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

Minakova, S.

Chapter 2

Background

In this chapter, we present an overview of concepts essential to understand the contributions of this thesis. In Section 2.1, we present the CNN model used to represent a CNN in this thesis. In Section 2.2, we describe the CNN deployment and inference at the Edge, briefly introduced in Section 1.3 because in this thesis, we study and propose novel methodologies for efficient CNN deployment and inference at the Edge. In Section 2.3, we introduce a typical edge platform used to execute CNNs inference. Namely, we introduce the well-known and state-of-the-art NVIDIA Jetson TX2 platform [71], used to perform experiments in this thesis. In Section 2.4, we explain the task- and data-level parallelism available in a CNN. We exploit the aforementioned types of parallelism to ensure efficient inference of CNNs at the Edge. In Section 2.5, we briefly describe the Cyclo-Static Data Flow (CSDF) [10] and the Synchronous Data Flow (SDF) [57] models of computation, widely used in the Embedded Systems community to represent applications executed at the Edge. Unlike the CNN model, introduced in Section 2.1, the SDF model and the CSDF model explicitly specify the parallelism, available within an application (or a part of an application such as a CNN), and enable for modelling of various manners of application execution. In this thesis, we use the CSDF model and the SDF model to represent an augmented design point (i.e., a CNN, executed in a specific manner) briefly introduced in Section 1.5. Finally, in Section 2.6, we describe the basic concepts of a Genetic Algorithm (GA): a well-known heuristic approach, widely used for finding optimal solutions for complex Design Space Exploration (DSE) problems. Some of the methodologies, presented in this thesis, are based on a GA.

2.1 CNN model

A Convolutional Neural Network (CNN) is commonly represented as a directed acyclic computational graph $CNN(L, E)$ with a set of nodes L, also called layers, and a set of edges E. An example of a CNN model with a set of layers $L = \{l_1, l_2, l_3, l_4, l_5\}$ and a set of edges $E = \{e_{12}, e_{23}, e_{34}, e_{45}\}$ is shown in Figure 2.1.

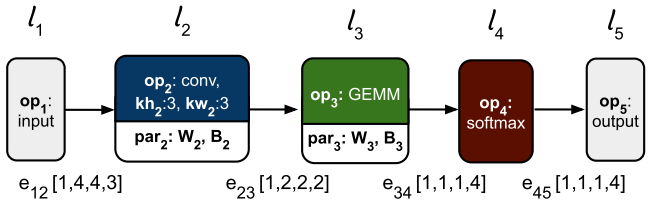


Figure 2.1: *CNN model*

The CNN model specifies transformations over the CNN input data (e.g. an image), that result into the CNN output data (e.g. an image classification result). The transformations are specified by the set of layers L. Edges in the set E specify data dependencies between the layers and determine the flow of data in a CNN. The detailed explanation and formal definition of a layer $l_i \in L$ and an edge $e_{ij} \in E$ of the CNN model are given in Section 2.1.1 and Section 2.1.2, respectively.

2.1.1 Layer in the CNN model

Every layer l_i in the CNN model represents part of the CNN functionality. It accepts as an input some data, produced by other layers, transforms this data using a mathematical operator, and provides output data. Formally, we define layer l_i as a set of attributes, summarized in Table 2.1. Column 1 lists the attributes; Column 2 provides a description of each attribute; Column 3 lists limitations, posed on the attribute by the CNN model; Column 4 shows the *default* value of an attribute, i.e., the value assigned to the attribute which is not defined explicitly. We note that some of the attributes (e.g., attributes I_i and O_i shown in Rows 4 to 5 in Table 2.1) only take the default value. Below, we explain the attributes of layer l_i, summarized in Table 2.1, using as an example layer l_2 shown in Figure 2.1.

Attributes $type_i$ and op_i (Rows 2 to 3 in Table 2.1) specify the type and performed operator of layer l_i, respectively [4]. These attributes determine the main difference between the layers of a CNN. The most common types of

Table 2.1: *Attributes of layer l_i*

attribute	description	limitations	default value
$type_i$	layer type	supported by the CNN model (see Table 2.2)	for known op_i see Table 2.2
op_i	operator	restricted by $type_i$ (see Table 2.2)	-
I_i	input edges	$I_i \subseteq E : \forall e_{ji} \in E, e_{ji} \in I_i$	
O_i	output edges	$O_i \subseteq E : \forall e_{ij} \in E : e_{ij} \in O_i$	
X_i	input data	see Equation 2.1	
Y_i	output data	see Equation 2.2	
Θ_i	sliding window	has smaller or equal size compared to X_i	window of size kh_ix kw_i
kh_i	kernel height	$0 < kh_i \leq X_i.h$; typically $kh_i = kw_i$; $kh_i = X_i.h$ if $type_i \in$*{data,FC}*	$X_i.h$ if $type_i \in$*{data,FC}*, else 1
kw_i	kernel width	$0 < kw_i \leq X_i.w$; typically $kw_i = kh_i$; $kw_i = X_i.w$ if $type_i \in$*{data,FC}*	$X_i.w$ if $type_i \in$*{data,FC}*, else 1
s_i	stride	$s_i = 1$ if $op_i \notin$ *{conv, max pool, average pool}*	1
pad_i	padding	an array of four integer numbers	[0,0,0,0]
par_i	(trainable) parameters	a set of parameters, specific for CNN layer [4]	$\emptyset$

Table 2.2: *Most common CNN layer types and operators*

layer type	operators
convolutional	*conv*
pooling	*(global) max pool, (global) average pool*
activation	*ReLU, thn, sigmoid*
data	*input, output*
fully connected (FC)	*GEMM, MatMUL, dot*
loss	*softmax*
normalization	*BatchNormalization, LRN*
arithmetic	*add*
transformation	*concat*

layers and operators performed by layers of these types are shown in Table 2.2. For example, layer l_2 shown in Figure 2.1 has $type_2 = convolutional$ and performs operator $op_2 = conv$. Operator op_2 performed by layer l_2 is explicitly specified in Figure 2.1, thus the type of layer l_2 is determined using Table 2.2.

Attributes I_i and O_i (Rows 4 to 5 in Table 2.1) specify the input and output edges of layer l_i, respectively. For example, layer l_2 shown in Figure 2.1 has input edges $I_2 = \{e_{12}\}$ and output edges $O_2 = \{e_{23}\}$.

Attributes X_i and Y_i (Rows 6 to 7 in Table 2.1) specify the input and output data of layer l_i, respectively. These attributes always take the default value, computed using Equation 2.1 and Equation 2.2.

$$X_i = \begin{cases} e_{ji}.data : e_{ji} \in I_i & \text{if } |I_i| = 1 \\ \{e_{ji}.data\}, \forall e_{ji} \in I_i & \text{otherwise} \end{cases} \tag{2.1}$$

$$Y_i = \begin{cases} e_{ij}.data : e_{ij} \in O_i & \text{if } |O_i| > 0 \\ \emptyset & \text{otherwise} \end{cases} \tag{2.2}$$

The value of attribute X_i is computed using Equation 2.1, where $e_{ji}.data$ is the data accepted by layer l_i and associated with input edge $e_{ji} \in I_i$ of layer l_i; $|I_i|$ is the total number of input edges of layer l_i. Typically, layer l_i has one input edge, i.e., $|I_i| = 1$. In this case, input data X_i of layer l_i is the data $e_{ji}.data$, associated with the only input edge e_{ji} of layer l_i. For example, layer l_2 shown in Figure 2.1 has one input edge e_{12}, and has input data $X_2 = e_{12}.data$. However, some layers may accept as an input data coming from multiple input edges (e.g., layers performing operator *concat* [4]) or accept no input data (e.g., layers performing the operator *input* [4]). Layers that accept no input data have $|I_i| = 0$ and $X_i = \emptyset$.

Analogously, the value of attribute Y_i is computed using Equation 2.2, where $e_{ij}.data$ is the data produced by layer l_i and associated with output edge $e_{ij} \in O_i$ of layer l_i; $|O_i|$ is the total number of output edges of layer l_i. Typically, layer l_i has at least one output edge and produces data $Y_i \neq \emptyset$, broadcasted to every output edge of layer l_i. For example, layer l_2 shown in Figure 2.1 produces output data $Y_2 = e_{23}.data$ onto its output edge e_{23}. However, some layers (e.g., layers performing the operator *output* [4]) do not produce data. These layers have $|O_i| = 0$ and $Y_i = \emptyset$.

Attributes Θ_i, kh_i, kw_i, s_i, and pad_i (Rows 8 to 12 in Table 2.1) are the *hyper-parameters* of layer l_i [4]. These attributes, obtained during the CNN design, specify how the layer processes its input data. To process its input data X_i, layer l_i moves along X_i with sliding window Θ_i and stride s_i, applying operator op_i to the area of X_i, covered by Θ_i. The sliding window Θ_i has

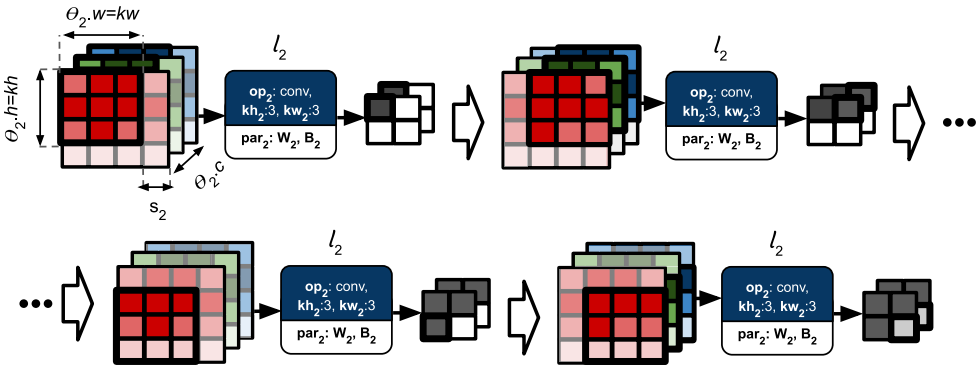


Figure 2.2: *Processing of input data X_2 by layer l_2*

smaller or equal size, compared to X_i. The height and width of window Θ_i are typically equal to the kernel height kh_i and kernel width kw_i of layer l_i, while the number of channels of Θ_i is typically equal to the number of channels of X_i [4]. The areas, covered by Θ_i, can overlap. Figure 2.2 shows an example, where layer l_2 shown in Figure 2.1 processes its input data by four parts, covered by sliding window Θ_2 of size 3 x 3 x 3 pixels, and stride $s_2 = 1$ pixel.

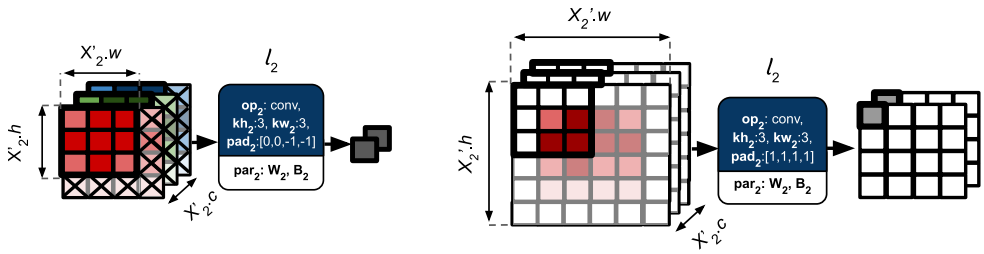
Before processing its input data, layer l_i may crop or extend its input data X_i to data X'_i with *padding* [4]. Typically, this is done to ensure that the input data of layer l_i can be covered by sliding window Θ_i of layer l_i integer number of times [4]. We specify the padding of layer l_i as attribute pad_i (Row 12 in Table 2.1). pad_i is an array of four integer numbers. Elements of pad_i, referred as $pad_i[0]$, $pad_i[1]$, $pad_i[2]$, and $pad_i[3]$, respectively, specify the crop/extension of the height and width of data X_i as given in Equation 2.3 and Equation 2.4, respectively.

$$X'_i.w = pad_i[0] + X_i.w + pad_i[2] \tag{2.3}$$

$$X'_i.h = pad_i[1] + X_i.h + pad_i[3] \tag{2.4}$$

By default, layer l_i has $pad_i = [0,0,0,0]$, which means that layer l_i does not crop or extend its input data X_i before processing. Figure 2.3 shows an example where layer l_2 crops (see Figure 2.3 (a)) and extends (see Figure 2.3 (b)) its input data X_i with padding pad_2 = [0,0,-1,-1] and pad_2 = [1,1,1,1], respectively.

Beside the hyper-parameters, layer l_i has (trainable) parameters such as weights and biases [4], specified as attribute par_i (Row 13 in Table 2.1). As mentioned in Chapter 1, these parameters of layer l_i are obtained during the



(a) *crop* **(b)** *extension*

Figure 2.3: *Padding*

CNN training and are used by operator op_i of layer l_i. For example, layer l_2 has parameters par_2 composed of weights W_2 and biases B_2, used to perform $op_2 = conv$.

2.1.2 Edge in the CNN model

Every edge $e_{ij} \in E$ in the CNN model specifies a data dependency between layers l_i and l_j of a CNN, such that the data produced by layer l_i is accepted as an input by layer l_j. Formally, we define edge e_{ij} as a tuple $(l_i, l_j, data)$, where $data$ is the data produced by layer l_i, accepted by layer l_j, and associated with edge e_{ij}. The data associated with edge e_{ij} is stored in a multidimensional array called tensor [4]. In this thesis, every data tensor has the shape $[batch, h, w, ch]$, where $batch, h, w, ch$ are the batch size [4], the height, the width, and the number of channels of the tensor, respectively. An example of edge $e_{12} = (l_1, l_2, data)$ is shown in Figure 2.1. Edge e_{12} represents the data dependency between layers l_1 and l_2, where layer l_2 accepts as an input the data produced by layer l_1. Edge e_{12} is annotated with shape [1,4,4,3]. This means that the data tensor, exchanged between layers l_1 and l_2, and associated with edge e_{12} has batch size = 1, height and width = 4, and number of channels = 3.

2.2 CNN deployment and inference at the Edge

The CNN inference is a process of applying the CNN to real-world data (e.g., images) and obtaining the CNN output (e.g., results of the input images classification). Nowadays, the CNN inference can be performed on a wide variety of hardware platforms. In this thesis, we concentrate on the CNN inference performed on edge (mobile and embedded) platforms, presented in Section 2.3.

Before the CNN inference can start, the CNN is *deployed* on a target platform, i.e., some memory of the platform is allocated to the CNN. The total amount of memory (in bytes), allocated to a CNN is computed as:

$$m = m_{par} + m_{buf} \tag{2.5}$$

where m_{par} is the memory, required to store the CNN parameters (weights and biases) and computed using Equation 2.6; m_{buf} is the memory, required to store the CNN intermediate computational results and computed using Equation 2.7.

$$m_{par} = \sum_{i \in [1,|L|]} |par_i| * par\_size \tag{2.6}$$

In Equation 2.6, $|par_i|$ is the total number of parameters, associated with layer $l_i \in L$ of the CNN; *par_size* is the size of one parameter in bytes;

$$m_{buf} = \sum_{B_k \in B} B_k.size \tag{2.7}$$

In Equation 2.7, B is a set of *buffers*, i.e., the memory segments, allocated to store the intermediate computational results of a CNN [76]. Every buffer $B_k \in B$ has one or several CNN edges e_{ij} allocated to it. Buffer B_k stores data $e_{ij}.data$, exchanged between CNN layers l_i and l_j during the CNN inference and is characterized with size (in bytes) computed as:

$$B_k.size = max_{e_{ij} \in B_k.edges}\{|e_{ij}.data| * token\_size\} \tag{2.8}$$

In Equation 2.8, $e_{ij} \in B_k.edges$ is an edge, storing data in buffer B_k; $|e_{ij}.data|$ is the total number of data elements (tokens), exchanged through edge e_{ij}; *token_size* is the size of one token in bytes.

A CNN deployed on an edge platform can start its inference phase when the CNN can utilize the memory and the computational resources available on the platform to perform the CNN functionality, i.e., to execute all the layers in the CNN. Every layer can be executed on processors, such as CPUs, GPUs and/or FPGAs [17], available in the platform. If a platform has parallel processors (accelerators), such as GPUs or FPGAs, computations within the layer can be represented as one or multiple kernels [17] and offloaded on these accelerators by the CPUs. Otherwise, these computations are performed on the CPUs. If the computations within a CNN layer are offloaded on an accelerator with local memory, e.g., a GPU, the CNN layer input data and parameters, required to perform the computations, are copied from the main memory into

the local memory of the accelerator and the results of the computation are copied back to the main memory.

The layers of a CNN are executed in a specific order, determined by the data dependencies within the CNN and the manner the CNN is executed. Typically, the CNN layers are executed in a *sequential manner*, where the CNN execution is represented as $|L|$ computational steps and at every i-th computational step, CNN layer $l_i \in L$ is executed. However, as it will be explained in Section 2.4, a CNN can also be executed in *alternative (non-sequential) manners*, that involve exploitation of task-level (pipeline) parallelism, where the layers of the CNN are executed in parallel (pipelined) fashion. In this thesis, we consider both sequential and non-sequential manners of CNN execution. To represent a CNN, executed in a specific manner, we use the CSDF and SDF models of computation, presented in Section 2.5.

2.3 Edge platform used for CNN inference

Modern edge platforms used to execute the CNN inference are complex systems that host a large number of specific hardware components: processors, memory, power supply elements, sensors and others [32, 109]. Figure 2.4 shows a simplified structure of the NVIDIA Jetson TX2 edge platform [71]: one of the best-known edge platforms used to execute CNNs.

To perform computations within a CNN, an edge platform may host multiple heterogeneous processors such as central processing units (CPUs), graphics processing units (GPUs), field-programmable gate arrays (FPGAs), and/or Tensor Processing Units (TPUs) [32, 109]. For example, the Jetson TX2 platform shown in Figure 2.4 hosts a double-core Denver 2 CPU and a

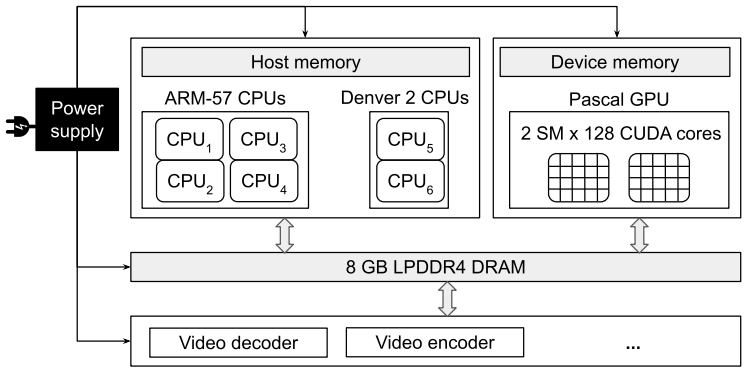


Figure 2.4: *Jetson TX2 edge platform*

quad-core ARM Cortex A57 CPU as well as an integrated Pascal GPU with a total of 256 CUDA cores. When the inference of a CNN is executed on the Jetson TX2 platform, computations within the CNN are typically performed on the GPU.

The memory infrastructure of an edge platform is used to store the CNN data and parameters, required for proper CNN inference. It typically consists of a *main memory*, accessible by all processes available on the platform, and a set of *local memories*, only accessible by specific processor(s). For example, the memory infrastructure of the Jetson TX2 platform shown in Figure 2.4 consists of the 8 GB LDDR4 DRAM, accessible by all the processors, available on the platform, as well as the *host_memory* and *device_memory*, i.e., the local memories, accessible only by the CPUs and the GPU, respectively.

The power supply elements of an edge platform provide power to all components available on the platform. Some edge platforms carry batteries that provide an autonomous limited power supply to the edge device. The Jetson TX2 platform, however, does not have a battery and requires an external power supply.

Finally, other components, available on the platform, e.g., video encoders and decoders, are used to collect data and facilitate parts of a CNN-based application other than the CNN itself.

2.4 Task- and data-level parallelism available in a CNN

As a computational model the CNN model is characterized with large amount of available parallelism. This parallelism can be exploited to speed-up the CNN inference and to efficiently utilize computational resources of an edge platform, where the CNN is executed.

The most widely exploited type of parallelism available within CNNs is the *data-level parallelism*, illustrated in Figure 2.5. This type of parallelism involves the same computation, e.g., Convolution, performed by a CNN layer over the CNN layer input data partitions. Efficient utilization of data-level parallelism allows to speed-up the inference of a CNN by accelerating the execution of individual CNN layers. This type of parallelism is exploited by the majority of existing Deep Learning (DL) frameworks, such as Keras [19], Pytorch [75], Tensorlow [1], TensorRT [72] and others [74]. The data-level parallelism, available within layer l_i of a CNN can be explicitly expressed by decomposition of the layer input data tensor X_i into a set of K sub-tensors $\{X_{i1}, X_{i2}, ..., X_{iK}\}$, where: 1) all sub-tensors $X_{ik}, k \in [1, K]$ can be processed in parallel by operator op_i. When layer l_i applies operator op_i to X_{ik}, it produces

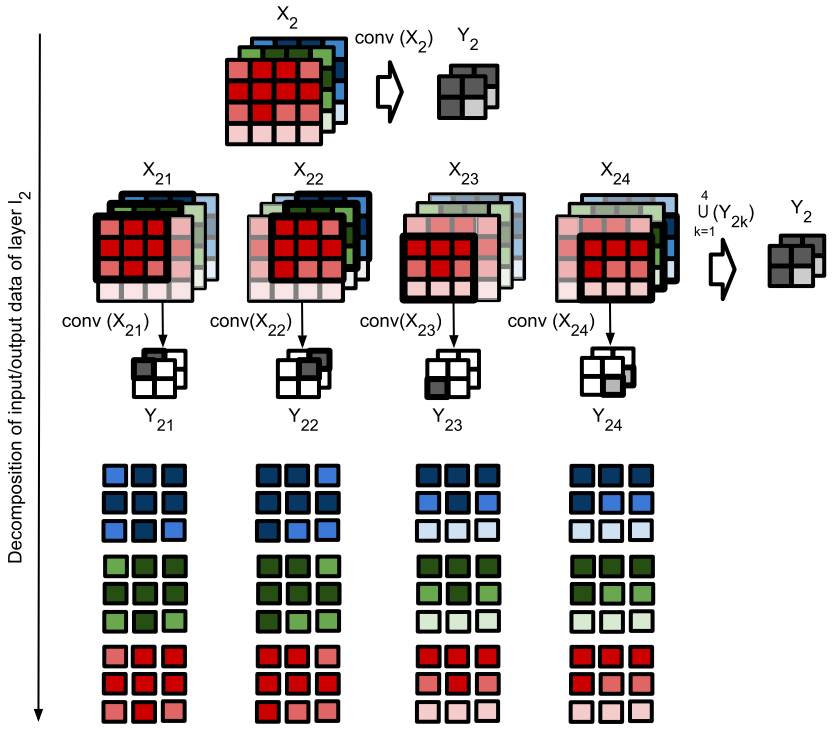


Figure 2.5: *data-level parallelism*

sub-tensor Y_{ik} of output data Y_i; 2) elements (pixels) within every X_{ik} can be processed in parallel. Figure 2.5 illustrates the data-level parallelism, available within convolutional layer l_2, shown in Figure 2.1 and explained in Section 2.1. In Figure 2.5, input data tensor X_2 of layer l_2 is decomposed into $K = 4$ overlapping sub-tensors $X_{2k}, k \in [1, 4]$ that can be processed in parallel. When layer l_2 processes sub-tensor X_{2k} with operator $op_2 = conv$, it produces sub-tensor Y_{2k} of output data Y_2, such that $Y_2 = \cup_{k=1}^{4} Y_{2k}$. Every sub-tensor X_{2k} shown in Figure 2.5 is subsequently decomposed into a set of pixels, where every pixel can be processed in parallel.

Another type of parallelism available in a CNN is known as *task-level parallelism* or *pipeline parallelism* [67, 101] among the CNN layers. This type of parallelism is related to the streaming nature of a CNN-based application, where the application accepts different input frames (images) from an input data stream. When a CNN is executed on a platform with multiple processors, the frames from the input data stream can be processed in a pipelined fashion by different layers of the CNN deployed on different processors.

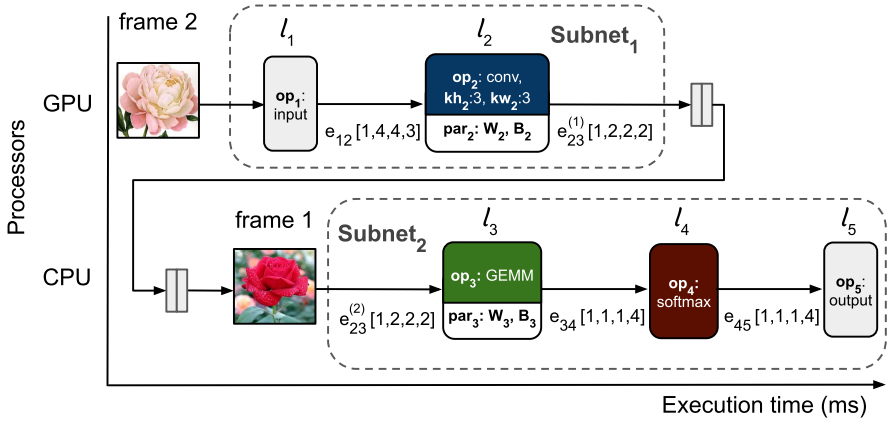


Figure 2.6: *task-level (pipeline) parallelism*

Figure 2.6 shows an example where the CNN shown in Figure 2.1 and explained in Section 2.1 is executed in a pipelined fashion on a platform with two processors: a CPU and a GPU. The layers of the CNN, representing computations within the CNN, are distributed over the platform processors: layers l_1 and l_2 are executed on the GPU, while layers l_3, l_4, and l_5 are executed on the CPU. The distributed layers form two CNN sub-graphs also referred as *partitions* [67, 101], annotated as $Subnet_1$ and $Subnet_2$. Partition $Subnet_1$ accepts frames from the application input data stream, processes these frames as specified by layers l_1 and l_2 and stores the results into a buffer associated with edge e_{23}. Partition $Subnet_2$ accepts the frames processed by partition $Subnet_1$ from edge e_{23}, further processes these, frames and produces the output data of the example CNN. Partitions $Subnet_1$ and $Subnet_2$ are executed on different processors in the platform and do not compete for the platform computational resources. Thus, when applied to different data (i.e., different frames), the partitions can be executed in parallel. In Figure 2.6, partitions $Subnet_1$ and $Subnet_2$ process frames *frame 2* and *frame 1* in parallel. This leads to overlapping execution of layers belonging to different partitions and enables for faster inference of the CNN, compared to conventional layer-by-layer (sequential) execution. However, pipelined CNN execution involves memory overheads. As shown in Figure 2.6, edge e_{23} of the example CNN is duplicated between the partitions $Subnet_1$ and $Subnet_2$ (see edges $e_{23}^{(1)}$ and $e_{23}^{(2)}$ and the corresponding buffers). Such duplication, called the double-buffering [37], is necessary for execution of the CNN as a pipeline. It prevents competition for memory (buffers) between the partitions when accessing data associated with edge e_{23}. If the double buffering is not enabled the CNN partitions compete

for access to edge e_{23}, thereby creating stalls in the pipeline and reducing the CNN throughput.

It is worth noting that the parallelism available in a CNN is not explicitly specified in the CNN model, introduced in Section 2.1. The number of parallel tasks, executed to perform the CNN model functionality, and the exact communication and synchronization mechanisms between these tasks are internally determined by the utilized DL framework, and can vary for different frameworks. For example, the well-known DL frameworks [1,75] represent the functionality of every CNN layer l_i as multiple tasks, where the total number of tasks depends on the number of CPUs available on the target edge platform. The frameworks [94,101] represent the functionality of the same layer l_i as one task or part of a task. Therefore, the task-level parallelism is not explicitly specified in the CNN model. Analogously, the data-level parallelism is not explicitly defined in the CNN model because the number of input/output data sub-tensors K, the number of elements within sub-tensors X_{ik} and Y_{ik}, and other decomposition parameters are determined by every design framework individually, can vary for different frameworks, and even within one framework. For example, the TensorRT framework [72] is capable of representing the *conv* operator as: 1) the $GEMM$ operator, so for every Convolutional layer $K = 1$; 2) a direct convolution where $K >= 1$ is computed from the attributes of a layer, performing the *conv* operator.

2.5 CSDF and SDF models of computation

The CSDF model [10] is a well-known dataflow model of computation, widely used for model-based design in the embedded systems community. An application modelled 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 2.7 shows an example of a CSDF model $G(A, C)$, where $A = \{a_1, a_2, a_3, a_4, a_5\}$ and $C = \{c_{12}, c_{22}, c_{23}, c_{34}, c_{45}\}$.

Every actor $a_i \in A$ in the CSDF model represents a certain functionality of the application, modelled 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_2 is shown in Figure 2.7. Actor a_2 performs execution sequence $F_2 = [conv, conv]$, shortly written as $[2 * conv]$, meaning actor a_2 has $P_2 = 2$ phases and performs function $f_2(p) = conv$ at each of its phases $p \in [1, 2]$.

Every FIFO communication channel $c_{ij} \in C$ represents a data dependency

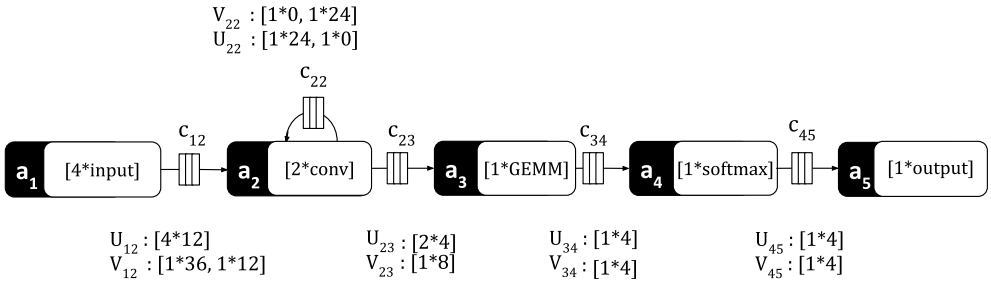


Figure 2.7: *CSDF model of computation*

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 $U_{ij} : [u_{ij}(1), u_{ij}(2),..., u_{ij}(P_i)]$ of length P_i specifies the production of data tokens into channel c_{ij} by its source actor a_i. Analogously, consumption sequence $V_{ij} : [v_{ij}(1), v_{ij}(2),..., v_{ij}(P_j)]$ 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_{22} is shown in Figure 2.7. For communication channel c_{22}, actor a_2 is a source and a sink actor. The production sequence $U_{22} : [24, 0]$, formally written as $U_{22} : [1 * 24, 1 * 0]$ specifies, that at phase $p = 1$ actor a_2 produces 24 tokens to channel c_{22}, and at phase $p = 2$ actor a_2 produces 0 tokens to channel c_{22}. Analogously, the consumption sequence $V_{22} : [1 * 0, 1 * 24]$, specifies that during phase $p = 1$ actor a_2 consumes 0 tokens from channel c_{22}, and at phase $p = 2$ actor a_2 consumes 24 tokens from channel c_{22}.

A special case of the CSDF model, where every actor has only one phase, is called Synchronous Data Flow (SDF) model [57]. An example of a SDF model is shown in Figure 2.8.

At every firing, actor $a_i \in A$ of the SDF model consumes $v_{ki}(1)$ data tokens, executes function $f_i(1)$ and produces $u_{ij}(1)$ data tokens. For example, actor a_2 shown in Figure 2.8, at every firing consumes 48 data tokens from channel c_{12}, executes function $f_2(1) = conv$ and produces 8 data tokens to channel c_{23}. For simplicity, we omit the number of phases while annotating the execution

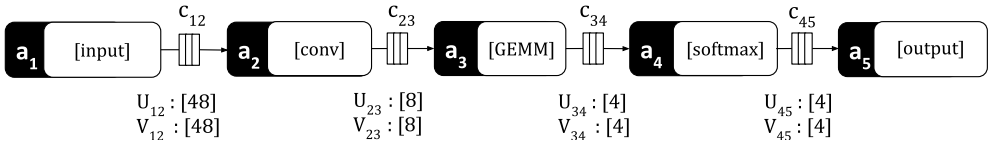


Figure 2.8: *SDF model of computation*

sequence, the production sequence, and the consumption sequence of the SDF model. For example, the consumption sequence of actor a_2 shown in Figure 2.8 is annotated simply as $[48]$ instead of $[1 * 48]$.

2.6 Genetic Algorithm (GA)

Genetic Algorithm (GA) [83] is a well-known heuristic approach, widely used for finding optimal solutions for complex Design Space Exploration (DSE) problems. In a GA, a population of candidate solutions to an optimization problem is evolved toward better solutions. A GA has two important problem-specific attributes: a chromosome and a fitness function.

A *chromosome* is a genetic representation of the solution. Typically, a chromosome is defined as a set of parameters (genes), joined into a string. An example of a chromosome is shown in Figure 2.9 This chromosome shows a distribution (mapping) of the computations within the layers of the CNN shown in Figure 2.1 and explained in Section 2.1 onto the computational resources of the Jetson TX 2 edge platform shown in Figure 2.4 and explained in Section 2.3. The chromosome shown in Figure 2.9 is a string of 5 genes, where every i-th gene, $i \in [1, 5]$, specifies a processor of the Jetson TX 2 platform which performs computations within layer l_i of the CNN. For example, the chromosome specifies that computations within layer l_1 of the CNN are performed on the GPU of the Jetson TX 2 platform.

A *fitness function* is a special function, which evaluates the quality of GA solutions, represented as chromosomes, and guides the GA-based search fro (pareto) optimal solutions. During the search, the fitness function should be minimized or maximized. For example, a fitness function can estimate the throughput of inference of the CNN, shown in Figure 2.1 and executed on the Jetson TX2 platform as specified in the chromosome shown in Figure 2.9.

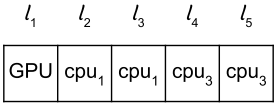


Figure 2.9: *Chromosome*

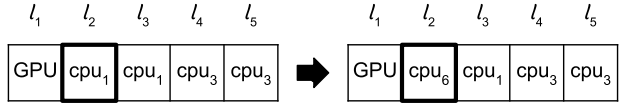


Figure 2.10: *Single-gene mutation*

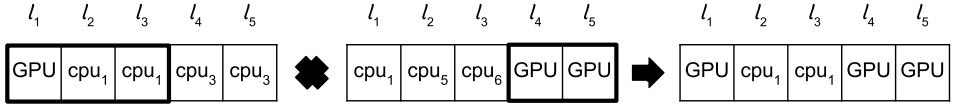


Figure 2.11: *Simple two-parent recombination (cross-over)*

If this fitness function is maximized during a GA-based search, it will guide the search towards finding chromosomes that ensure high CNN inference throughput.

Once the chromosome and the fitness function are defined, a GA can proceed to perform evolution, i.e., search for optimal solutions. The evolution is an iterative process. It starts from a population of randomly generated chromosomes. At each iteration, the fitness of every chromosome in the population is evaluated using the fitness function. Then, the chromosomes with the best score are selected from the population and are subjected to two genetic operators, called recombination (cross-over) and mutation. During the re-combination two selected chromosomes exchange parts (typically, halves) to produce a new chromosome. During the mutation, one or multiple genes of the chromosome randomly change their values. In this thesis, we use a standard two-parent crossover and a single-gene mutation as proposed in [83] and illustrated in Figure 2.11 and Figure 2.10, respectively. The new population of candidate chromosomes, generated using the recombination and mutation, is used in the next iteration of the GA-based search. The GA-based search terminates when either a maximum number of iterations or a termination condition (e.g., satisfactory fitness level) has been reached.

Beside the two problem-specific attributes, mentioned above, a GA also has a number of parameters such as the maximum number of GA iterations, the number of individuals in the initial population, the probability of mutation in the chromosomes and others [83]. These parameters are typically user-defined.