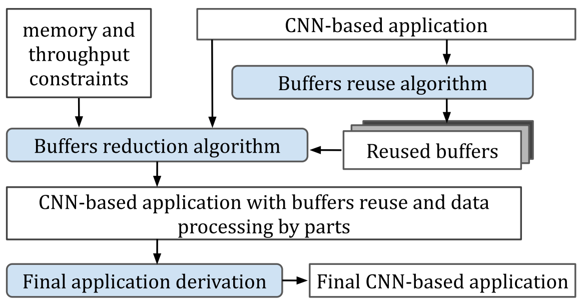


Fig. 4. Our methodology design flow

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 5. 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 (explained in Section 2.5) combined with buffers reuse (as proposed in Section 5) to the CNN-based application. We note that unlike the methodology in [17], 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. As explained in Section 2.5, the introduction of data processing by parts in a CNN requires the layers of the CNN to be executed in a specific order, defined by a proper schedule. Therefore, our buffers reduction algorithm also finds and enforces a specific schedule in the CNNs used by the application. 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 output by our methodology. This step is described in Section 7.

5 BUFFERS REUSE ALGORITHM

In this section, we present our buffers reuse algorithm, Algorithm 1, 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 1 reuses CNN buffers among the visited edges, thereby introducing memory reuse into the CNN-based application and reducing the application memory cost. Algorithm 1 accepts as an input a CNN-based application with naive buffers allocation, explained in Section 2.3. As an output Algorithm 1 produces a set of buffers B, reused among all the CNNs of

Algorithm 1: 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

Table 8. 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	16384	8192	10

the CNN-based application. An example of buffers B generated by Algorithm 1 for the example CNN-based application APP, explained in Section 2.3, is given in Table 8.

Unlike the naive CNN buffers allocation given in Table 1, the buffers in Table 8 are reused among CNNs and within the CNNs of application APP. For example, as shown in Column 2 in Table 8, CNN buffer B_1, generated by Algorithm 1, 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 3, explained in Section 2.4, the reused buffers B, produced by Algorithm 1, occupy 32778* $data\_bytes$ bytes of memory, while the initial, non-reuse buffers, given in Table 1 in Section 2.3, occupy 59658* $data\_bytes$ bytes of memory.

In Line 1, Algorithm 1 sets the CNN buffers B to an empty set. In Lines 4 to 35, Algorithm 1 visits every edge e^n_{ij} of every partition $P_m \in P$ of the CNN-based application. In Line 4, Algorithm 1 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 1 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 simultaneously 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 1 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 2.3. 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 1 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 1 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 1 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 1 assigns buffer B_{best} to edge e^n_{ij} 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^n_{ij}. Finally, in Line 36, Algorithm 1 returns the CNN buffers B.

6 BUFFERS REDUCTION ALGORITHM

In this section, we present our buffers sizes reduction algorithm, Algorithm 2. This algorithm introduces data processing by parts (explained in Section 2.5) and buffers reuse (as proposed in Section 5) 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 2 uses a multi-objective Genetic Algorithm (GA) [25]: a well-known heuristic approach, widely used for finding optimal solutions for complex design space exploration problems.

Algorithm 2 accepts as inputs: 1) a CNN-based application with naive buffers allocation, explained in Section 2.3; 2) a list of reused buffers B obtained using Algorithm 1, presented in Section 5; 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 $\{T^c_1, ..., T^c_N\}$, where $T^c_n, n \in [1, N]$ specifies the minimum throughput (in fps) which has to be demonstrated by CNN^n used by the application; 4) A set of standard user-defined GA parameters GA_par such as initial population size, number of GA iterations, mutation and crossover probabilities [25]. As outputs, Algorithm 2 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 7.

As an example, taking CNN-based application $APP = (\{CNN^1, CNN^2\}, B^{naive}, P, J, \{\{\{l^1_1\}, \{l^1_2\}, \{l^1_3\}, \{l^1_4\}, \{l^1_5\}\}, \{\{l^2_1\}, \{l^2_2\}\}, \{\{l^2_3\}, \{l^2_4\}\}\})$ introduced in Section 2.3, reused buffers B shown in Table 8, constraints $M^c = 0.02$ MegaBytes (20000 bytes), $T^c = \{0, 0\}$, and standard GA parameters GA_par [25], Algorithm 2 produces as output application $APP' = (\{CNN^1, CNN^2\}, B^{reduced}, P, J, \{\{l^1_1\}, \{l^1_2\}, [\{l^1_3\}, \{l^1_4\}, \{l^1_5\}]\text{x}32\}, \{\{l^2_1\}, \{l^2_2\}\}, \{\{l^2_3\}, \{l^2_4\}\}\})$ and a set of phases $\Phi = \{(l^1_1, 1), (l^1_2, 1), (l^1_3, 32), (l^1_4, 32), (l^1_5, 32), (l^2_1, 1), (l^2_2, 1), (l^2_2, 1), (l^2_2, 1)\}$. Application APP' uses buffers $B^{reduced}$, produced by Algorithm 2 and shown in Table 9. We note that according to Equation 3, the reduced CNN buffers produced by Algorithm 2 occupy 19712* $data\_bytes$ bytes of memory (see Table 9), while the CNN buffers obtained by only using buffers reuse occupy 32778* $data\_bytes$ bytes of memory (see Table 8). The difference occurs because, besides buffers reuse, Algorithm 2 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 2 enforces a specific execution order for the layers of CNN^1 which processes data by parts. This is expressed in APP' through $schedule'_1 = \{\{l^1_1\}, \{l^1_2\}, [\{l^1_3\}, \{l^1_4\}, \{l^1_5\}]\text{x}32\}$. The set

Table 9. 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

Table 10. Chromosome

l^1_1	l^1_2	l^1_3	l^1_4	l^1_5	l^2_1	l^2_2	l^2_3	l^2_4
0	0	1	1	1	0	0	0	0

Algorithm 2: 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: $APP^{out} = (\{CNN^1, ..., CNN^N\}, B^{reduced}, P, J, \{schedule'_1, ..., schedule'_{|P|}\}), \Phi$

1 $APP^{out} \leftarrow (\{CNN^1, ..., CNN^N\}, B, P, J, \{schedule_1, ..., schedule_{|P|}\})$;
2 M = compute memory cost of APP^{out}, using Equation 1;
3 **if** $M \leq M^c$ **then**
4     $\Phi \leftarrow \{(l_i^n, 1)\}, n \in [1, N], i \in [1, |L^n|]$;
5   **return** (APP^{out}, Φ);
6 $X \leftarrow$ binary string of length $\sum_{n=1}^{N} |L^n|$;
7 $fitness = minimize(EvalMemory(APP^{in}, X), -EvalThroughput(APP^{in}, X, 1), ..., -EvalThroughput(APP^{in}, X, N))$;
8 $pareto \leftarrow GA(X, GA\_par, fitness)$;
9 $S \leftarrow \emptyset$;
10 **for** $X \in pareto$ **do**
11   **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**
12         $S \leftarrow S \cup X$;
13 **if** $S \neq \emptyset$ **then**
14   X^{best} = select from S chromosome X with minimal memory footprint $M = EvalMemory(APP^{in}, X)$;
15 **else**
16   X^{best} = select from $pareto$ chromosome X with minimal memory footprint $M = EvalMemory(APP^{in}, X)$;
17 $(APP^{out}, \Phi) \leftarrow DeriveApplicationWithReducedBuffs(APP^{in}, X^{best})$;
18 **return** (APP^{out}, Φ);

19 **Function** $EvalMemory(APP^{in}, X)$**:**
20     $(APP^X, \Phi) \leftarrow DeriveApplicationWithReducedBuffs(APP^{in}, X)$;
21   M = compute memory cost of APP^X, using Equation 1;
22   **return** M;

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

27 **Function** $DeriveApplicationWithReducedBuffs(APP^{in}, X)$**:**
28     $B^{min} \leftarrow \emptyset$; $\Phi \leftarrow \emptyset$;
29   **for** $P_p \in APP^{in}$ **do**
30         $\Phi_p \leftarrow \{(l_i^n$, Equation 6 $(X.l_i^n))\}, l_i^n \in P_p.L$;
31         $G^p(A^p, C^p) \leftarrow$ CNN-to-CSDF (P_p, Φ_p) [17];
32         $B_p^{min}, schedule'_p \leftarrow$ use SDF3 [28] to derive minimum-sized buffers and a schedule that enables execution of partition P_p represented as CSDF model G^p with these buffers;
33         $B^{min} \leftarrow B^{min} \cup B_p^{min}$;
34         $\Phi \leftarrow \Phi \cup \Phi_p$;
35     $APP^{parts} \leftarrow (\{CNN^1, ..., CNN^N\}, B^{min}, P, J, \{schedule'_1, ..., schedule'_{|P|}\})$;
36     $B^{reduced} \leftarrow$ Algorithm 1 (APP^{parts});
37     $APP^{reduced} = (\{CNN^1, ..., CNN^N\}, B^{reduced}, P, J, \{schedule'_1, ..., schedule'_{|P|}\})$
38   **return** $(APP^{reduced}, \Phi)$;

Φ specifies that each of layers l_3^1, l_4^1, and l_5^1 in CNN^1 performs 32 phases (processes its input data by 32 parts), while layers l_1^1, l_2^1 of CNN^1 and all layers of CNN^2 perform one phase (do not process data by parts).

In Lines 1 to 3, Algorithm 2 checks if utilization of only buffers reuse is sufficient to meet the memory constraint. To perform the check, in Line 1, Algorithm 2 generates an application that employs only buffers reuse (uses buffers

B, obtained using Algorithm 1). In Lines 2 and 3, Algorithm 2 checks whether this application meets the memory constraint. If so (the condition in Line 3 is met), in Line 5, Algorithm 2 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 2 performs a GA-based search to find a set of layers that have to process data by parts. To this end, Algorithm 2 uses a standard GA with two-parent crossover and a single-gene mutation as presented in [25] and two problem-specific GA attributes: a chromosome and a fitness function [25]. The chromosome is a representation of a GA solution as a set of parameters (genes), joined into a string [25]. In Algorithm 2, 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, $|L^n|$ 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 10 gives an example of a chromosome, which specifies data processing by parts as in the example application APP', mentioned above.

The fitness-function evaluates the quality of GA solutions, represented as chromosomes, and guides the GA-based search. During the search, the fitness function should be minimized or maximized. The fitness function used by Algorithm 2 is defined in Line 7. It specifies that during the GA-based search Algorithm 2 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 2 uses function $EvalMemory$ and function $EvalThroughput$, explained in Section 6.2.

In Line 8, Algorithm 2 performs the GA-based search, which delivers a set of pareto-optimal solutions (chromosomes) called a pareto-front [25]. From this pareto-front, in Lines 9 to 16, Algorithm 2 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 2 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 2 selects the best chromosome from this subset. Otherwise, in Line 16, Algorithm 2 selects the best chromosome from the pareto-front.

In Line 17, Algorithm 2 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 function $DeriveApplicationWithReducedBuffs$, explained in Section 6.1. Finally, in Line 18, Algorithm 2 returns application APP^{out} and set Φ.

6.1 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 5, Algorithm 2 uses function $DeriveApplicationWithReducedBuffs$ defined in Lines 27 to 38. In Line 28, Algorithm 2 defines an empty set B^{min} of buffers with minimum size and no reuse, and an empty set of phases Φ. In Lines 29 to 34, Algorithm 2 visits every partition P_p in the input application APP^{in}. In Line 30, Algorithm 2 uses chromosome X and Equation 6 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 Table 2, explained in Section 2.5. 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.

$$\Phi_i^n(x) = \begin{cases} \textit{determine using Table 2} & \textit{if } x = 1 \\ 1 & \textit{otherwise} \end{cases} \tag{6}$$

In Line 31 to 32, Algorithm 2 obtains a set of buffers B_p^{min} for partition P_p, 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 2 obtains specific schedule $schedule_p'$, which enables to correctly execute partition P_p with buffers B_p^{min}. To do so, Algorithm 2 converts every CNN partition into a functionally equivalent CSDF model (Line 31) using the CNN-to-CSDF conversion procedure in [17] and feeds the obtained CSDF models to the SDF3 embedded systems design and analysis tool [28]. In Lines 33 and 34, Algorithm 2 accumulates the minimum sized buffers and phases obtained in Lines 30 to 32 in sets B^{min} and Φ, respectively. In Line 35, Algorithm 2 generates application APP^{parts} which processes data by parts as specified in chromosome X without buffers reuse. In Lines 36 to 37, Algorithm 2 introduces buffers reuse into application APP^{parts}, thereby obtaining application $APP^{reduced}$, returned as output by function $DeriveApplicationWithReducedBuffs$.

6.2 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 2 and function $EvalThroughput$ defined in Lines 23 to 24 of Algorithm 2. 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 Section 6.1. Function $EvalMemory$ computes the memory cost of application APP^X using Equation 1. 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.

7 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 2, explained in Section 6. To derive the final CNN-based application, we use a DL framework, such as TensorRT [19], 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.

8 EXPERIMENTAL RESULTS

In this section, we evaluate the efficiency of our methodology. The experiments are performed in two steps. First, in Section 8.1, we compare our proposed methodology to the existing memory reuse methodologies proposed in [23] and [17]. Then, in Section 8.2, 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 2.2; 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.

8.1 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 [23] and [17]. The comparison between our methodology and the methodologies in [23] and [17] in terms of memory reduction principles is summarized in Table 11.

Table 11. Comparison of the memory reduction principles and features associated with the memory reuse methodologies in [23], [17], and our proposed methodology

memory reuse principle or feature	[23]	[28]	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

To evaluate the efficiency of our methodology and study the impact of the memory reuse principles and features summarized in Table 11 on CNN-based applications, we apply our methodology and the methodologies in [23] and [17] to six real-world CNN-based applications from the three common categories, introduced in Section 8. The applications are listed in Column 1 in Table 12. 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 2.3; 3) reduced using the methodology in [23]; 4) reduced using the methodology in [17].

Taking into account that both the related work in [17] 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 [20]. Every CNN is implemented using the Tensorrt DL framework [19], 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 12, 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.

Columns 4 to 7 show the memory cost of the CNN-based 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

Table 12. Experimental Results

Application			Memory (MB)				Throughput (fps)			
No	CNN(s)	Memory constraint (MB)	no reduction	[23]	[17]	ours	no reduction	[23]	[17]	ours
CNN-based applications with one CNN and no exploitation of task-level (pipeline) parallelism										
1	MobileNet V2 1.0	25	58.63	20.32	16.2	20.32	46	46	40	46
		15				14.98				41
		min				14.90				40.5
2	EfficientNet B0	150	161.33	39.14	42.97	39.14	168.35	168.35	98	168.35
		40				39.14				168.35
		min				27.30				128.5
CNN-based applications, exploiting pipeline parallelism, as proposed in [18]										
3	MobileNet V2 1.0	30	61.69	20.32	17.38	30	49	46	43	49
		15				15.92				43.65
		min				15.92				43.65
4	EfficientNet B0	150	163.65	39.14	44.18	45	170.3	168.35	98.8	170.3
		50				45				170.3
		min				31.34				124.24
Multi-CNN applications										
5	Inception V2	200	380	175	226	175	94	94	67	94
	Mobilenet V1 0.25						432	432	183	432
	ResNet V1 50						55	55	46	55
	Inception V2	min				162	94	94	67	75
	Mobilenet V1 0.25						432	432	183	244
	ResNet 50						55	55	46	47
6	DenseNet121	500	625	291	184	161	52	52	37	52
	Mobilenet V1 1.0						59	59	50	59
	Resnet v1 50						55	55	46	55
	DenseNet121	min				155	52	52	37	41
	Mobilenet V1 1.0						59	59	50	54
	Resnet v1 50						55	55	46	49

relevant related work (the methodologies in [23] and [17]), 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 [23] and [17]. For example, the memory cost of application 1 can be reduced to 14.90 MB by our methodology and to 20.32 MB and 16.2 MB by the methodologies in [23] and [17], 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 [17] 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 [17]; 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 [17] always decreases the throughput of Mobilenet V2 CNN by $46 - 40 = 6$ fps. The difference occurs because, unlike the methodology in [17], our methodology searches for an optimal (balanced) memory-throughput trade-off (see Algorithm 2).

Columns 8 to 9 show that the methodology in [23] does not introduce throughput decrease into the CNN-based applications exploiting no task-level parallelism and multi-CNN applications. However, [23] 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 [23] 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 [23], 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 [23] and the methodology in [17]. For example, our methodology is able to reduce the memory of multi-CNN application 6, shown in Rows 12 to 13 in Table 12 to 155 MB. This is $\approx$ 2 times more memory reduction than offered by the methodology in [23] and $\approx$ 15% more memory reduction than offered by the methodology in [17]. The difference occurs because: 1) our methodology combines memory reuse principles offered by the methodologies in [23] and [17]; 2) Unlike the methodologies in [23] and [17], 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.

8.2 Joint use of CNN 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] and our proposed methodology to four CNN-based applications, executed on the NVIDIA Jetson TX2 edge platform [20]. The applications are summarized in Table 13 and explained in details in Section 8.2.1. 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 Section 8.2.2. The comparison of the measurements along with analysis and conclusions are presented in Section 8.2.3.

8.2.1 Experimental setup. The applications that we use to study the effectiveness of our methodology when used jointly with CNN quantization, are summarized in Table 13. 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 [9], composed of RGB images with 224 pixels height and width. The baseline topology and weights of every CNN are taken from the applications library of the TensorFlow DL framework [1], which is well-known and widely used for CNNs design and

Table 13. 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	

training. For execution at the edge, the CNNs are implemented using the Tensorrt DL framework [19], 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.

8.2.2 Experimental results. The experimental results for the four CNN-based applications, summarized in Table 13, are shown in Figure 5. 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 half-, mixed- and int-quantization offered by the TensorFlow DL framework, as well as for the baseline CNNs with no quantization and original 32-bit floating-point weights and data precision. The types of quantization offered by the TensorFlow DL framework are summarized in Table 3 and explained in details in Section 3. The bar plots 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), the CNN throughput (the second column)[1], and the CNN memory cost (the third column). For example, the bar plot in Figure 5(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 5(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 5(b) shows that our methodology decreases the throughput of the Mobilenet V2 CNN with half-quantization by $79 - 71 = 8$ fps.

8.2.3 Analysis and conclusions. In this section, we compare and analyse the experimental results, presented in Section 8.2.2.

First, we compare the CNNs accuracy. To do that, we analyse the plots shown in the first column in Figure 5. 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 5. So, we see that our methodology may decrease the CNNs throughput. For example, Figure 5(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 2.5, 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 5(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 5. The plots show that our methodology enables to further reduce the memory cost of the quantized CNNs.

[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.

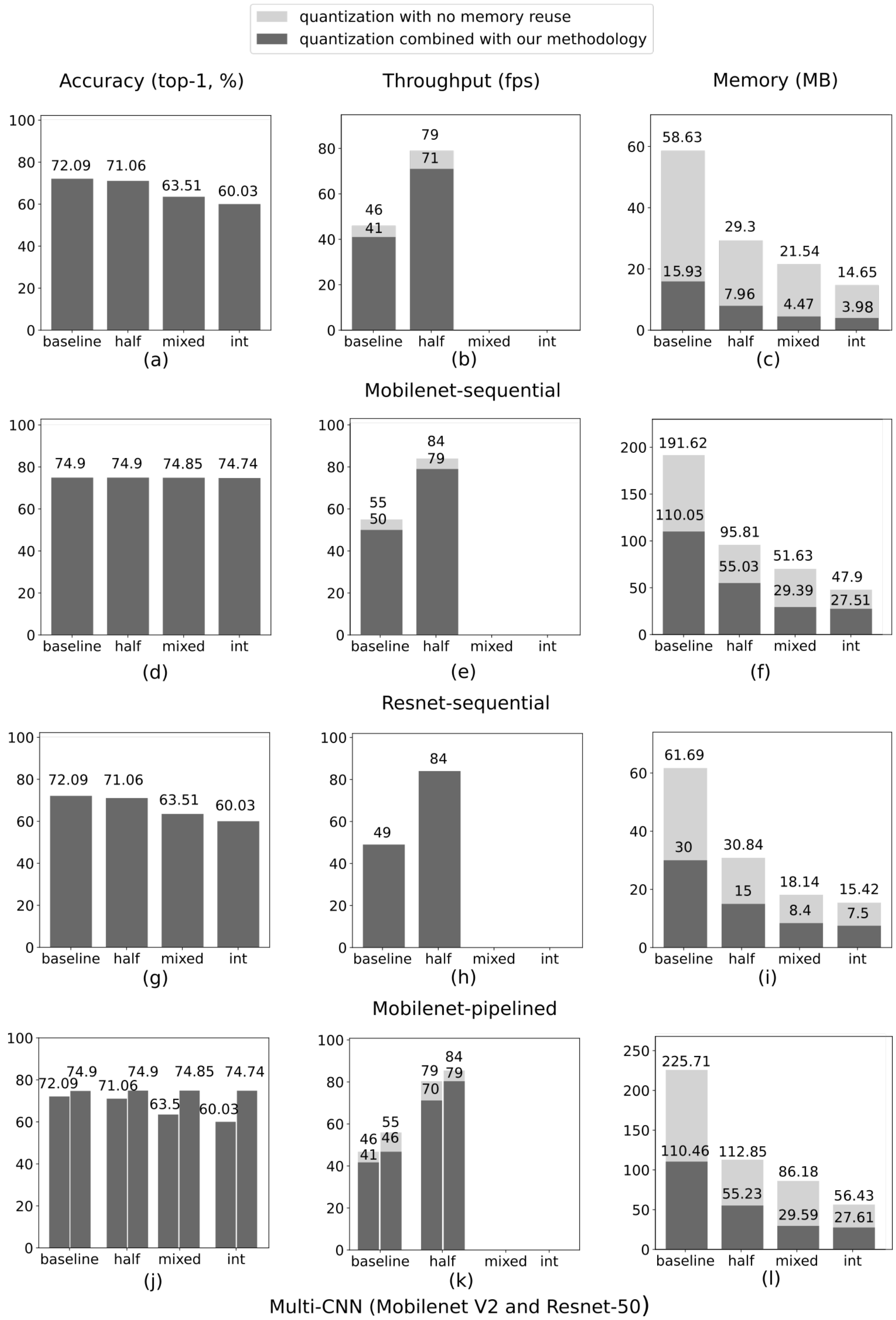


Fig. 5. Experimental results

For example, Figure 5(c) shows that our methodology reduces 3.7 times the memory cost of Mobilenet V2 CNN with half-quantization. Analogously, Figure 5(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 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, 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.*

9 RELATED WORK

The most common CNN memory reduction methodologies, namely pruning and quantization, reviewed in surveys [6, 7, 11, 30], reduce the memory cost of CNN-based applications by reducing the number or size of CNN parameters (weights and biases) [3]. However, at high CNN memory reduction rates these approaches decrease the CNN accuracy, whereas high accuracy is very important for many CNN-based applications [3]. In contrast, our memory reduction approach does not change the CNN model parameters and therefore does not decrease the CNN accuracy.

The knowledge distillation approaches, reviewed in surveys [7, 30], try to replace an initial CNN in a CNN-based application by an alternative CNN with the same functionality but smaller size. However, these approaches involve CNN training from scratch and do not guarantee that the accuracy of the initial CNN can be preserved. In contrast, our memory reduction approach is a general systematic approach which always guarantees preservation of the CNN accuracy.

The CNN buffers reuse methodologies, such as the methodology proposed in [23], and the methodologies reviewed in [15], 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 [26, 27, 29]. Thus, the methodologies in [15, 23] are not suitable for multi-CNN applications. Moreover, these methodologies do not account for concurrent execution of CNN layers. Therefore, they are not applicable to CNN-based applications, exploiting task-level (pipeline) parallelism [18, 31], 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 [17] 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 [17], our proposed methodology involves a balanced memory-throughput trade-off in a CNN-based application, and therefore does not involve unnecessary decrease of CNN throughput.

The CNN layers fusion methodologies, such as the methodologies [4, 21] and the methodologies adopted by the Deep Learning (DL) frameworks, such as the TensorRT DL framework [19] or the PyTorch DL framework [22], 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 8) we implicitly use the CNN layers fusion by implementing the CNNs with the TensorRT DL framework [19], which has built-in CNN layers fusion.

CONCLUSIONS

We propose a memory-throughput trade-off methodology for CNN-based applications at the edge. 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 pipeline parallelism, reuse of memory among different CNNs, and a memory-throughput trade-off balancing mechanism. Thus, our methodology offers a balanced memory-throughput trade-off for a wide range of CNN-based applications, including CNN-based applications exploiting task-level (pipeline) parallelism and multi-CNN applications. The evaluation results show that our methodology 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. Additionally, our evaluation results show that our methodology can be efficiently combined with orthogonal memory reduction methodologies such as quantization 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 at high CNN memory reduction rates, is easily compensated by the CNN throughput increase, introduced by the quantization.

ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon 2020 Research and Innovation program under grant agreement No. 780788.

REFERENCES

[1] Martin Abadi et al. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283.

[2] Martin Abadi, Michael Isard, and Derek G. Murray. 2017. A Computational Model for TensorFlow: An Introduction. In *1st ACM SIGPLAN International Workshop on Machine Learning and Programming (MAPL)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3088525.3088527

[3] Md. Zahangir Alom et al. 2018. The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches. *CoRR* abs/1803.01164 (2018). arXiv:1803.01164 http://arxiv.org/abs/1803.01164

[4] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783725

[5] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jan A. Peperstraete. 1996. Cyclo-Static Dataflow. *IEEE Transactions on Signal Processing* 44, 2 (1996), 397–408. https://doi.org/10.1109/78.485935

[6] Davis W. Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John V. Guttag. 2020. What is the State of Neural Network Pruning?. In *Proceedings of Machine Learning and Systems (MLSys)*.

[7] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2018. A Survey of Model Compression and Acceleration for Deep Neural Networks. *IEEE Signal Processing Magazine* 35 (2018), 126–136.

[8] Vadim Demichev, Christoph B. Messner, Spyros I. Vernardis, Kathryn S. Lilley, and Markus Ralser. 2020. DIA-NN: neural networks and interference correction enable deep proteome coverage in high throughput. *Nature Methods* 17, 1 (2020), 41–44. https://doi.org/10.1038/s41592-019-0638-x

[9] Jia Deng et al. 2009. Imagenet: A large-scale hierarchical image database. In *CVPR*. IEEE, 248–255.

[10] S. Devi, P. Malarvezhi, R. Dayana, and K. Vadivukkarasi. 2020. A Comprehensive Survey on Autonomous Driving Cars: A Perspective View. *Wirel. Pers. Commun.* 114, 3 (2020), 2121–2133. https://doi.org/10.1007/s11277-020-07468-y

[11] Amir Gholami, Sehoon Kim, Dong Zhen, Zhewei Yao, Michael Mahoney, and Kurt Keutzer. 2021. A Survey of Quantization Methods for Efficient Neural Network Inference. *ArXiv* abs/2103.13630 (2021).

[12] Kaiming He et al. 2016. Deep Residual Learning for Image Recognition. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*

[13] Steve Heath. 2002. *Embedded Systems Design 2nd Edition.* Newnes, 321–325.

[14] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. In *Conference on Computer Vision and Pattern Recognition (CVPR).*

[15] Hai Jin, Bo Liu, Wenbin Jiang, Yang Ma, Xuanhua Shi, Bingsheng He, and Shaofeng Zhao. 2018. Layer-Centric Memory Reuse and Data Migration for Extreme-Scale Deep Learning on Many-Core Architectures. *ACM Transactions on Architecture and Code Optimization* 15, 3 (2018). https://doi.org/10.1145/3243904

[16] Di Liu, Hao Kong, Xiangzhong Luo, Weichen Liu, and Ravi Subramaniam. 2020. Bringing AI To Edge: From Deep Learning's Perspective. arXiv:2011.14808 [cs.LG]

[17] Svetlana Minakova and Todor Stefanov. 2020. Buffer Sizes Reduction for Memory-efficient CNN Inference on Mobile and Embedded Devices. In *Euromicro Conference on Digital System Design (DSD).* IEEE Xplore, 133–140. https://doi.org/10.1109/DSD51259.2020.00031

[18] Svetlana Minakova, Erqian Tang, and Todor Stefanov. 2020. Combining Task- and Data-level Parallelism for High-Throughput CNN Inference on Embedded CPUs-GPUs MPSoCs. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS).* Springer International Publishing, Cham, 18–35.

[19] NVIDIA. 2016. TensorRT - high performance neural network inference optimizer and runtime engine for production deployment. https://docs.nvidia.com/deeplearning/sdk/tensorrt-developer-guide/index.html.

[20] NVIDIA. 2017. Jetson TX2. //https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2.

[21] MohammadHossein Olyaiy, Christopher Ng, and Mieszko Lis. 2021. Accelerating DNNs Inference with Predictive Layer Fusion. In *ICS*. Association for Computing Machinery, 291–303. https://doi.org/10.1145/3447818.3460378

[22] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Zachary DeVito Edward Yang, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. *NIPS 2017 Autodiff Workshop* (2017).

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

[24] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).* 4510–4520. https://doi.org/10.1109/CVPR.2018.00474

[25] Kumara Sastry, David Goldberg, and Graham Kendall. 2005. *Genetic Algorithms.* Springer US, Boston, MA, 97–125. https://doi.org/10.1007/0-387-28356-0_4

[26] Benedetta Savelli, Alessandro Bria, Mario Molinara, Claudio Marrocco, and Francesco Tortorella. 2020. A multi-context CNN ensemble for small lesion detection. *Artificial Intelligence in Medicine* 103 (2020). https://doi.org/10.1016/j.artmed.2019.101749

[27] Marco Seeland and Patrick Mader. 2021. Multi-view classification with convolutional neural networks. *PLoS ONE* (2021). https://doi.org/10.1371/journal.pone.0245230

[28] Sander Stuijk, Marc Geilen, and Twan Basten. 2006. SDF3: SDF For Free. In *Sixth International Conference on Application of Concurrency to System Design (ACSD'06).* 276–278.

[29] Ben Taylor, Vicent Sanz Marco, Willy Wolff, Yehia Elkhatib, and Zheng Wang. 2018. Adaptive Selection of Deep Learning Models on Embedded Systems. In *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2018).* Association for Computing Machinery, New York, NY, USA, 31–43. https://doi.org/10.1145/3211332.3211336

[30] Mario P. Vestias. 2019. A Survey of Convolutional Neural Networks on Edge with Reconfigurable Computing. *Algorithms* 12, 8 (2019). https://doi.org/10.3390/a12080154

[31] Siqi Wang, Gayathri Ananthanarayanan, Yifan Zeng, Neeraj Goel, Anuj Pathania, and Tulika Mitra. 2020. High-Throughput CNN Inference on Embedded ARM Big.LITTLE Multicore Processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2020), 2254–2267. https://doi.org/10.1109/TCAD.2019.2944584