System-level design for efficient execution of CNNs at the edge
Minakova, S.

Citation

Version: Publisher's Version
License: Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from: https://hdl.handle.net/1887/3487044

Note: To cite this publication please use the final published version (if applicable).
Chapter 4

Methodology for low-memory CNN inference


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

4.1 Problem statement

As mentioned in Chapter 1 in Section 1.2, in order to be deployed and executed on an edge platform, a CNN is required to have low memory footprint. This is because modern edge platforms have limited memory resources. For example,
the basic version of the Raspberry Pi 4 [30] embedded platform has 1 GB of memory. For comparison, deployment and inference of the state-of-the-art VGG-19 CNN [4], requires about 700 MB of memory. If deployed on the Raspberry Pi 4, VGG-19 CNN would occupy almost all memory available on the platform and leave insufficient memory space for the operating system running on the platform, libraries required to execute the CNN inference, storage of the CNN input and output data, etc.

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

### 4.2 Contributions

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

---

1The percentage is given for the CNNs deployed and executed with no memory reduction
processes only a part of its input data; 2) phases are executed in a specific order; 3) the platform memory, allocated to store intermediate computational results of a CNN is reused between the data parts. As the data processing by parts may cause CNN execution time overheads (e.g. CNN layers may require time to switch among the data parts), our methodology may reduce the CNN throughput. However, unlike the most common pruning and quantization methodologies [11,17,31,98], our methodology does not change the number and precision of CNN parameters and therefore does not decrease the CNN accuracy. Thus, our methodology is orthogonal to the pruning and quantization methodologies, and can be combined with these methodologies for further CNN memory footprint reduction. Our proposed methodology, presented in Section 4.5, is our main novel contribution. Other important novel contributions are:

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

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

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

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

4.3 Related Work

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

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

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

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

4.4 Motivational Example

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

In Table 4.1, we give four examples (Ex1, Ex2, Ex3, Ex4) of inference of the CNN, shown in Figure 4.1 and represented as the CNN model, introduced in Section 2.1. In each of these examples the CNN inference is executed on the Jetson TX2 platform introduced in Section 2.3 and shown in Figure 2.4. Every layer of the CNN is executed in one or multiple phases. For every
<table>
<thead>
<tr>
<th>Phases</th>
<th>Buffer sizes (bytes)</th>
<th>Phases execution</th>
<th>Layer phases</th>
<th>Thr.</th>
<th>Phases execution</th>
<th>Layer phases</th>
<th>Thr.</th>
<th>Phases execution</th>
<th>Layer phases</th>
<th>Thr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX2</td>
<td>1024</td>
<td>1842</td>
<td>2</td>
<td>16</td>
<td>1024</td>
<td>1842</td>
<td>2</td>
<td>16</td>
<td>1024</td>
<td>1842</td>
</tr>
<tr>
<td>EX3</td>
<td>1024</td>
<td>1842</td>
<td>2</td>
<td>16</td>
<td>1024</td>
<td>1842</td>
<td>2</td>
<td>16</td>
<td>1024</td>
<td>1842</td>
</tr>
<tr>
<td>EX4</td>
<td>1024</td>
<td>1842</td>
<td>2</td>
<td>16</td>
<td>1024</td>
<td>1842</td>
<td>2</td>
<td>16</td>
<td>1024</td>
<td>1842</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1024</td>
<td>1842</td>
<td>2</td>
<td>16</td>
<td>1024</td>
<td>1842</td>
<td>2</td>
<td>16</td>
<td>1024</td>
<td>1842</td>
</tr>
</tbody>
</table>

Table 4.1: Execution of CNN inference with phases.
4.4. Motivational Example

Figure 4.1: Example CNN

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

Example Ex1, given in Row 3 of Table 4.1, describes the CNN inference typically performed by state-of-the-art DL frameworks, such as TensorFlow, Keras, Caffe2, and other [74]. In Ex1, every layer has one phase. The CNN inference is performed in 5 computational steps. At every \( i \)-th computational step, \( i \in [1,5] \), the single phase of layer \( l_i \) is executed. During its single
Figure 4.2: Input data processing by layer $l_2$

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

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

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

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

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

Example Ex4, given in Row 6 of Table 4.1, demonstrates how several Convolutional layers in one CNN can be executed with phases, and how data padding is processed with phases. In Ex4, the CNN inference is executed in 54 computational steps. Layers $l_1$ and $l_2$ have 32 and 16 phases, respectively, as in Ex3. Additionally, layer $l_3$ has 4 phases, i.e., processes its input data in four parts. As explained in Section 2.1, layer $l_3$ has padding $pad_3$, which crops its input data. With data processing by parts, the data crop is also performed by parts, as shown in Figure 4.3. At phases $l_{31}$ and $l_{34}$, layer $l_3$ accepts data $X_{3k}^{[1,6,16,4]}$ and crops it to data $X_{3k}^{[1,5,14,4]}$. At phases $l_{32}$ and $l_{33}$, it accepts data...
and crops it to data $X'_{3k}$. The maximum amount of data to be stored in $B_2$ is $X'_{3k}$. Thus, buffer $B_2$ occupies $1 * 6 * 16 * 4 = 384$ bytes of memory. Compared to Ex1, Ex4 reduces the total buffer sizes by 53% at the cost of 12.7% throughput decrease. As can be seen from Column 12 of Table 4.1, Ex4 is the most memory-efficient example among all presented examples.

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

4.5 Methodology

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

![Figure 4.4: Methodology for low-memory CNN inference](image-url)
design tools for automated performance/memory analysis, transformations and optimizations. Therefore, to enable for CNN execution with phases and utilization of existing embedded design tools, e.g., SDF3 [91], for the CNN analysis, in Step 2, we automatically convert a CNN model into a functionally equivalent CSDF model. In Step 3, we use the SDF3 tool to analyse the CNN and obtain a set of buffers $B$, used to store the intermediate computational results of the CNN, represented as the CSDF model at Step 2. Every buffer $B_k \in B$ is characterized with minimum size. Together with buffers $B$, the SDF3 tool obtains specific execution order of phases, which enables to correctly execute the CNN inference with buffers $B$. Thus, in our 3-step methodology, we use processing data by parts to ensure the CNN inference with minimum buffer sizes.

4.5.1 Phases derivation

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

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

<table>
<thead>
<tr>
<th>Algorithm 3: Phases derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> CNN($L, E$)</td>
</tr>
<tr>
<td><strong>Result:</strong> Set of phases $\Phi$</td>
</tr>
<tr>
<td>1 $\Phi \leftarrow \emptyset$;</td>
</tr>
<tr>
<td>2 for $l_i \in L$ do</td>
</tr>
<tr>
<td>3 \hspace{1cm} if size of $\Theta_i = size of X_i$ then</td>
</tr>
<tr>
<td>4 \hspace{1cm} $h_{\text{out_min}} = Y_i.h$;</td>
</tr>
<tr>
<td>5 \hspace{1cm} else</td>
</tr>
<tr>
<td>6 \hspace{1cm} $h_{\text{out_min}} = 1$;</td>
</tr>
<tr>
<td>7 \hspace{1cm} $Y_{ik} \leftarrow$ part of $Y_i$ of shape $[Y_i.batch, h_{\text{out_min}}, Y_i.w, Y_i.c]$;</td>
</tr>
<tr>
<td>8 \hspace{1cm} $\Phi_i \leftarrow Y_i.h/Y_{ik}.h$;</td>
</tr>
<tr>
<td>9 \hspace{1cm} $\Phi \leftarrow \Phi + \Phi_i$;</td>
</tr>
<tr>
<td>10 return $\Phi$</td>
</tr>
</tbody>
</table>
part of input and output data of layer $l_i$ is characterized with minimum height, determined by the attributes of layer $l_i$, shown in Table 2.1 and explained in Section 2.1. An example of such layer execution is shown in Figure 4.2(c), explained in Section 4.4, where layer $l_2$ performs $\Phi_2 = 16$ phases. At each phase $k \in [1, 16]$ layer $l_2$ accepts an input data part $X_{2k}$ with minimum height of 17 pixels, and produces an output data part $Y_{2k}$ with height of 1 pixel.

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

### 4.5.2 CNN-to-CSDF model conversion

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

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

**Algorithm 4: CNN-to-CSDF conversion**

```
Input: CNN\((L, E)\), \(\Phi = \{\Phi_1, \Phi_2, ..., \Phi_{\ell}\}\)

Result: \(G(A, C)\)

1. \(A, C \leftarrow \emptyset; G(A, C) \leftarrow \text{CSDF model } (A, C)\)
2. foreach \(l_i \in L\) do
   3. \(F_i \leftarrow \emptyset\)
   4. \(a_i \leftarrow \text{actor } (F_i)\)
   5. \(A \leftarrow A + a_i\)
   6. \(\{\Theta_i, \theta_i, \sigma_i\} \leftarrow \text{attributes of } l_i\) (see Table 2.1);
   7. \(P_i \leftarrow \Phi_i\)
   8. for \(p \in [1, P_i]\) do
      9. \(f_i(p) = \theta_i;\)
      10. \(F_i = F_i + f_i(p)\)
      11. if \(s_i < \Theta_i, h\) then
           12. \(c_{ij} \leftarrow \text{channel}(a_i, a_j)\)
           13. \(C \leftarrow C + c_{ij}\)
   14. foreach \(c_{ij} \in E\) do
      15. \(c_{ij} \leftarrow \text{channel}(a_i, a_j)\)
      16. \(C \leftarrow C + c_{ij}\)
    17. foreach \(c_{ij} \in C\) do
       18. \(\{X_i, Y_i, kh_i, s_i, pad_i\} \leftarrow \text{attributes of } l_i\) (see Table 2.1);
       19. \(\{X_j, Y_j, kh_j, s_j, pad_j\} \leftarrow \text{attributes of } l_j\) (see Table 2.1);
       20. if \(i = j\) then
          21. for \(p \in [1, P_i]\) do
              22. \(u_{ij}(p) = \begin{cases} 0 & \text{if } p = P_i \\ X_i.batch * (\Theta_i, h - s_i) * X_i.w * X_i.ch & \text{otherwise} \end{cases}\)
              23. \(v_{ij}(p) = \begin{cases} 0 & \text{if } p = 1 \\ X_i.batch * (\Theta_i, h - s_i) * X_i.w * X_i.ch & \text{otherwise} \end{cases}\)
          24. else for \(p \in [1, P_i]\) do
              25. \(u_{ij}(p) = Y_i.batch * 1 * Y_i.w * Y_i.ch;\)
              26. \(v_{ij}(1) = X_i.batch * (kh_j - pad_j[1]) * X_i.w * X_i.ch;\)
              27. \(hpad_j = \begin{cases} pad_j[1] + pad_j[3] & \text{if } P_j = 1 \\ pad_j[3] & \text{otherwise} \end{cases};\)
              28. if \(\neg c_{ij} \lor P_j = 1\) then
                  29. for \(p \in [2, P_j - 1]\) do
                      30. \(v_{ij}(p) = X_j.batch * kh_j * X_j.w * X_j.ch;\)
                      31. \(v_{ij}(P_j) = X_j.batch * (kh_j - hpad_j) * X_j.w * X_j.ch;\)
              32. else for \(p \in [2, P_j - 1]\) do
                  33. \(v_{ij}(p) = X_j.batch * s_j * X_j.w * X_j.ch;\)
                  34. \(v_{ij}(P_j) = X_j.batch * (s_j - hpad_j) * X_j.w * X_j.ch;\)
      35. return \(G(A, C)\)
```
CSDF model topology generation

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

The number of phases \( \Phi_i \) of actor \( a_i \) representing layer \( l_i \) is specified in the input set of phases \( \Phi \). For example, actor \( a_3 \) performs execution sequence \( F_3 = [P_3 * op_3] = [4 * conv] \), where \( op_3 = conv \) is the operator, performed by layer \( l_3 \), 4 is the number of phases \( \Phi_3 \), specified for layer \( l_3 \) in the input set of phases \( \Phi \).

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

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

Production/consumption sequences derivation

The production sequence \( U_{ij} = \{u_{ij}(p)\}, p \in [1, P_i] \) and the consumption sequence \( V_{ij} = \{v_{ij}(p)\}, p \in [1, P_j] \) are derived for every channel \( c_{ij} \in C \) of CSDF graph \( G(A, C) \) in Lines 24 to 36 of Algorithm 4. For every data reuse channel \( c_{ij} \in C, i = j \), storing the overlapping/reuse data between subsequent firings of actor \( a_i \), the elements of the production/consumption sequences are computed in Lines 21 to 23 of Algorithm 4. Since at the last phase \( P_i \) of actor \( a_i \) there is no need to produce data to be reused, the last
element of the production sequence \( u_{ij}(P_i) \) is set to 0 in Line 22 of Algorithm 4. Since at the first phase actor \( a_i \) has not yet produced data in the data reuse channel \( c_{ij} \), the first element of the consumption sequence \( v_{ij}(1) \) is set to 0 in Line 23 of Algorithm 4. For all other phases of actor \( a_i \) the elements of the production/consumption sequences are computed as the number of tokens in a tensor of shape \([X_i.b_{\text{batch}}, (\Theta_i - s_i), X_i.w, X_i.ch]\), reused between the subsequent firings of actor \( a_i \). For example, data reuse channel \( c_{33} \) has production sequence \( U_{33} : [3 \times 128, 1 \times 0] \) and consumption sequence \( V_{33} : [1 \times 0, 3 \times 128] \).

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

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

The computation of consumption sequence elements \( v_{ij}(p) \), \( p \in [2, P_j] \) is divided in two different cases, determined by the presence of data overlapping in the channel sink actor \( a_{ij} \), corresponding to layer \( l_j \). If data overlapping is not presented in actor \( a_j \) (Lines 29-32 of Algorithm 4), actor \( a_j \) consumes all input data from its non-overlapping input channel \( c_{ij} \). If data overlapping/reuse is presented in actor \( a_j \) (Lines 34-36 of Algorithm 4), actor \( a_j \) consumes from channel \( c_{ij} \) only non-overlapping data. The overlapping/reuse data is consumed by actor \( a_j \) from its self-loop channel \( c_{jj} \). In total, actor \( a_j \) consumes data \( X_{jp} \) at phases \( p \in [2, P_j - 1] \) (Lines 30-31, 34-35 of Algorithm 4), and all the remaining data at phase \( p = P_j \) (Lines 32, 36 of Algorithm 4). Consumption of all the remaining data from CSDF channels allows to empty the FIFO buffers.
and ensure the CSDF model consistency [10].

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

## 4.6 Experimental Results

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

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

<table>
<thead>
<tr>
<th>CNN</th>
<th>Memory footprint (MB)</th>
<th>Memory reduction (%)</th>
<th>Throughput (fps)</th>
<th>Throughput reduction (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TensorRT ours</td>
<td>TensorRT ours</td>
<td></td>
<td></td>
</tr>
<tr>
<td>resnet18</td>
<td>51.6 49</td>
<td>5</td>
<td>137 121</td>
<td>12</td>
</tr>
<tr>
<td>googlenet</td>
<td>38.7 31</td>
<td>19.8</td>
<td>118 103</td>
<td>13</td>
</tr>
<tr>
<td>tiny yolo v2</td>
<td>77.3 64.3</td>
<td>16.8</td>
<td>131 105</td>
<td>20</td>
</tr>
<tr>
<td>inception v1</td>
<td>38.3 31</td>
<td>19</td>
<td>122 106</td>
<td>13</td>
</tr>
<tr>
<td>VGG 19</td>
<td>594 577</td>
<td>2.8</td>
<td>15 14.7</td>
<td>2</td>
</tr>
<tr>
<td>densenet121</td>
<td>43 40</td>
<td>7.5</td>
<td>62 49</td>
<td>21</td>
</tr>
<tr>
<td>squeezenet</td>
<td>10.4 6.4</td>
<td>38</td>
<td>342 262</td>
<td>23</td>
</tr>
</tbody>
</table>
methodology. This application is automatically generated by the TensorRT DL framework from the input CNN description in .onnx format; 2) a C++ CNN inference with phases application, which implements the CNN inference with phases, derived from the same input CNN by our methodology presented in Section 4.5. To implement this application, we use the TensorRT DL framework as well as a custom code generation component, which offers support of the CNN inference with phases (CSDF) model, unsupported by the TensorRT DL framework. The TensorRT DL framework is used to define CNN operators, while our custom code is used to define the CSDF model.

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

<table>
<thead>
<tr>
<th>CNN</th>
<th>Buffer sizes (MB)</th>
<th>parameters (MB)</th>
<th>Total phases</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TensorRT</td>
<td>ours</td>
<td>TensorRT</td>
</tr>
<tr>
<td>resnet18</td>
<td>4.8</td>
<td>2.2</td>
<td>46.8</td>
</tr>
<tr>
<td>googlenet</td>
<td>10.7</td>
<td>3</td>
<td>28</td>
</tr>
<tr>
<td>tiny yolo v2</td>
<td>14.2</td>
<td>0.8</td>
<td>63.5</td>
</tr>
<tr>
<td>inception v1</td>
<td>10.3</td>
<td>3</td>
<td>28</td>
</tr>
<tr>
<td>VGG 19</td>
<td>19.3</td>
<td>2.3</td>
<td>574.7</td>
</tr>
<tr>
<td>densenet121</td>
<td>10.9</td>
<td>7.9</td>
<td>32.1</td>
</tr>
<tr>
<td>squeezenet</td>
<td>5.1</td>
<td>1.4</td>
<td>5</td>
</tr>
</tbody>
</table>
squeezenet CNN (see Row 9 in Table 4.3) requires 1.4 to 5.1 MegaBytes of memory to store its intermediate computational results and and 5 MegaBytes of memory to store its parameters. Analogously, the VGG 19 CNN (see Row 7 in Table 4.3) requires 2.3 to 19.3 MegaBytes of memory to store its intermediate computational results and 574.5 MegaBytes of memory to store its parameters. In other words, the squeezenet CNN requires similar amount of memory to store its intermediate computational results and its parameters, while the VGG 19 CNN requires much more memory to store its parameters than to store its intermediate computational results. Consequently, our methodology achieves a significant, 38%, memory reduction for the squeezenet CNN and small, 2.8%, memory reduction for the VGG 19 CNN (see Row 7 and Row 9, Column 4 in Table 4.2). Secondly, as explained in Section 4.3, unlike the TensorRT buffers reuse methodology, our methodology reuses data within CNN layers. As shown in Section 4.4, the more phases are performed by layers of a CNN, the more memory is reused within the CNN layers and the more memory reduction can our methodology achieve. Thus, the number of phases performed by the CNN layers affects the memory reduction, achieved by our methodology. The number of phases performed by the CNN layers, when the CNNs are executed with the TensorRT buffers reduction methodology and with our methodology, is shown in Columns 5 and 6 in Table 4.3, respectively. When a CNN is executed with the TensorRT buffers reduction methodology, every layer of the CNN performs one phase. Therefore, the total number of phases performed by the layers of a CNN corresponds to the number of layers in the CNN. When a CNN is executed with our methodology, the total number of phases performed by the CNN layers is computed as $\sum_{\Phi_i \in \Phi} \Phi_i$, where $\Phi$ is a set of phases, derived for the CNN using Algorithm 3 introduced in Section 4.5.1. For example, Row 5, Columns 5 and 6 in Table 4.3 shows that the tiny yolo v2 CNN performs 33 phases when is executed with the TensorRT buffers reduction methodology and 3796 phases when executed with our methodology. We believe that the high, 16.8%, memory reduction, achieved by our methodology for the tiny yolo v2 CNN (see Column 4, Row 5 in Table 4.2) is due to the large amount of memory reuse within the CNN layers that our methodology introduced into the tiny yolo v2 CNN by increasing the total number of CNN phases $3796/33 \approx 115$ times.

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

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