Buffer Sizes Reduction for Memory-efficient CNN Inference on Mobile and Embedded Devices

Svetlana Minakova, Todor Stefanov
Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands
s.minakova@liacs.leidenuniv.nl, t.p.stefanov@liacs.leidenuniv.nl

Abstract—**Nowadays, convolutional neural networks (CNNs) are the core of many intelligent systems, including those that run on mobile and embedded devices. However, the execution of computationally demanding and memory-hungry CNNs on resource-limited mobile and embedded devices is quite challenging. One of the main problems, when running CNNs on such devices, is the limited amount of memory available. Thus, reduction of the CNN memory footprint is crucial for the CNN inference on mobile and embedded devices. The CNN memory footprint is determined by the amount of memory required to store CNN parameters (weights and biases) and intermediate data, exchanged between CNN operators. The most common approaches, utilized to reduce the CNN memory footprint, such as pruning and quantization, reduce the memory required to store the CNN parameters. However, these approaches decrease the CNN accuracy. Moreover, with the increasing depth of the state-of-the-art CNNs, the intermediate data exchanged between CNN operators takes even more space than the CNN parameters. Therefore, in this paper, we propose a novel approach, which allows to reduce the memory, required to store intermediate data, exchanged between CNN operators. Unlike pruning and quantization approaches, our proposed approach preserves the CNN accuracy and reduces the CNN memory footprint at the cost of decreasing the CNN throughput. Thus, our approach is orthogonal to the pruning and quantization approaches, and can be combined with these approaches for further CNN memory footprint reduction.**

Keywords-**Convolutional Neural Networks, Dataflow models, CSDF, memory efficiency, buffer size reduction**

I. Introduction

Convolutional Neural Networks (CNNs) are biologically inspired graph computational models, represented as directed acyclic graphs with nodes, associated with CNN operators such as Convolution or Softmax, and edges, representing dependencies between the CNN operators [11]. Due to their ability to handle large, unstructured data, CNNs are widely used to perform tasks such as image classification, object detection, segmentation, and others [13]. The CNN design, i.e., the selection of the CNN graph topology and operators, is a complex task, which does not have strict rules and typically requires substantial knowledge in the field of deep learning (DL) [13]. Once the CNN is designed, it is deployed on hardware platforms to perform training and inference phases. At the training phase, the optimal CNN parameters (weights and biases) are established. At the inference phase, a trained CNN is applied to the actual data and performs the task for which the CNN is designed. Due to the high computational and memory requirements of state-of-the-art CNNs, their training and inference phases are usually performed by high-performance platforms and provided as cloud services. However, there is an increasing number of applications, that require execution of the CNN inference phase on mobile and embedded devices. Examples of such applications are self-driving cars [15], which cannot tolerate high latencies, occurring in cloud services due to communication with a server, or Structural Health Monitoring Systems [5], which require local data storage to ensure the data privacy.

Execution of computationally demanding and memory hungry CNNs on resource-limited mobile and embedded devices is quite challenging. One of the main problems, when running CNNs on such devices, is the limited amount of memory available, especially because not all the memory can be used for the CNN deployment. For example, the basic version of the Raspberry Pie 4 [4] embedded device has 1 GB of memory, while the inference of the state-of-the-art VGG-19 CNN [13], requires about 700 MB of memory for its deployment. If deployed on the Raspberry Pie 4, VGG-19 would occupy almost all memory available on the device. This leaves insufficient memory space for the operating system running on the device, libraries required to execute the CNN inference, storage of the CNN input and output data, etc. Thus, reduction of the CNNs memory footprint is crucial for the execution of the CNN inference on resource-limited mobile and embedded devices.

When a CNN is deployed on a hardware platform, it requires sufficient amounts of memory to store its parameters (weights and biases) and intermediate computational results, exchanged between its operators [11]. A number of approaches have been introduced to reduce the CNN memory footprint. The most common of these approaches, namely pruning and quantization [17], reduce the memory required to store the CNN parameters. However, these approaches decrease the CNN accuracy, while high accuracy is very important for most CNN-based applications [13]. Moreover, with the increasing depth of the state-of-the-art CNNs [13], the intermediate data exchanged between CNN operators takes even more space than the CNN parameters. For example, for the MobileNet [12] and DenseNet [8] CNNs, the intermediate data comprises up to 63% and 80% of the total CNN memory requirement, respectively.

Therefore, in this paper, we propose an approach which reduces the amount of memory, required to store the intermediate data, exchanged between CNN operators. Our proposed approach is based on the ability of CNN operators to process data by parts [13]. It specifies the execution of every CNN operator in several phases, such that at each phase, the CNN operator processes only a part of its input data. To represent the execution of a CNN with phases over several input data parts, we propose and utilize an automated conversion of the CNN into a functionally equivalent Cyclo-Static Dataflow model (CSDF) [7] (see Section V). We use the automatically generated CSDF model and existing embedded system design tools such as SDF3 [14] to find the execution order of the phases of CNN operators which reduces the memory footprint.

As with all existing approaches used to reduce the CNN memory footprint, our approach involves a trade-off [17]. However, while the pruning and quantization approaches reduce the CNN memory footprint at the cost of decreasing the CNN accuracy, our approach preserves the CNN accuracy and reduces the CNN memory footprint at the cost of decreasing the CNN throughput. Thus, our approach is orthogonal to the pruning and quantization approaches, and can be combined with these approaches for further CNN memory footprint reduction. Moreover, based on current trends where the computational power of mobile and embedded devices is rapidly increasing [19], allowing high CNN throughput, while CNNs accuracy is increasing slowly [13], we believe that introducing a small CNN throughput loss is preferred over introducing CNN accuracy loss.

Paper contributions

We propose a novel automated and systematic approach, which reduces the CNN memory footprint at the cost of the CNN throughput decrease. Our proposed approach is based on the ability of CNN operators to process data by parts [13]. It specifies the execution of every CNN operator in several phases, such that: 1) at each phase, the CNN operator processes only a part of its input data; 2) phases are executed in a specific order. To represent the execution of a CNN with phases, we propose and utilize a novel automated approach to convert the CNN into a functionally equivalent Cyclo-Static Dataflow model (CSDF) [7] (see Section V). To find the proper execution order of the phases, we use the automatically generated CSDF model and existing embedded system design tools such as SDF3 [14]. To the best of our knowledge such memory reduction approach has not been proposed before. In Section VI, we provide an experiment where we apply our approach to real-world state-of-the-art CNNs. We show, that our memory reduction approach allows to reduce the CNN memory footprint by 17% to 64% at the cost of 3% to 54% decrease of the CNN throughput.

II. Related work

The most common approaches used to reduce the CNNs memory footprint are pruning, quantization, and Knowledge Distillation, reviewed in surveys [17], [19]. The pruning and quantization approaches reduce the number or size of CNN parameters, thereby, reducing the memory required for the CNN inference. However, these approaches decrease the CNN accuracy, while high accuracy is very important for most CNN-based applications [13]. In contrast, our memory reduction approach does not change the CNN model parameters, and therefore, does not decrease the CNN accuracy.

The Knowledge Distillation (KD) approaches try to shift knowledge from an initial CNN into another CNN, with smaller size but with the same accuracy. However, KD approaches involve training from scratch and do not guarantee that the accuracy of the initial CNN can be preserved. Moreover, KD approaches can only be applied to CNNs designed to perform classification [17], while many CNNs are designed to perform other tasks, such as object detection or segmentation [13]. In contrast, our memory reduction approach is a general systematic approach, which always guarantees preservation of the CNN accuracy, and is not limited to CNNs designed to perform classification tasks.

The dynamic memory allocation approach, utilized in the TensorFlow DL framework [10], allocates memory between the CNN operators at run-time as soon as this memory is needed. This approach allows to significantly reduce the total memory cost of the CNN inference. However, the dynamic memory allocation leads to large performance overheads. In contrast, in our approach, the memory required to store intermediate data exchanged between CNN operators is allocated before the CNN inference execution and does not involve memory allocation overheads at the CNN inference run-time.

The approaches, proposed in [18], reduce the CNN memory footprint by sharing memory between CNN operators, executed at different computational steps. However, these approaches are not very efficient for CNNs with residual connections, such as ResNets [9] and DenseNets [8], because the intermediate data associated with residual connections has to be stored for many computational steps. In contrast, our approach effectively reduces the memory footprint for both residual and non-residual connections, and therefore is applicable to a wider range of state-of-the-art CNNs.

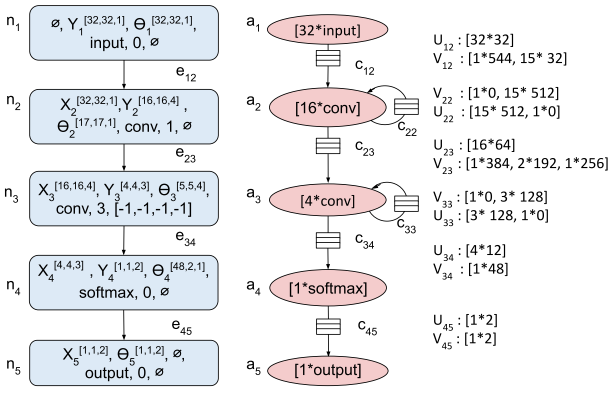
III. Preliminaries

In this section, we provide a brief overview of the CNN and the CSDF computational models, and the CNN inference on mobile and embedded devices. This overview is essential for understanding the proposed memory reduction approach.

A. CNN computational model

A Convolutional Neural Network (CNN) is a computational model [11], commonly represented as a directed acyclic computational graph $CNN(N, E)$ with a set of nodes N, also called layers, and a set of edges E. An example of a CNN model $CNN(N, E)$ is given in Figure 1(a), where $N = \{n_1, n_2, n_3, n_4, n_5\}$, $E = \{e_{12}, e_{23}, e_{34}, e_{45}\}$.

Each layer $n_i \in N$ accepts input data X_i and provides output data Y_i. To obtain the output data Y_i from the input data X_i, layer n_i moves along X_i with sliding window Θ_i



(a) CNN model (b) CSDF model

Fig. 1: CNN and CSDF computational models

and stride s_i, applying operator op_i to an area of X_i, covered by Θ_i. The areas, covered by Θ_i can overlap. If input data X_i cannot be covered by sliding window Θ_i integer number of times, layer n_i crops or extends X_i with $padding$ [13] pad_i and processes cropped/extended input data X_i', which can be covered by sliding window Θ_i integer number of times. The layers input and output data is stored in multidimensional arrays, called $tensors$. In this paper, each tensor T has the format $T^{[H^T,W^T,C^T]}$, where H^T is the tensor height, W^T is the tensor width, C^T is the number of channels. We define a layer as a tuple $n_i = (X_i, Y_i, \Theta_i, op_i, s_i, pad_i)$, where

- $X_i^{[H^{X_i},W^{X_i},C^{X_i}]}$ is the input data of n_i;
- $Y_i^{[H^{Y_i},W^{Y_i},C^{Y_i}]}$ is the output data of n_i;
- $\Theta_i^{[H^{\Theta_i},W^{\Theta_i},C^{\Theta_i}]}$ is the sliding window of n_i, $C_i^{\Theta} = C_i^{X}$;
- s_i is the stride, with which n_i moves over X_i;
- op_i is the operator of n_i;
- pad_i is the padding of n_i;

Where pad_i is an array of four integer elements, $pad_i[k], k \in [0,3]$, and for every $n_i \in N$ $H^{X_i'} = pad_i[1] + H^{X_i} + pad_i[3]$ and $W^{X_i'} = pad_i[0] + W^{X_i} + pad_i[2]$; $pad_i = [0,0,0,0]$, if not specified otherwise. The performed operator op_i and the relationships between the layer attributes are restricted with the layer type. The most common layer types [13] and their operators are listed in Column 1 in Table I. Column 2 in Table I lists the relationships between the attributes of the layers.

TABLE I: Most common CNN layer types

type (op_i)	relationships between attributes
convolutional ($conv$)	$H^{\Theta_i} \le H^{X_i}$; $W^{\Theta_i} \le W^{X_i}$;
pooling({$maxpool$, $avgpool$, $globalmaxpool$, $globalavgpool$})	$H^{\Theta_i} \le H^{X_i}$; $W^{\Theta_i} \le W^{X_i}$; typically $H^{\Theta_i} = W^{\Theta_i} = s_i$;
nonlinear/activation ($relu, thn, sigm$)	$H^{\Theta_i} = W^{\Theta_i} = s_i = 1$
data ($input, output$)	
FC ($dot, loss$)	$H^{\Theta_i} = H^{X_i}$; $W^{\Theta_i} = W^{X_i}$; $s_i = 1$
normalization (BN, LRN)	

An example of layer $n_3 = (X_3^{[16,16,4]}, Y_3^{[4,4,3]}, \Theta_3^{[5,5,4]}, conv, 3, [-1,-1,-1,-1])$ is given in Figure 1(a). Layer n_3 is a convolutional layer, which performs operator $conv$ with a sliding window $\Theta_3^{[5,5,4]}$, moving over the input data $X_3^{[16,16,4]}$ with stride $s_3 = 3$. Padding pad_3 crops input data $X_3^{[16,16,4]}$ to $X_3'^{[-1+16+(-1),-1+16+(-1),4]} = X_3'^{[14,14,4]}$.

Each CNN edge $e_{ij} \in E$, represents a data dependency between layers n_i and n_j, such that $Y_i \subseteq X_j$. We define a CNN edge as a tuple $e_{ij} = (n_i, n_j)$. An example of a CNN edge $e_{12} = (n_1, n_2)$ is given in Figure 1(a). Edge e_{12} represents the data dependency between layers n_1 and n_2, such that $Y_1 = X_2$.

B. CSDF model of computation

The CSDF model [7] is a well-known dataflow model of computation, widely used for model-based design in the embedded systems community. An application modeled as a CSDF is a directed graph $G(A, C)$ with set of nodes A, also called actors, communicating with each other through a set of communication FIFO channels C. Figure 1(b) shows an example of a CSDF graph $G(A, C)$, where $A = \{a_1, a_2, a_3, a_4, a_5\}$ and $C = \{c_{12}, c_{22}, c_{23}, c_{33}, c_{34}, c_{45}\}$. Every actor $a_i \in A$ represents a certain functionality of the application, modeled as a CSDF graph, and performs an execution sequence $F_i = [f_i(1), f_i(2), \cdots, f_i(P_i)]$ of length P_i, where $p \in [1, P_i]$ is called a phase of actor a_i. At every phase actor a_i executes function $f_i(((p-1) mod P_i) + 1)$. An example of CSDF actor a_3 is shown in Figure 1(b). Actor a_3 performs execution sequence $F_3 = [conv, conv, conv, conv]$, shortly written as $[4 * conv]$, meaning actor a_3 has $P_3 = 4$ phases and performs function $f_3(p) = conv$ at each of its phases $p \in [1, 4]$.

Every FIFO communication channel $c_{ij} \in C$ represents data dependency and transfers data in tokens between its source actor a_i and its sink actor a_j. Every $c_{ij} \in C$ has production sequence U_{ij} and consumption sequence V_{ij}. Production sequence $U_{ij} : [u_{ij}(1), u_{ij}(2), \cdots, u_{ij}(P_i)]$ of length P_i specifies the production of data tokens into channel c_{ij} by its source actor a_i. Analogously, consumption sequence $V_{ij} : [v_{ij}(1), v_{ij}(2), \cdots, v_{ij}(P_j)]$ of length P_j specifies the consumption of data tokens from channel c_{ij} by its sink actor a_j. An example of communication channel c_{33} is shown in Figure 1(b). For communication channel c_{33}, actor a_3 is a source and a sink actor. The production sequence $U_{33} : [128, 128, 128, 0]$, shortly written as $U_{33} : [3 * 128, 1 * 0]$ specifies, that during the phases $p \in [1, 3]$ actor a_3 produces 128 tokens to channel c_{33}, and at phase $p = 4$ actor a_3 produces 0 tokens to channel c_{33}. Analogously, the consumption sequence $V_{33} : [1*0, 3*128]$, specifies that during phase $p = 1$ actor a_3 consumes 0 tokens from channel c_{33}, and during phases $p \in [2, 4]$ a_3 consumes 128 tokens from channel c_{33}.

C. Memory and throughput of CNN inference on mobile and embedded devices

The CNN memory footprint m is computed as:

$$m = m_{par} + m_{buf} \tag{1}$$

where m_{par} is the memory, required to store CNN parameters, typically computed as the total number of the CNN model parameters, multiplied by the size of one parameter; m_{buf} is the memory, required to store CNN intermediate

TABLE II: Execution of CNN inference with phases

Ex.	Layer phases					Phases execution order	Buffer sizes (Bytes)					Thr. (fps)
	n_1	n_2	n_3	n_4	n_5		b_{12}	b_{23}	b_{34}	b_{45}	Total	
Ex1	$X_{1k} = \emptyset$, $Y_{1k}^{[32,32,1]}$ $\phi_1 = 1$	$X_{2k}^{[32,32,1]}$, $Y_{2k}^{[16,16,4]}$ $\phi_2 = 1$	$X_{3k}^{[16,16,4]}$, $Y_{3k}^{[4,4,3]}$ $\phi_3 = 1$	$X_{4k}^{[4,4,3]}$, $Y_{4k}^{[1,1,2]}$ $\phi_4 = 1$	$X_{5k}^{[1,1,2]}$, $Y_{5k} = \emptyset$ $\phi_5 = 1$	$n_{11}, n_{21}, n_{31}, n_{41}, n_{51}$	1024	1024	48	2	2098	334
Ex2	$X_{1k} = \emptyset$, $Y_{1k}^{[24,32,1]}$ $\phi_1 = 2$	$X_{2k}^{[24,32,1]}$, $Y_{2k}^{[8,16,4]}$ $\phi_2 = 2$	$X_{3k}^{[16,16,4]}$, $Y_{3k}^{[4,4,3]}$ $\phi_3 = 1$	$X_{4k}^{[4,4,3]}$, $Y_{4k}^{[1,1,2]}$ $\phi_4 = 1$	$X_{5k}^{[1,1,2]}$, $Y_{5k} = \emptyset$ $\phi_5 = 1$	n_{11}, n_{21}, n_{12}, n_{22}, n_{31}, n_{41}, n_{51}	768	1024	48	2	1842	333
Ex3	$X_{1k} = \emptyset$, $Y_{1k}^{[1,32,1]}$ $\phi_1 = 32$	$X_{2k}^{[17,32,1]}$, $Y_{2k}^{[1,16,4]}$ $\phi_2 = 16$	$X_{3k}^{[16,16,4]}$, $Y_{3k}^{[4,4,3]}$ $\phi_3 = 1$	$X_{4k}^{[4,4,3]}$, $Y_{4k}^{[1,1,2]}$ $\phi_4 = 1$	$X_{5k}^{[1,1,2]}$, $Y_{5k} = \emptyset$ $\phi_5 = 1$	$n_{1(1-17)}$, n_{21}, $[n_{1(18-21)}, n_{2(2-16)}]$, n_{31}, n_{41}, n_{51}	544	1024	48	2	1618	310
Ex4	$X_{1k} = \emptyset$, $Y_{1k}^{[1,32,1]}$ $\phi_1 = 32$	$X_{2k}^{[17,32,1]}$, $Y_{2k}^{[1,16,4]}$ $\phi_2 = 16$	$X_{31}^{[6,16,4]}$, $X_{32}^{[5,16,4]}$, $X_{33}^{[5,16,4]}$, $X_{34}^{[6,16,4]}$, $Y_{3k}^{[1,4,3]}$ $\phi_3 = 4$	$X_{4k}^{[4,4,3]}$, $Y_{4k}^{[1,1,2]}$ $\phi_4 = 1$	$X_{5k}^{[1,1,2]}$, $Y_{5k} = \emptyset$ $\phi_5 = 1$	$n_{1(1-17)}$, n_{21}, $[n_{1(18-22)}, n_{2(2-6)}]$, n_{31}, $[n_{1(23-25)}, n_{2(7-9)}]$, n_{32}, $[n_{1(26-28)}, n_{2(10-12)}]$, n_{33}, $[n_{1(29-32)}, n_{2(13-16)}]$, n_{34}, n_{41}, n_{51}	544	384	48	2	978	308

computational results. The CNN intermediate computational results are stored in CNN buffers [18]. Every CNN buffer b_{ij} is associated with CNN edge e_{ij} and stores data Y_i, exchanged between CNN layers n_i and n_j during CNN inference. The size of buffer b_{ij} is computed as the number of elements in data Y_i multiplied by the size of one element in Y_i. Allocation of the memory for CNN parameters and buffers and the access to this memory during the CNN execution are determined by the platform memory structure [16].

The CNN throughput is typically measured in frames per second (fps) and computed as the number of CNN input frames [17], divided by the time, required to perform the CNN inference for all the input frames. During the CNN inference, every CNN layer is executed on processors, such as CPUs, GPUs and/or FPGAs [17], available in the platform. If a platform has GPUs or FPGAs, computations within the layer are represented as one or multiple kernels [17], and offloaded on the GPUs and FPGAs by the CPUs. Otherwise, these computations are performed on the CPUs. The time τ_i, required to execute CNN layer n_i on a mobile/embedded device is computed as:

$$\tau_i = \tau_{access_i} + \tau_{kernel_i} + \tau_{op_i} \tag{2}$$

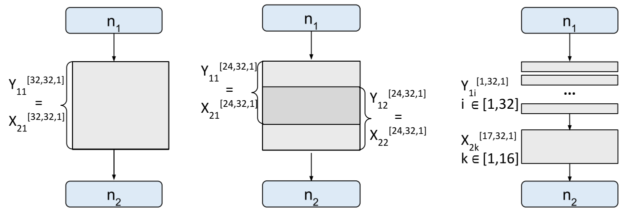
where τ_{access_i} is the time, required to access the CNN parameters and intermediate computational results, stored in the device memory. This time is negligible for CPU-based devices or devices with unified memory structure [16]; τ_{kernel_i} is the time, required to launch kernels on devices with GPUs or FPGAs; τ_{op_i} is the time, required to perform the CNN operator on a processor in the platform where the processor could be CPU or GPU or FPGA.

IV. Motivational example

CNN layers do not process their input data at once. As explained in Section III-A, CNN layers move along the input data with a sliding window and apply operators to input data parts. In this section, we show how processing input data by parts can be utilized to reduce the CNN memory footprint. We define the processing of an input data part by a CNN layer as a phase. If a layer has one phase, it processes its input data as one part. If a layer has two phases, it processes its input data in two parts, etc.

In Table II, we give four examples (Ex1, Ex2, Ex3, Ex4) of inference of the CNN, shown in Figure 1(a) and explained in Section III-A. In each of these examples the CNN inference is executed on a platform with a few CPUs and one GPU, and every CNN layer is executed in one or several phases. For every layer n_i, Columns 2 to 6 of Table II list: 1) the number of phases ϕ_i; 2) part X_{ik} of the input data X_i, processed by layer n_i at its k-th phase, $k \in [1, \phi_i]$; 3) part Y_{ik} of the output data Y_i, produced by layer n_i at its k-th phase, $k \in [1, \phi_i]$. All phases are executed in a specific order, given in Column 7, where n_{ik} denotes the execution of the k-th phase of layer n_i. The execution order ensures functional equivalence of all examples, given in Table II, and allows to reduce the CNN buffer sizes as explained below. Columns 8 to 12 in Table II show the sizes of the CNN buffers b_{ij}, explained in Section III-C. For simplicity of these examples, the buffer sizes are computed with the assumption that one element in any CNN data tensor requires 1 byte of memory for its storage. Column 13 in Table II shows the CNN inference throughput, explained in Section III-C.

Example Ex1, given in Row 3 of Table II, describes the CNN inference typically performed by the state-of-the-art DL frameworks, such as TensorFlow, Keras, Caffe2, and other [6]. In Ex1, every layer has one phase. The CNN inference is performed in 5 computational steps. At every i-th computational step, $i \in [1, 5]$, the single phase of layer n_i is executed. During its single phase, layer n_i processes its whole input data. Figure 2(a) shows how layer n_2 processes its input data in Ex1. Layer n_2 processes its whole input data X_2 as a single data part $X_{21} = X_2$. Data X_{21} is provided to layer n_2 by layer n_1 and stored in buffer b_{12}. To store


n1
Y11[32,32,1] = X21[32,32,1]
n2
n1
Y11[24,32,1] = X21[24,32,1]
Y12[24,32,1] = X22[24,32,1]
n2
n1
Y1i[1,32,1] i ∈ [1,32]
...
X2k[17,32,1] k ∈ [1,16]
n2

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

Fig. 2: Input data processing by layer n_2

data $X_{21}^{[32,32,1]}$ buffer b_{12} occupies 32 * 32 * 1 = 1024 bytes of memory.

Example Ex2, given in Row 4 of Table II, shows how processing data by parts, combined with specific execution order of the phases, allows to reduce the CNN buffer sizes at the cost of decreasing the CNN throughput. In Ex2, CNN layer n_2 processes its input data X_2 in two overlapping parts, X_{21} and X_{22}, as shown in Figure 2(b). Data parts X_{21} and X_{22} are provided to layer n_2 by layer n_1 and stored in buffer b_{12} during the CNN inference. The CNN inference is performed in 7 computational steps. At step 1, phase n_{11} is executed and data $Y_{11} = X_{21}$ is produced in b_{12}. At step 2, phase n_{21} is executed and data X_{21} is processed by layer n_2. After being processed, data X_{21} is not needed anymore and is removed from b_{12}. At step 3, phase n_{12} is executed and data $Y_{12} = X_{22}$ is produced in b_{12}. At step 4, phase n_{21} is executed and data X_{22} is processed by layer n_2. Steps 1 to 4 in Ex2 are functionally equivalent to steps 1 to 2 in Ex1. However, in Ex2 at every computational step, buffer b_{12} has to store only a part of the input data ($X_{21}^{[24,32,1]}$ for steps 1 to 2 and $X_{22}^{[24,32,1]}$ for steps 3 to 4, respectively). Therefore, in Ex2, b_{12} occupies 24 * 32 * 1 = 786 bytes of memory, instead of 1024 bytes, as in Ex1. Compared to Ex1, Ex2 reduces the total buffer sizes by 12% at the cost of only 0.3% throughput decrease due to the increased number of CNN computational steps in Ex2, compared to Ex1.

Example Ex3, given in Row 5 of Table II, demonstrates one more way of executing layers n_1 and n_2 with phases, shown in Figure 2(c). In Ex3, layer n_1 has 32 phases and layer n_2 has 16 phases. The CNN inference is performed in 51 computational step. During the first 17 steps, phases n_{11}, n_{12}, ..., n_{117}, shortly written as $n_{1(1-17)}$, are executed. At every phase, layer n_1 produces data $Y_{1k}^{[1,32,1]} \subset X_{21}$ in buffer b_{12}, until sufficient data $X_{21}^{[17,32,1]}$ is accumulated. Then, at step 18, phase n_{21} is executed. To execute phase n_{22}, data $X_{22}^{[17,32,1]}$ should be accumulated in b_{12}. However, some of this data is already in b_{12}. As explained in Section III-A, data between subsequent execution steps of layer n_2 is overlapping. If the overlapping part is stored in buffer b_{12}, only new (non-overlapping) data should be produced in b_{12} to enable the execution of phase n_{22}. This new data can be produced by execution of one phase of layer n_1. Thus, phases 18-32 of layer n_1 and phases 2-16 of layer n_2 are executed in order $[n_{1(18-32)}, n_{2(2-16)}]$, meaning, that a phase of layer n_1 is followed by a phase of layer n_2, e.g., phase n_{118} is followed by phase n_{22}, and this pattern repeats, until all phases of layers n_1 and n_2 are executed. The maximum amount of data, stored between layers n_1 and n_2 per computational step corresponds to data part $X_{2k}^{[17,32,1]}$, accumulated in b_{12}. Thus, in Ex3, buffer b_{12} occupies 17 * 32 * 1 = 544 bytes of memory. Compared to Ex1, Ex3 reduces the total buffer sizes by 23% at the cost of 7% throughput decrease.

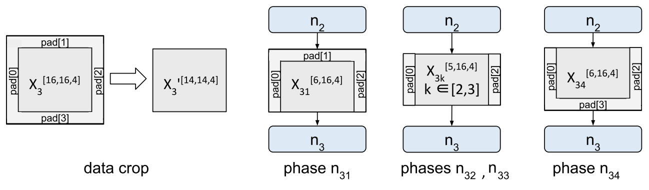


Fig. 3: Input data processing by layer n_3, Ex4

Example Ex4, given in Row 6 of Table II, demonstrates how several Convolutional layers in one CNN can be executed with phases, and how data padding is processed with phases. In Ex4, the CNN inference is executed in 54 computational steps. Layers n_1 and n_2 have 32 and 16 phases, respectively, as in Ex3. Additionally, layer n_3 has 4 phases, i.e., processes its input data in four parts. As explained in Section III-A, layer n_3 has padding pad_3, which crops its input data. With data processing by parts, the data crop is also performed by parts, as shown in Figure 3. At phases n_{31} and n_{34}, layer n_3 accepts data $X_{3k}^{[6,16,4]}$ and crops it to data $X_{3k}'^{[5,14,4]}$. At phases n_{32} and n_{33}, it accepts data $X_{3k}^{[5,16,4]}$ and crops it to data $X_{3k}'^{[5,14,4]}$. The maximum amount of data to be stored in b_{23} is $X_{3k}^{[6,16,4]}$. Thus, buffer b_{23} occupies 6 * 16 * 4 = 384 bytes of memory. Compared to Ex1, Ex4 reduces the total buffer sizes by 53% at the cost of 12.7% throughput decrease. As can be seen from Column 12 of Table II, Ex4 is the most memory-efficient example among all presented examples.

The examples, provided in this section, demonstrate that there are many possible ways to execute the CNN inference with phases. Obtaining the most memory-efficient way is not trivial even for our small example CNN, shown in Figure 1(a), let alone for real-world state-of-the-art CNNs that are much larger and much more complex. Therefore, a systematic and automated approach for finding the CNN inference execution with phases, which ensures minimum buffer sizes, is required. In this paper, we propose such an approach. Our approach consists of two steps. In the first step, presented in Section V, we automatically convert a CNN model into a functionally equivalent CSDF model because the CNN model does not have means for explicit specification of the CNN execution with phases, whereas the CSDF model has such means. By doing this automated conversion, we specify the execution of every CNN layer in one or several phases, such that at each phase, a minimum part of the layer input data is processed, thereby ensuring minimum buffer sizes to exchange data between the layers. Moreover, the CSDF model is accepted as an input by many existing embedded systems design tools for automated performance/memory analysis, transformations and optimizations. So, in the second step, we use such existing embedded design tools, e.g., SDF3 [14], to find a proper execution order of the phases with the minimum buffer sizes, specified in the first step. Thus, in our 2-step approach, we combine processing data by parts with specific execution order of the phases to ensure the CNN inference with minimum buffer sizes.

V. CNN-to-CSDF model conversion

The automated conversion of a CNN into a functionally equivalent CSDF model, utilized in our memory reduction approach, is given in Algorithm 1. Algorithm 1 accepts as input a CNN description, for example in the ONNX format [3], and returns as output CSDF model $G(A, C)$. The first step in our approach is the construction of a CNN model $CNN(N, E)$ from the CNN description - Line 1 of Algorithm 1. It includes the extraction of the topology of the CNN model (CNN layers and edges) and the parameters of the CNN model (e.g. CNN layers operators), which is a straightforward automated action. An example of the constructed model $CNN(N, E)$ is given in Figure 1(a), explained in Section III-A. In Lines 2-21, explained in Section V-A, Algorithm 1 generates the topology of the CSDF model $G(A, C)$. In Lines 22-42, explained in Section V-B, Algorithm 1 derives the production/consumption sequences for every channel in $G(A, C)$. Finally, in Line 43, Algorithm 1 returns $G(A, C)$, which is functionally equivalent to the input $CNN(N, E)$ model. An example of the CNN-to-CSDF conversion is shown in Figure 1, where the CNN model $CNN(N, E)$, shown in Figure 1(a), explained in Section III-A, is converted into the functionally equivalent CSDF model $G(A, C)$, shown in Figure 1(b), explained in Section III-B. The examples, provided in this section for Algorithm 1, are referring to the CNN-to-CSDF conversion, shown in Figure 1.

A. CSDF model topology generation

The CSDF model topology generation is performed in Lines 2-21 of Algorithm 1. In Line 2, Algorithm 1 generates a new CSDF model $G(A, C)$ with an empty set of actors A and an empty set of communication channels C. In Lines 4-15 Algorithm 1 converts every layer n_i of the CNN model $CNN(N, E)$ into a functionally equivalent CSDF actor $a_i \in A$. As explained in Section IV, in our approach, every CNN layer n_i is obtaining the layer output data Y_i from the layer input data X_i by applying operator op_i to the minimum parts of X_i, until all the X_i is processed. To reproduce this functionality, every actor $a_i \in A$ performs execution sequence $F_i = \{f_i(p)\}, p \in [1, P_i]$, where every function $f_i(p) \in F_i$ is specified as $f_i(p) = op_i$ (Lines 13-15 of Algorithm 1). On each phase $p \in [1, P_i]$, actor a_i applies operator op_i to the part of input data $X_{ip}^{[H^{\Theta_i}, W^{X_i}, C^{X_i}]}$ of the layer n_i and produces a part of output data $Y_{ip}^{[1, W^{Y_i}, C^{Y_i}]}$. The number of phases P_i of actor a_i is computed in Lines 9-12 of Algorithm 1. If in layer n_i no overlapping ($s \geq H^{\Theta}$ in Line 9 of Algorithm 1) and no padding ($hpad = 0$ in Line 9 of Algorithm 1) occurs, or the whole X_i should be processed at once ($H^{\Theta} = H^X$ in Line 9 of Algorithm 1), then the number of phases is set to 1 in Line 10 of Algorithm 1.

Algorithm 1: CNN-to-CSDF conversion

```
   Input: CNN description
   Result: $G(A, C)$
1  construct $CNN(N, E)$ from CNN description;
2  $A, C \leftarrow \emptyset$; $G(A, C) \leftarrow$ CSDF model $(A, C)$ ;
3  foreach $n_i \in N$ do
4      $F_i \leftarrow \emptyset$;
5      $a_i \leftarrow$ actor $(F_i)$;
6      $A \leftarrow A + a_i$;
7      $(X, Y, \Theta, op, s, pad) \leftarrow n_i$;
8      $hpad = pad[1] + pad[3]$;
9      if $(s \geq H^{\Theta} \wedge hpad = 0) \vee H^{\Theta} = H^X$ then
10         $P_i = 1$;
11     else
12         $P_i = (H^X + hpad - max(H^{\Theta}, s))/s + 1$;
13     for $p \in [1, P_i]$ do
14         $f_i(p) = op$;
15         $F_i = F_i + f_i(p)$;
16     if $s < H^{\Theta}$ then
17         $c_{ii} \leftarrow channel(a_i, a_i)$;
18         $C \leftarrow C + c_{ii}$;
19 foreach $e_{ij} \in E$ do
20     $c_{ij} \leftarrow channel(a_i, a_j)$;
21     $C \leftarrow C + c_{ij}$;
22 foreach $c_{ij} \in C$ do
23     if $i = j$ then
24         $(X, Y, \Theta, op, s, pad) \leftarrow n_i$;
25         for $p \in [1, P_i]$ do
26             $u_{ij}(p) = \begin{cases} 0 & if\ \ p = P_i \\ (H^{\Theta} - s) * W^X * C^X & otherwise \end{cases}$
27             $v_{ij}(p) = \begin{cases} 0 & if\ \ p = 1 \\ (H^{\Theta} - s) * W^X * C^X & otherwise \end{cases}$
28     else
29         $(X, Y, \Theta, op, s, pad) \leftarrow n_i$;
30         for $p \in [1, P_i]$ do
31             $u_{ij}(p) = W^Y * C^Y$;
32         $(X, Y, \Theta, op, s, pad) \leftarrow n_j$;
33         $v_{ij}(1) = (H^{\Theta} - pad[1]) * W^X * C^X$;
34         $hpad = \begin{cases} pad[1] + pad[3] & if\ \ P_j = 1 \\ pad[3] & otherwise \end{cases}$;
35         if $\nexists c_{jj} \vee P_j = 1$ then
36             for $p \in [2, P_j - 1]$ do
37                 $v_{ij}(p) = H^{\Theta} * W^X * C^X$;
38             $v_{ij}(P_j) = (H^{\Theta} - hpad) * W^X * C^X$;
39         else
40             for $p \in [2, P_j - 1]$ do
41                 $v_{ij}(p) = s * W^X * C^X$;
42             $v_{ij}(P_j) = (s - hpad) * W^X * C^X$;
43 return $G(A, C)$
```

Otherwise, in Line 12 of Algorithm 1, the number of phases is set to the number of steps, required for actor a_i to process input data X_i by parts T_{in}. For example, actor a_3 performs execution sequence $F_3 = [P_3 * op_3] = [4 * conv]$, where $op_3 = conv$ is the operator, performed by layer n_3. As for layer n_3 the condition in Line 9 of Algorithm 1 is not met ($hpad = -2 < 0 \wedge H^{\Theta_3} = 5 < H^{X_3} = 16$), the number of phases P_3 of actor a_3 is computed in Line 12 of Algorithm 1 as $P_3 = (16 + (-2) - max(5, 3))/3 + 1 = 4$.

In Lines 16-18 Algorithm 1 models overlapping data reuse, explained in Ex3 in Section IV. In Line 16, Algorithm 1 checks, if the data overlapping occurs in layer $n_i \in N$. If data overlapping occurs in layer n_i, in Lines 17-18 Algorithm 1 models data overlapping for corresponding actor a_i. Since the CSDF model does not allow internal state specification in actors, the data overlapping/reuse is modeled as self-loop FIFO channels c_{ii}, that store and reuse the overlapping data between subsequent firings of actor a_i. For example, the data overlapping occurs in layer n_3 ($s_3 = 3 < H^{\Theta_3} = 5$). Therefore, in Lines 17-18, Algorithm 1 creates self-loop channel c_{33}, which stores the overlapping/reuse data for actor a_3.

Finally, in Lines 19-21, Algorithm 1 converts every input CNN model edge $e_{ij} \in E$, representing a data dependency between layers $n_i \in N$ and $n_j \in N$, into communication FIFO channel $c_{ij} \in C$, representing data dependency between actors $a_i \in A$ and $a_j \in A$.

B. Production/consumption sequences derivation

The production sequence $U_{ij} = \{u_{ij}(p)\}$, $p \in [1, P_i]$ and the consumption sequence $V_{ij} = \{v_{ij}(p)\}$, $p \in [1, P_j]$ are derived for every channel $c_{ij} \in C$ of CSDF graph $G(A, C)$ in Lines 22-42 of Algorithm 1. For every data reuse channel $c_{ij} \in C, i = j$, storing the overlapping/reuse data between subsequent firings of actor a_i, the elements of the production/consumption sequences are computed in Lines 24-27 of Algorithm 1. Since at the last phase P_i of actor a_i there is no need to produce data to be reused, the last element of the production sequence $u_{ij}(P_i)$ is set to 0 in Line 26 of Algorithm 1. Since at the first phase actor a_i has not yet produced data in the data reuse channel c_{ij}, the first element of the consumption sequence $v_{ij}(1)$ is set to 0 in Line 27 of Algorithm 1. For all other phases of actor a_i the elements of the production/consumption sequences are computed in Lines 26-27 of Algorithm 1 as the number of tokens in a tensor of shape $(H^{\Theta} - s) * W^X * C^X$, reused between the subsequent firings of actor a_i. For example, data reuse channel c_{33} has production sequence $U_{33} : [3{*}128, 1{*}0]$ and consumption sequence $V_{33} : [1 * 0, 3 * 128]$.

For CSDF channels c_{ij}, that are not data reuse channels, i.e. $i \neq j$, the elements of the production/consumption sequences are computed in Lines 29-42 of Algorithm 1. Every element of the production sequences $u_{ij}(p), p \in [1, P_i]$ is computed in Lines 30-31 of Algorithm 1 as the number of elements in tensor Y_{ip}, produced by actor a_i at its phase p. For example, actor a_3 at its every phase $p \in [1, 4]$ produces data $Y_{3p}^{[1,4,3]}$, $p \in [1, 4]$, to channel c_{34}. Therefore, the elements of production rate of channel c_{34} are computed in Lines 30-31 of Algorithm 1 as $u_{34}(p) = 1 * 4 * 3 = 12$.

Every element of the consumption sequences $v_{ij}(p), p \in [1, P_j]$ is computed in Lines 32-42 of Algorithm 1 as the number of elements in data tensor, consumed by actor a_j from non-overlapping channel c_{ij} on the actors phase $p \in [1, P_j]$ in order to produce data Y_{jp}. The first element of the consumption sequences $v_{ij}(1)$ is computed in Line 33 of Algorithm 1. If no padding occurs at the first phase of actor a_j ($pad[1] = 0$ in Line 33 of Algorithm 1), actor a_j consumes from c_{ij} data $X_{jp}^{[H^{\Theta_i}, W^{X_i}, C^{X_i}]}$. If actor a_j crops data at the first phase ($pad[1] < 0$ in Line 33 of Algorithm 1), actor a_j consumes from c_{ij} data X_{jp} and data to be cropped. If actor a_j extends data at the first phase ($pad[1] > 0$ in Line 33 of Algorithm 1), actor a_j consumes from c_{ij} part of data X_{jp}, which is not provided by padding.

The computation of consumption sequence elements $v_{ij}(p), p \in [2, P_j]$ is divided in two different cases, determined by the presence of data overlapping in the channel sink actor a_j, corresponding to layer n_j. If data overlapping is not presented in actor a_j (Lines 35-38 of Algorithm 1), actor a_j consumes all input data from its non-overlapping input channel c_{ij}. If data overlapping/reuse is presented in actor a_j (Lines 39-42 of Algorithm 1), actor a_j consumes from channel c_{ij} only non-overlapping data. The overlapping/reuse data is consumed by actor a_j from its self-loop channel c_{jj}. In total, actor a_j consumes data X_{jp} at phases $p \in [2, P_j - 1]$ (Lines 36-37, 40-41 of Algorithm 1), and all the remaining data at phase $p = P_j$ (Lines 38,42 of Algorithm 1). Consumption of all the remaining data from CSDF channels allows to empty the FIFO buffers and ensure the CSDF model consistency [7].

For example, communication channel c_{23} has consumption sequence $V_{23} : [1{*}384, 2{*}192, 1{*}256]$. The first element of the consumption sequence is computed in Line 33 of Algorithm 1 as $v_{23}(1) = (5 - (-1)) * 16 * 4 = 384$, where $5 * 16 * 4 = 320$ elements are elements of input data tensor $X_{3p}^{[5,16,4]}$, $p \in [1, 4]$, used by actor a_3 to produce data Y_{3p}, and $1{*}16{*}4 = 64$ elements are cropped by actor a_3 according to the padding pad_3. As data overlapping/reuse is presented for a_3 ($\exists c_{33}$), $v_{23}(p), p \in [2, 4]$ are computed in Lines 40-42 of Algorithm 1. At phases $p \in [2, 4 - 1]$ actor a_3 consumes non-overlapping data $3{*}16{*}4 = 192$ from channel c_{23}, i.e., $v_{23}(p) = 192, p \in [2, 3]$. At the last phase actor a_3 consumes the remaining data $(3 - (-1)) * 16 * 4 = 256$ from channel c_{23}, i.e. $v_{23}(4) = 256$.

VI. Evaluation

In this section, we evaluate our memory reduction approach in terms of achieved memory footprint reduction as well as we show the cost of this memory footprint reduction in terms of decreased CNN inference throughput. To this end, we take real-world CNNs from the ONNX models Zoo [2] and obtain their memory footprint and inference throughput when these CNNs are executed using our approach. Then, we compare the obtained memory footprint and inference throughput with the memory footprint and inference throughput obtained when these CNNs are executed as in the state-of-the-art DL frameworks, such as TensorFlow, Keras, Caffe2, and other, reviewed in survey [6]. Recall that, during the CNN inference execution in the state-of-the-art DL frameworks, every CNN layer has one phase at which it processes its whole input data, whereas in our approach every CNN layer has one or several phases and at each phase it processes a minimum part of its input data.

We obtain the CNN memory footprint and inference throughput in two steps. At the first step, we represent the

CNN inference with phases as a CSDF model by performing a CNN-to-CSDF model conversion. In our approach, the CNN-to-CSDF conversion is performed using Algorithm 1, presented in Section V. To represent the CNN inference, performed by the state-of-the-art DL frameworks, and explained in Ex1 in Section IV, as a CSDF model, we perform a one-to-one CNN-to-CSDF conversion, where every CNN layer is converted into a CSDF actor with one phase and every CNN edge is converted into a CSDF channel.

At the second step, we take the CSDF models, obtained at the first step above, and use the SDF3 embedded systems design, analysis and optimization tool [14] to evaluate the CNN memory footprint and inference throughput. The SDF3 tool accepts as an input the CNN inference, modeled as a CSDF graph, and computes: 1) the execution order of the phases in the CSDF model, required to perform the CNN inference with minimum buffer sizes; 2) the guaranteed maximum throughput, achievable by the CNN inference, executed with the minimum buffer sizes.

The evaluation results are given in Table III. Columns 2 and 3 in Table III show the memory footprint of the CNNs, computed by Equation 1, given in Section III-C, such that: 1) the total buffer sizes m_{buf} are computed using the SDF3 tool; 2) memory m_{par} is computed as the total number of the CNN model parameters, provided for every CNN model in the ONNX models Zoo, multiplied by the amount of memory, occupied by a CNN parameter; 3) Every CNN parameter or intermediate data element is stored in floating-point precision and occupies 4 bytes of memory. Column 4 in Table III shows the CNN memory reduction, achieved by our approach in comparison to the approach used in the state-of-the-art DL frameworks [6]. The reduction is computed as $(m_{DL} - m_{our})/m_{DL} * 100\%$, where values m_{DL} and m_{our} are given in Column 2 and Column 3, respectively. Column 4 indicates that our systematic and automated memory reduction approach allows to reduce the CNN memory footprint by 17% to 64%, depending on the CNN. Columns 5 and 6 in Table III show the throughput τ of the CNN inference, when executed as in the state-of-the-art DL frameworks, and as in our approach, respectively. In order to allow the SDF3 tool to evaluate the CNN inference throughput, every actor in the corresponding CSDF model should be annotated with an execution time number per phase. The execution time per phase for a CSDF actor, representing the functionality of a corresponding CNN layer, is computed by Equation 2, given in Section III-C. The times τ_{access_i}, τ_{kernel_i}, and τ_{op_i}, required for Equation 2, are obtained by real measurements, performed on the NVIDIA Jetson TX2 embedded device [1]. Column 7 in Table III shows the CNN throughput decrease, introduced by our approach, when compared to the execution approach used in the state-of-the-art DL frameworks [6]. The decrease is computed as $(\tau_{DL} - \tau_{our})/\tau_{DL} * 100\%$, where values τ_{DL} and τ_{our} are given in Column 5 and Column 6, respectively. Column 7 indicates that our approach introduces 3% to 54% decrease in the CNN throughput, depending on the CNN. Columns 4 and 7 in Table III show that, overall, our memory reduction approach is efficient because the percentage of achieved CNN memory reduction exceeds the percentage of introduced CNN throughput decrease.

TABLE III: Evaluation of our memory reduction approach

CNN	Memory (MB)			Throughput (fps)		
	m_{DL}	m_{our}	reduction	τ_{DL}	τ_{our}	decrease
resnet18	84	52	38%	46	36	22%
googlenet	86	50	41%	60	42	30%
tiny yolo v2	142	66	54%	26	21	22%
inception v1	81	48	41%	66	46	31%
VGG 19	700	579	17%	3.97	3.87	3%
densenet121	327	119	64%	36	19	47%
squeezenet	34	12	64%	272	125	54%

Conclusions

We propose a novel CNN memory footprint reduction approach. Our proposed approach is based on the ability of CNN operators to process data by parts, and allows to reduce the CNN memory footprint at the cost of decreasing the CNN throughput. The key feature of our approach is the execution of CNN operators in phases, achieved by our proposed novel algorithm which converts a CNN to a functionally equivalent CSDF model. The evaluation results show that our memory reduction approach allows to reduce the CNN memory footprint by 17% to 64% at the cost of 3% to 54% decrease of the CNN throughput.

Acknowledgement

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

References

[1] Nvidia jetson embedded mpsoc. //https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2.
[2] Onnx models zoo. https://github.com/onnx/models.
[3] Open neural network exchange format (onnx). https://onnx.ai.
[4] Rpi 4 basic kit. https://www.canakit.com/raspberry-pi-4-basic-kit.html.
[5] A. Monteiro et al. Embedded application of convolutional neural networks on raspberry pi for shm. *Electronics Letters*, 2018.
[6] A. Parvat et al. A survey of deep-learning frameworks. In *ICISC*, 2017.
[7] G. Bilsen et al. Cyclo-static dataflow. In *IEEE TSP*, 1996.
[8] G. Huang et al. Densely connected convolutional networks. In *CVPR*, 2017.
[9] K. He et al. Deep residual learning for image recognition. *CVPR*, 2016.
[10] M. Abadi et al. Tensorflow: Large-scale machine learning on heterogeneous systems. http://tensorflow.org/, 2015.
[11] M. Abadi et. al. A computational model for tensorflow: An introduction. In *MAPL*. ACM, 2017.
[12] M. Sandler et al. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*.
[13] Md. Z. Alom et al. The history began from alexnet: A comprehensive survey on deep learning approaches. *CoRR*, 2018.
[14] S. Stuijk et al. Sdf3: Sdf for free. In *ACSD*, 2006.
[15] T. Do et al. Real-time self-driving car navigation using deep neural network. In *GTSD*, pages 7–12, 2018.
[16] W. Li et al. An evaluation of unified memory technology on nvidia gpus. In *CCGRID*, 2015.
[17] Yu Cheng et al. A survey of model compression and acceleration for deep neural networks. *IEEE Signal Processing Magazine*, 2018.
[18] Y. Pisarchyk and J. Lee. Efficient memory manegement for deep neural net inference. In *MLSys*, 2020.
[19] M. Vestias. A survey of convolutional neural networks on edge with reconfigurable computing. *Algorithms*, 2019.