Chapter 3

Methodology for high-throughput CNN inference

Svetlana Minakova, Erqian Tang, Todor Stefanov. "Combining Task- and Data-level Parallelism for High-Throughput CNN Inference on Embedded CPUs-GPUs MPSoCs". *In Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pp. 18-35, Pythagoreio, Samos Island, Greece, July 05-09, 2020.

IN this chapter, we present our methodology for high-throughput CNN inference at the Edge, which corresponds to the first research contribution of this thesis summarized in Section 1.5.1. 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 3.1 introduces, in more details, the problem addressed by our novel methodology. Section 3.2 summarizes the novel research contributions, presented in this chapter. An overview of the related work is given in Section 3.3. Section 3.4 presents the platform model, used in this Chapter to represent a target edge platform, where the high-throughput CNN inference is executed. Section 3.5 presents our proposed methodology. Section 3.6 presents the experimental study performed by using the proposed methodology. Section 3.7 ends the chapter with conclusions.

3.1 Problem statement

As mentioned in Chapter 1 (see Section 1.2), many CNN-based applications require CNNs to process their input data streams fast, i.e., to have high throughput. These applications are often executed on edge platforms based on CPUs-GPUs multi-processor systems-on-chip (MPSoCs) [63]. Due to their specific design, CPUs-GPUs MPSoCs offer energy-efficient and high-performance solutions, which makes them very suitable for running high-throughput CNN inference at the Edge [14]. However, achieving high-throughput execution of the computationally-intensive CNN inference phase on embedded CPUs-GPUs MPSoCs is a complex task.

On the one hand, it requires effective utilization of parallelism, available in a CNN. When the CNN inference is executed on an embedded CPUs-GPUs MPSoC, the CNN computational workload is distributed among the heterogeneous MPSoC processors: embedded CPUs and GPUs. Due to their specific structure, the CPUs are more suitable for handling task-level parallelism, compared to GPUs, whereas GPUs are more suitable for handling data-level parallelism, compared to CPUs [88]. Thus, for efficient execution of the CNN inference on an embedded MPSoC, the task-level parallelism should be handled by the CPUs, available in an embedded MPSoC, i.e., different CNN layers should, if possible, be executed on different CPUs, and the overall CNN computational workload should be balanced among the CPUs [6]. Additionally, the data-level parallelism, available within CNN layers, should be handled by embedded GPUs, i.e., the embedded CPUs should offload data-parallel computations within the CNN layers onto the embedded GPUs, thereby accelerating the computations within CNN layers for further improvement of the CNN inference throughput, already achieved by efficient task-level parallelism exploitation. Thus, efficient execution of the CNN inference on an embedded CPUs-GPUs MPSoC involves efficient exploitation of both task-level parallelism and data-level parallelism, available in the CNN.

On the other hand, effective utilization of task- and data-level parallelism requires proper communication and synchronization between tasks, executed on different processors of an embedded MPSoC. In this respect, attempting to utilize an unnecessary large amount of CNN parallelism on limited embedded MPSoC resources, results in unnecessary communication and synchronization overheads, that reduce the CNN inference throughput. Thus, to achieve high CNN inference throughput, the CNN inference, executed on an embedded MPSoC, should utilize the right amount of parallelism, which matches the computational capacity of the MPSoC.

Based on the discussion above, we argue, that efficient execution of the

CNN inference on a CPUs-GPUs embedded MPSoC requires:

1. efficient handling of the task-level parallelism, available in a CNN, by CPUs;
2. CPU workload balancing;
3. efficient handling of the data-level parallelism, available in a CNN, by GPUs;
4. efficient exploitation of task- and data-level parallelism, which matches the computational capacity of an embedded MPSoC.

However, the existing Deep Learning (DL) frameworks [1,42,43,49,72,74, 75,90,94,101], that enable execution of the CNN inference on embedded CPUs-GPUs MPSoCs, only partially satisfy requirements 1) to 4), mentioned above. These frameworks can be divided into two main groups. The first group includes frameworks [101] and [94], that exploit only task-level parallelism, available in a CNN, and efficiently utilize only embedded CPUs. Thus, these frameworks satisfy requirements 1) and 2), mentioned above, and do not satisfy requirement 3). The second group includes frameworks [1,42,43,49, 72,74,75,90], that exploit only data-level parallelism, available in a CNN, and efficiently utilize only embedded GPUs. Thus, these frameworks satisfy requirement 3), mentioned above, but do not satisfy requirements 1) and 2). Moreover, all frameworks [1,42,43,49,72,74,75,90,94,101] directly utilize the CNN computational model to execute the CNN inference on embedded CPUs-GPUs MPSoCs. The large amount of parallelism, available in a CNN model, typically does not match the limited computational capacity of embedded CPUs-GPUs MPSoC. Thus, frameworks [1,42,43,49,72,74,75,90,94,101] do not satisfy requirement 4), mentioned above.

Therefore, in this chapter, we propose a novel methodology for efficient execution of the CNN inference on embedded CPUs-GPUs MPSoCs.

3.2 Contributions

In this chapter, we propose a novel methodology for execution of the CNN inference on embedded CPUs-GPUs MPSoCs (Section 3.5). Our methodology exploits task-level (pipeline) and data-level parallelism, available in a CNN and explained in Section 2.4, to efficiently distribute (map) the computations within the CNN to the computational resources of an edge platform. Thus, our methodology takes full advantage of all CPU and GPU resources, available in an MPSoC, and ensures high-throughput CNN inference execution on the MPSoC. Exploitation of task-level (pipeline) parallelism together with data-level parallelism for high-throughput CNN inference at the edge is our main novel contribution. Other important novel contributions are:

1. the automated conversion of a CNN model into a functionally equivalent SDF model (Section 3.5.1). Unlike the CNN model, presented in Section 2.1 and typically used to represent CNNs, the SDF model, presented in Section 2.5, can explicitly specify task- and data-level parallelism, available in a CNN. Moreover, unlike the CNN model, the SDF model has the tasks communication and synchronization mechanisms, suitable for efficient mapping and execution of a CNN on an embedded MPSoC. Thus, a conversion of a CNN model into a SDF model enables for efficient mapping and execution of a CNN on an embedded CPUs-GPUs MPSoC.

2. the automated conversion of a CNN model into a functionally equivalent platform-aware executable CSDF model (see Section 2.5 for the CSDF model definition), which efficiently utilizes CPUs-GPUs embedded MPSoC computational resources (Section 3.5.3);

3. taking state-of-the-art CNNs from the ONNX models zoo [7] and mapping them on a Nvidia Jetson MPSoC [71], we achieve a 1.36% to 42% higher throughput, when the CNN inference is executed with our methodology, compared to the throughput of the CNN inference, executed by the best-known and state-of-the-art Tensorrt DL framework [72] for Nvidia Jetson MPSoCs (Section 3.6).

3.3 Related work

The well-known Deep Learning (DL) frameworks, such as TensorFlow [1], Pytorch [75] and others [74] and some of the Deep Learning frameworks for embedded devices such as [42,43,49,50,72,90] efficiently exploit data-level parallelism, available in a CNN, for efficient utilization of embedded GPUs. However, these frameworks do not exploit task-level parallelism, available in a CNN. They execute the CNN inference layer-by-layer, i.e., at every computational step only one CNN layer is executed. Such layer-by-layer execution of CNN layers is performed either on a single CPU, which utilizes GPU devices for acceleration, or on all available embedded CPUs. Thus, at every computational step, either some of the embedded CPUs are not utilized, or embedded GPUs are not utilized. Therefore, these frameworks cannot take full advantage of all CPU and GPU resources and cannot achieve high CNN inference throughput, typically required for the CNN inference, executed on embedded MPSoCs [23,24,87]. Unlike these frameworks, our methodology exploits together both task-level parallelism and data-level parallelism, available in the CNN. In our methodology, the CNN layers are distributed on embedded CPUs, such that the CNN workload is balanced among the CPUs, and at every computational step several CNN layers are executed in parallel (pipeline) fashion. At the same time, some of the computations within CNN layers are performed on efficiently-shared embedded GPU devices. Thus, in our methodology, at every computational step all available CPU and GPU resources are efficiently utilized. Therefore, our methodology allows to achieve higher CNN inference throughput, compared to the frameworks, presented in [1,42,43,49,72,74,75,90].

The frameworks, presented in [101] and [94], exploit task-level parallelism, available among CNN layers, for efficient execution of the CNN inference on an embedded MPSoC. In these frameworks, CNN layers are distributed on the embedded CPUs and executed in parallel (pipeline) fashion, which provides higher CNN throughput than sequential (layer-by-layer) execution of CNN layers. However, these frameworks do not utilize embedded GPUs, available in an MPSoC. As a consequence, these frameworks cannot increase further the CNN inference throughput. In contrast, in our methodology, the throughput, achieved by efficient task-level parallelism exploitation, is further increased by exploitation of data-level parallelism, i.e., by exploitation of embedded GPU devices to accelerate the computations within CNN layers. In our methodology, some computations within CNN layers are offloaded onto embedded GPUs and performed in parallel. Parallel execution of computations within CNN layers allows to reduce the execution time of individual CNN layers and to increase the CNN inference throughput. Therefore, our methodology ensures higher CNN inference throughput, compared to frameworks [101] and [94].

3.4 Edge platform model

In this Chapter, we represent an edge platform as a platform model. The platform model provides a simplified, yet accurate description of computational resources, available on the platform. As mentioned above, in this Chapter we concentrate on edge platforms based on embedded CPUs-GPUs MPSoCs, which computational resources are composed of CPUs and embedded GPUs.

Formally, we define a platform model as a set $platform = \{cpu, gpu\}$, where $cpu = \{cpu_1, cpu_2, ..., cpu_n\}$ is a set of CPU cores, available on the platform and used for CNN inference; $gpu = \{gpu_1, gpu_2, ..., gpu_m\}$ is a set of all GPU devices, available in the platform, and typically $m \leq n$. For example, we model the Jetson TX2 edge platform shown in Figure 2.4 and explained in Section 2.3, as platform model $Jetson = \{cpu, gpu\}$, where $cpu = \{cpu_1, cpu_2, ..., cpu_5\}$ is a set of 5 out of 6 CPU cores, available on the platform and used for CNN inference. The sixth core available on the platform is not included in the model because it is allocated to other parts of a CNN-based application and is not used for CNN inference; $gpu = \{gpu_1\}$ is a set of GPUs, available on the platform and used for CNN inference.

3.5 Methodology

In this section, we present our methodology for high-throughput CNN inference at the Edge. Our methodology, shown in Figure 3.1, consists of three main steps. In Step 1 (Section 3.5.1), we convert a CNN, represented as a CNN model (see Section 2.1 for the CNN model definition) into a functionally equivalent SDF model (see Section 2.5 for the SDF model definition). Unlike the CNN model, the SDF model explicitly specifies task- and data-level parallelism, available in a CNN, as well as it explicitly specifies the tasks communication and synchronization mechanisms, suitable for efficient mapping and execution of a CNN on an embedded MPSoC. Thus, a conversion of a CNN model into a SDF model enables for efficient mapping and execution of a CNN on an embedded CPUs-GPUs MPSoC.

In Step 2 (Section 3.5.2), we find an efficient mapping of the SDF model, obtained in Step 1, on an embedded CPUs-GPUs MPSoC represented as the edge platform model, proposed in Section 3.4. The mapping describes the

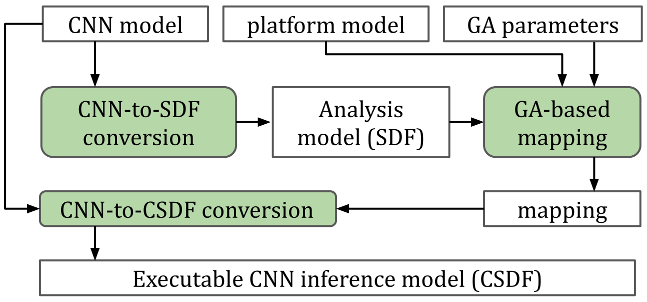


Figure 3.1: *Methodology for high-throughput CNN inference*

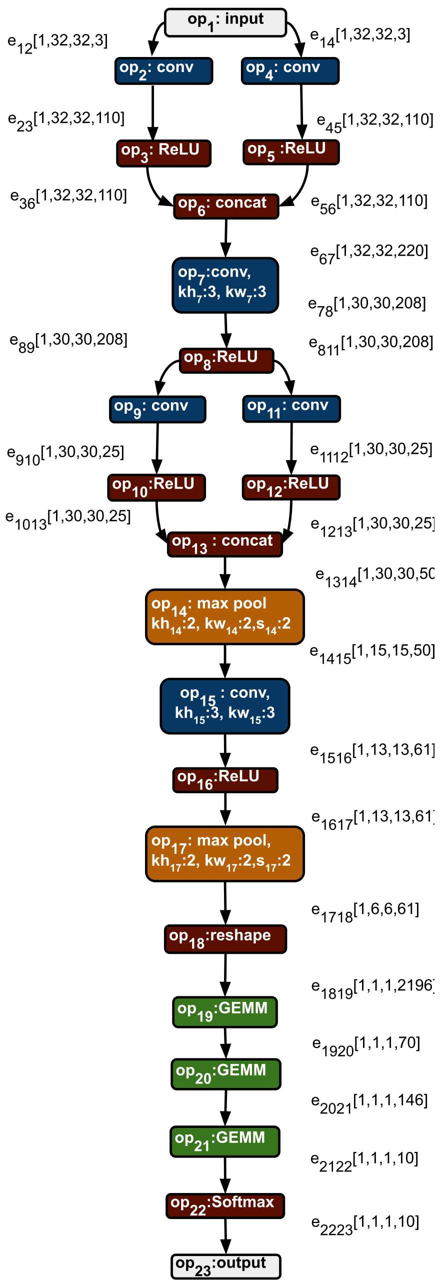


Figure 3.2: *CNN (input) model*

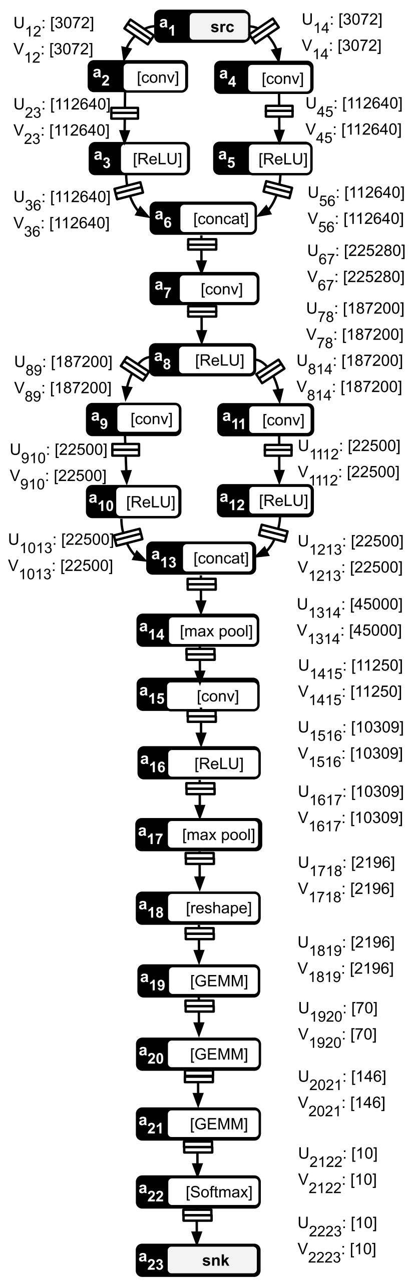


Figure 3.3: *SDF (analysis) model*

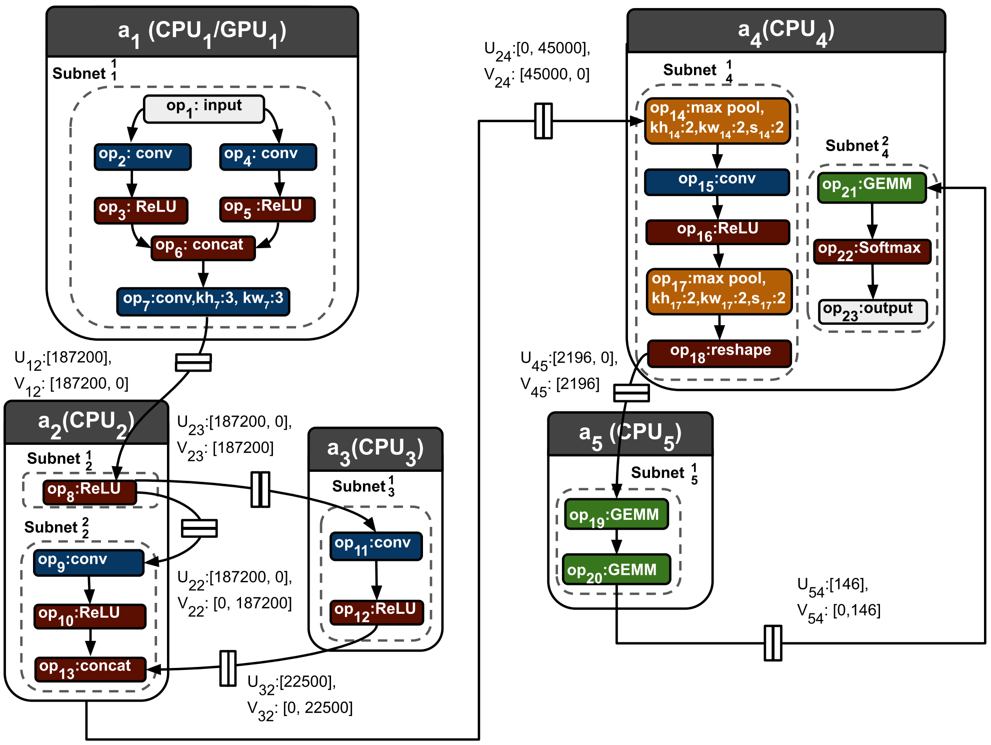


Figure 3.4: *CSDF (executable CNN inference) model*

distribution of the CNN inference computational workload on an embedded MPSoC. The mapping is considered efficient when it ensures high-throughput CNN inference. To find such a mapping, we propose to utilize a simple Genetic Algorithm (GA), which basic concepts and standard parameters are presented in Section 2.6.

Finally, in Step 3 (Section 3.5.3), we use the mapping, obtained in Step 2, to convert a CNN model into a final platform-aware executable application model. The final application model is represented as a Cyclo-Static Dataflow (CSDF) model (see Section 2.5 for the CSDF model definition). The CSDF model, obtained in Step 3, describes the CNN inference as an application, efficiently distributed over embedded MPSoC processors and exploiting the right amount of task- and data-level parallelism, which matches the computational capacity of an embedded MPSoC.

To illustrate the Steps performed by our methodology, we use an example, where we apply our methodology to 1) the CNN model shown in Figure 3.2; 2) the *Jetson* platform model introduced in Section 3.4; 3) a set of GA parameters where the initial population size = 1000, number of epochs = 500, mutation probability = 5%. The SDF model and the CSDF model, automatically obtained in Step 1 and Step 3 of our methodology from the aforementioned inputs 1), 2) and 3), are shown in Figure 3.3 and Figure 3.4, respectively.

3.5.1 CNN-to-SDF conversion

In this section, we show how we automatically convert a CNN model, introduced in Section 2.1, into a functionally equivalent SDF model, introduced in Section 2.5. The conversion procedure is given in Algorithm 1. An example of the CNN-to-SDF conversion, performed by Algorithm 1, is given in Section 3.5, where the CNN model, shown in Figure 3.2, is automatically converted into the SDF model, shown in Figure 3.3.

Algorithm 1 accepts as an input a CNN model $CNN(L, E)$ and generates as an output a functionally equivalent SDF model $G(A, C)$. In Line 1, it creates an empty SDF model. In Lines 2 to 6, Algorithm 1 converts every CNN layer l_i into a functionally equivalent actor a_i. According to the definition of the SDF model, given in Section 2.5, the sequence F_i, executed by actor a_i, has a single phase. At its single phase, actor a_i executes operator op_i of layer l_i, thereby reproducing the functionality of layer l_i. In Lines 7 to 12, Algorithm 1 converts every CNN edge e_{ij} into FIFO channel c_{ij}. In Lines 9 to 11, Algorithm 1 defines the production sequence U_{ij} and the consumption sequence V_{ij} of channel c_{ij}. Both sequences have a single element, computed as the number of data

Algorithm 1: CNN-to-SDF conversion

```
   Input: $CNN(L, E)$
   Result: $G(A, C)$
1  $A, C \leftarrow \emptyset$; $G(A, C) \leftarrow$ SDF model $(A, C)$;
2  for $l_i \in L$ do
3      $F_i = \emptyset$;
4      $F_i \leftarrow F_i + op_i$;
5      $a_i \leftarrow$ actor $(F_i)$;
6      $A \leftarrow A + a_i$;
7  for $e_{ij} \in E$ do
8      $c_{ij} \leftarrow$ FIFO channel $(a_i, a_j)$;
9      $U_{ij} \leftarrow \emptyset$; $V_{ij} \leftarrow \emptyset$;
10     $U_{ij} \leftarrow U_{ij} + |e_{ij}.data|$;
11     $V_{ij} \leftarrow V_{ij} + |e_{ij}.data|$;
12     $C \leftarrow C + c_{ij}$;
13 return $G(A, C)$
```

elements $|e_{ij}.data|$, exchanged through edge e_{ij} of the CNN model.

Unlike the CNN model $CNN(L,E)$, accepted as an input by Algorithm 1, the functionally equivalent SDF model $G(A,C)$, generated by Algorithm 1, explicitly specifies both task-level and data-level parallelism, which could be exploited during the CNN inference phase, as well as this SDF explicitly specifies the communication and synchronization mechanism between the actors/tasks, needed to execute the CNN inference properly. The task-level parallelism, available among CNN layers, is explicitly specified in the SDF model topology, where every actor $a_i \in A$ is a task, performing the functionality of CNN layer $l_i \in L$, and the total number of tasks, needed to perform the CNN model functionality, is equal to the number of actors in the SDF model. The communication and synchronization between the tasks, are explicitly specified by the SDF FIFO channels, where every channel $c_{ij} \in C$ specifies, that actor $a_i \in A$ communicates with actor $a_j \in A$ through a FIFO buffer, and the production-consumption rates of the channels $c_{ij} \in C$ determine the frequency and the order of the actors firings - for more details see [57]. The data-level parallelism is explicitly specified in the channels production rates. For example, production rate $U_{36} = [112640]$ of FIFO channel c_{36}, shown in Figure 3.3, explicitly specifies that, when actor a_3 fires, it produces 112640 data tokens, and each token can be obtained in parallel by executing 112640 parallel $ReLU$ operations within each firing of a_3.

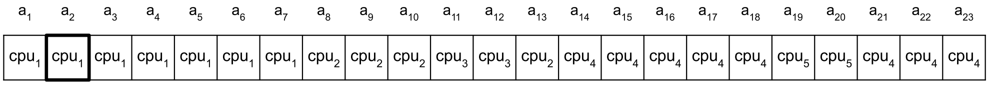
The SDF explicit specification of the tasks, that can be potentially performed during the CNN inference, and the SDF explicit specification of the communication and synchronization between the tasks, allow to perform a search for efficient mappings of the CNN onto an embedded CPUs-GPUs MPSoC.

3.5.2 GA-based mapping

In this section, we show how we obtain an efficient mapping of a SDF model $G(A,C)$, generated by Algorithm 1, onto an embedded CPUs-GPUs MPSoC $Jetson = \{\{cpu_1, cpu_2, ..., cpu_5\}, \{gpu_1\}\}$ introduced in Section 3.4. In our methodology, the CNN inference tasks, explicitly specified as SDF actors, are executed on embedded CPU cores, that are able to efficiently handle the task-level parallelism. To efficiently utilize the data-level parallelism, available within the tasks, some of the CPU cores offload computations on the embedded GPUs. Since the number of embedded GPU devices is limited, it may occur, that the efficient exploitation of task-level parallelism, by embedded CPUs, is disrupted due to CPUs competition for the limited embedded GPU devices. To avoid such disruption, for every embedded GPU $gpu_j \in gpu$, we allocate a

Table 3.1: *Mapping example*

cpu_1/gpu_1	cpu_2	cpu_3	cpu_4	cpu_5
$a_1, a_2, a_3, a_4, a_5, a_6, a_7$	a_8, a_9, a_{10}, a_{13}	a_{11}, a_{12}	$a_{14}, a_{15}, a_{16}, a_{17}, a_{18}, a_{21}, a_{22}, a_{23}$	a_{19}, a_{20}



a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}	a_{18}	a_{19}	a_{20}	a_{21}	a_{22}	a_{23}
cpu_1	cpu_1	cpu_1	cpu_1	cpu_1	cpu_1	cpu_1	cpu_2	cpu_2	cpu_2	cpu_3	cpu_3	cpu_2	cpu_4	cpu_4	cpu_4	cpu_4	cpu_4	cpu_5	cpu_5	cpu_4	cpu_4	cpu_4

Figure 3.5: *Mapping chromosome example*

single CPU core $cpu_i \in cpu$, which offloads computations on gpu_j.

Based on the discussion above, we define a mapping of SDF model $G(A, C)$ onto *Jetson*, as a partition of actors set A into n subsets, where $n = |cpu|$ is the number of CPU cores, available in the MPSoC. We denote such mapping as ${}^nA = \{{}^nA_1, {}^nA_2, ..., {}^nA_n\}$, where each ${}^nA_i \in {}^nA$ is a subset of actors, mapped on cpu_i, such that $\cap_{i=1}^{n} {}^nA_i = \emptyset$, and $\cup_{i=1}^{n} {}^nA_i = A$. The first $m = |gpu|$ number of CPU cores in mapping ${}^nA$ offload computations on the corresponding embedded GPUs, i.e., the computations within every actor $a_k \in {}^nA_j, j \in [1, m]$ are performed on gpu_j, and the computations within every actor $a_k \in {}^nA_i, i \in [m+1, n]$ are performed on cpu_i. An example of mapping ${}^5A = \{{}^5A_1, {}^5A_2, {}^5A_3, {}^5A_4, {}^5A_5\}$ of the SDF model $G(A, C)$, shown in Figure 3.3 on the *Jetson* CPUs-GPUs MPSoC, is given in Table 3.1. Every Column in Table 3.1 corresponds to a subset ${}^5A_i, i \in [1, 5]$. For example, Column 1 in Table 3.1 corresponds to subset ${}^5A_1 = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7\}$. The actors within subset ${}^5A_1$ are mapped on cpu_1, which offloads computations on gpu_1. Column 2 in Table 3.1 describes subset ${}^5A_2 = \{a_8, a_9, a_{10}, a_{13}\}$. Every actor $a_i \in {}^5A_2$ is mapped on cpu_2. Since the example *Jetson* MPSoC does not have gpu_2, all computations within actors in ${}^5A_2$ are performed only on cpu_2.

We consider that a mapping is efficient, if it ensures that the workload is balanced [6] among all embedded CPU cores, including those, that offload computations on embedded GPUs. We note, that obtaining such an efficient mapping of an SDF graph onto a CPUs-GPUs MPSoC is a complex Design Space Exploration (DSE) problem. In our methodology, to solve this problem, we propose to use a simple Genetic Algorithm (GA) with a standard two-parent crossover and a single-gene mutation, as introduced in Section 2.6. To utilize such a GA for searching of an efficient mapping ${}^nA$, we represent mapping ${}^nA$, as a mapping chromosome: a string of length $|A|$, where every gene is a CPU core $cpu_i \in cpu$. An example of the chromosome, corresponding to mapping ${}^5A$, shown in Table 3.1, is given in Figure 3.5.

In our methodology, we search for a mapping, in which the workload is balanced among all CPU cores, available in the MPSoC, i.e., the difference in execution time between every pair of CPU cores $(cpu_i \in cpu, cpu_j \in cpu), i \neq j$, is minimized. Thus, we define a specific fitness-function $fitness$ to be minimized during the GA-based search as:

$$fitness = \sum_{\forall(cpu_i, cpu_j) \in cpu^2} |\tau_{cpu_i} - \tau_{cpu_j}| \tag{3.1}$$

where τ_{cpu_i} and τ_{cpu_j} are the total execution time of cpu_i and cpu_j, respectively. For every $cpu_i \in cpu$, τ_{cpu_i} is computed as:

$$\tau_{cpu_i} = \tau^{t}_{cpu_i} + \tau^{com}_{cpu_i} \tag{3.2}$$

where $\tau^{t}_{cpu_i}$ is the time, required by cpu_i to execute all tasks, mapped on cpu_i; $\tau^{com}_{cpu_i}$ is the time, required for communication of cpu_i with other embedded processors. The time $\tau^{t}_{cpu_i}$ is computed as:

$$\tau^{t}_{cpu_i} = \sum_{a_k \in {}^{n}A_i} \tau_{(f_k(1), cpu_i)} \tag{3.3}$$

where ${}^{n}A_i$ is the set of all actors, mapped on cpu_i; $f_k(1)$ is the function, performed by actor $a_k \in {}^{n}A_i$ at every firing; $\tau_{(f_k(1), cpu_i)}$ is the time, taken by cpu_i to execute $f_k(1)$, measured on the MPSoC. The time $\tau^{com}_{cpu_i}$ is computed as:

$$\tau^{com}_{cpu_i} = \sum_{a_k \in {}^{n}A_i} (\tau_w * \sum_{c_{kj} \in C} u_{kj}(1) + \tau_r * \sum_{c_{qk} \in C} v_{qk}(1)) \tag{3.4}$$

where ${}^{n}A_i$ is the set of all actors, mapped on cpu_i; $c_{kj} \in C$ is an output channel of actor $a_k \in {}^{n}A_i$, to where, at each firing, actor a_k produces $u_{kj}(1)$ tokens; $c_{qk} \in C$ is an input channel of actor a_k, from where, at each firing, actor a_k consumes $v_{qk}(1)$ tokens; τ_r and τ_w specify the time, needed by a CPU core, to read and write one data token, respectively. τ_r and τ_w are measured on the MPSoC.

3.5.3 CNN-to-CSDF model conversion

In this section, we show how we automatically convert a CNN model, introduced in Section 2.1, into a final executable platform-aware application, represented as a CSDF model, introduced in Section 2.5. The conversion procedure is given in Algorithm 2.

Algorithm 2: CNN-to-CSDF conversion

```
Input: $CNN(L,E), {}^nA$
Result: $G(A,C)$
1  $A, C \leftarrow \emptyset$; $G(A,C) \leftarrow$ CSDF model $(A,C)$;
2  $E_{out} = \emptyset$;
3  for ${}^nA_i \in {}^nA$ do
4      $F_i = \emptyset$; $p = 1$;
5      $Q = \emptyset$; $visited = \emptyset$;
6      for $l_k : a_k \in {}^nA_i \wedge l_k \notin visited$ do
7          $L_i^p, E_i^p \leftarrow \emptyset$;
8          $Q = Q + l_k$;
9          while $Q \neq \emptyset$ do
10             $l_j = Q.pop()$;
11             $L_i^p = L_i^p + l_j$;
12             $visited = visited + l_j$;
13             if $\exists e_{js} \in E : a_s \notin {}^nA_i$ then
14                 for $e_{js} \in E$ do
15                     $E_{out} = E_{out} + e_{js}$;
16                 break;
17             else
18                 for $e_{js} \in E, l_s \notin visited$ do
19                     $Q = Q + l_s$;
20                     $E_i^p = E_i^p + e_{js}$;
21         $Subnet_i^p$ = new Subnet $(L_i^p, E_i^p)$;
22         $F_i = F_i + Subnet_i^p$;
23         $p = p + 1$;
24     $a_i \leftarrow$ actor $(F_i)$;
25     $A = A + a_i$;
26 for $e_{ij} \in E_{out}$ do
27     $a_k \in A : l_i \in L_k^g$; $a_r \in A : l_j \in L_r^z$;
28     $c_{kr} \leftarrow$ FIFO channel $(a_k, a_r)$;
29     $u_{kr}(p) = \begin{cases} |e_{ij}.data|, \text{ if p = g} \\ 0, \text{ otherwise} \end{cases}$
30     $v_{kr}(p) = \begin{cases} |e_{ij}.data|, \text{ if p = z} \\ 0, \text{ otherwise} \end{cases}$
31 return $G(A,C)$
```

Algorithm 2 accepts as inputs a CNN model $CNN(L,E)$ and an efficient mapping ${}^nA$, obtained in Section 3.5.2, and generates a CSDF model $G(A,C)$, which performs the functionality of the CNN model $CNN(L,E)$, efficiently mapped on an embedded MPSoC, as specified by mapping ${}^nA$. An example of the CSDF model $G(A,C)$, generated by Algorithm 2, using as inputs the CNN model $CNN(L, E)$, shown in Figure 3.2, and mapping ${}^5A$, shown in Table 3.1 and explained in Section 3.5.2, is given in Figure 3.4.

In Line 1, Algorithm 2 creates an empty CSDF model. In Lines 3-25, Algorithm 2 generates the set of actors A, such that every actor $a_i \in A$ represents the functionality of all CNN layers, mapped on CPU core cpu_i, as specified in mapping ${}^nA$, where for $\forall l_k \in L$, executed on cpu_i, $\exists a_k \in {}^nA_i$. At every phase $p \in [1, P_i]$ actor a_i executes function $Subnet_i^p$, implemented by means of an existing DL framework. Every $Subnet_i^p$ performs layer-by-layer execution of layers $L_i^p \subseteq L$, mapped on cpu_i, and connected via edges E_i^p. For example, actor a_3, shown in Figure 3.4, represents the functionality of all CNN layers, mapped on cpu_3. It executes $F_3 = \{Subnet_3^1\}$, where $Subnet_3^1$ performs layer-by-layer execution of layers $L_3^1 = \{l_{11}, l_{12}\}$, connected via edges $E_3^1 = \{e_{1112}\}$, on cpu_3.

Every edge $e_{js} \in E$ between layers l_j and l_s, sequentially executed on the same CPU core, is implemented by means of an existing DL framework, e.g. as device memory, shared by layers l_j and l_s [72]. If layers l_j and l_s, connected via edge $e_{js} \in E$, are executed on different CPU cores, the task-level parallelism is exploited between these layers, and edge e_{js} is converted into a FIFO channel, which explicitly specifies and implements the communication and synchronization between actors, executing layers l_j and l_s. For example, edge e_{811}, shown in Figure 3.2, connects layer l_8, executed by actor a_2 on cpu_2, and layer l_{11}, executed by actor a_3 on cpu_3. Thus, edge e_{811} is converted into a FIFO channel c_{23}, shown in Figure 3.4, where c_{23} explicitly specifies and implements the communication and synchronization between actor a_2, executing layer l_8 and actor a_3, executing layer l_{11}.

Between some actors, cyclic dependencies occur, that may lead to deadlocks in the CSDF model. To avoid the deadlocks, Algorithm 2 specifies the execution of every actor a_i in one or more phases, such that at every phase $p \in [1, P_i]$, actor a_i has no cyclic dependencies. For the example, shown in Figure 3.4, a cyclic dependency occurs between actors a_2 and a_3. If actor a_2 would execute layers l_8 and l_{13} in one phase, according to the semantics of the CSDF model [10], it would expect 187200 data tokens to be present in channel c_{12} and 22500 data tokens to be present in channel c_{32}, before it can fire. However, data in channel c_{32}, should be produced by actor a_3, which, before it can fire, expects actor a_2 to produce 187200 data tokens in channel c_{23}. Thus, such execution would lead to a deadlock in the CNN inference. To avoid the deadlock, Algorithm 2 specifies the execution of actor a_2 in 2 phases. At phase $p = 1$, actor a_2 executes only layer l_8. It consumes data only from channel c_{12}, and produces data to channel c_{23}, such that actor a_3 can fire.

At phase $p = 2$, actor a_2 consumes data only from channel c_{32}, and executes layers l_9, l_{10} and l_{13}. Thus, at every phase $p = [1, 2]$, actor a_2 has no cyclic dependencies, and no deadlock occurs in the CSDF model execution.

In Lines 5-23, Algorithm 2 performs a mapping-aware Breadth-First Search (BFS) [26] over the CNN model graph and determines functions $Subnet_i^p, p \in [1, P_i]$, executed by actor a_i. In Line 7, for every not-visited layer l_k, mapped on cpu_i, Algorithm 2 creates an empty set of layers L_i^p and an empty set of edges E_i^p. In Line 8, it adds layer l_k to the BFS queue [26] Q, and starts BFS. In Lines 10-12, Algorithm 2 extracts layer l_j from Q and adds l_j to L_i^p. In Line 13, Algorithm 2 checks, if layer l_j, mapped on cpu_i, has at least one child layer l_s, which is not mapped on cpu_i. If the condition in Line 13 is met, to avoid the deadlocks, which can occur in a CSDF model, as discussed above, Algorithm 2 stops adding layers to L_i^p and goes to Lines 14-15, where it adds every output edge of layer l_j to the list of outer edges E_{out}, utilized in Lines 26-30 of Algorithm 2 for CSDF channels generation. If every child layer l_s of layer l_j is mapped on cpu_i (condition in Line 13 of Algorithm 2 is not met), in Lines 18-20, Algorithm 2 adds every connection e_{js} to the set E_i^p, and every layer l_s to Q and continues BFS.

In Line 21, Algorithm 2 creates function $Subnet_i^p$, which performs layer-by-layer execution of layers L_i^p, connected via edges E_i^p. In Line 22, Algorithm 2 adds function $Subnet_i^p$ to execution sequence F_i of actor a_i. When all layers, mapped on cpu_i, are visited, Algorithm 2 adds actor a_i, which executes F_i, to the CSDF model actors set (see Lines 24-25).

In Lines 26-30, Algorithm 2 converts every outer edge $e_{ij} \in E_{out}$ into a CSDF channel c_{kr}, specifying and implementing the communication and synchronization between actor $a_k \in A$ executing layer l_i, and actor $a_r \in A$ executing layer l_j . For example, for edge e_{78}, shown in Figure 3.2, Algorithm 2 creates FIFO channel c_{12}, shown in Figure 3.4, where actor a_1 executes layer l_7, and actor a_2 executes layer l_8.

3.6 Experimental results

In this section, we present our results from an experiment, where real-world CNNs from the ONNX models zoo [7] are mapped and executed on the NVIDIA Jetson TX2 embedded CPUs-GPUs MPSoC [71]. We compare the CNN inference throughput, which we measure, when the CNN is mapped on the NVIDIA Jetson TX2 by: 1) the popular ARM CL framework [8], which on the NVIDIA Jetson MPSoC can exploit only task-level parallelism, available in the CNN; 2) the best-known and state-of-the-art for the NVIDIA Jetson

Table 3.2: *Experimental results, average over 100 runs*

CNN	Throughput (fps)			Thr. increase, compared to TensorRT (%)
	ARM CL	TensorRT	Our	
bvlc alexnet	8.7	104	140	35
VGG 19	1.84	15	21.3	42
bvlc googlenet	3.9	118	154	31
tiny yolo v2	3.2	131	133	1.36
inception v1	4.25	122	166	36
resnet18	8.7	137	143	4.37
densenet121	3	62	69	12
Emotion FER	21.2	325	416	28

TX2 MPSoC, TensorRT DL framework [72], which exploits only data-level parallelism, available in the CNN; 3) our methodology, explained in Section 3.5, which exploits both task- and data-level parallelism and uses the ARM CL framework to implement CNN layers on embedded CPUs together with the TensorRT framework to implement CNN layers on embedded GPUs. For every CNN in the experimental results: 1) The throughput is measured on the platform as an average value over 100 CNN inference executions; 2) the original (float32) data precision is utilized, such that the baseline CNN accuracy is preserved; 3) The dataset parameters, such as size and precision of input data samples as well as the batch size are obtained from the ONNX model representation; 4) The GA, utilized for efficient mapping search (see Section 3.5.2) is executed with initial population size = 1000, number of epochs = 500, mutation probability = 5%. If for 50 epochs no improvements are achieved by the GA, the GA stops.

The experimental results are given in Table 3.2. Column 1 lists the CNNs. Columns 2-4 show the CNN inference throughput in frames per second (fps) for ARM CL, TensorRT, and our methodology, respectively. Columns 2 and 4 in Table 3.2 show that the throughput achieved by the ARM CL framework is much lower than the throughput, achieved by our methodology. This difference occurs because our methodology exploits both task- and data-level parallelism, available in the CNN, whereas the ARM CL framework, executing the CNN inference on the NVIDIA Jetson MPSoC, does not offload computations on the embedded GPU, available in the MPSoC. Therefore, ARM CL does not efficiently exploit the data-level parallelism, available in the CNN. Columns 3 and 4 in Table 3.2 show that our methodology achieves higher inference throughput than the TensorRT framework. This difference occurs because our methodology exploits both task- and data-level parallelism, whereas TensorRT executes the CNN inference layer-by-layer and exploits only data-level parallelism, available in the CNN. Column 5 shows the throughput increase achieved by our methodology in comparison with the TensorRT framework, which achieves highest throughput for every CNN among the TensorRT and ARM CL frameworks. The numbers in Column 5 indicate that our methodology enables to achieve 1.36% to 42% throughput increase compared to the TensorRT framework.

3.7 Conclusion

We propose a novel methodology which exploits both task- and data-level parallelism, available in a CNN, and takes full advantage of all CPU and GPU resources, available in a MPSoC, to achieve high-throughput CNN inference execution. We evaluated our proposed methodology by mapping a set of real-world CNNs on the NVIDIA Jetson TX2 embedded CPUs-GPUs MPSoC. The evaluation results show that taking real-world CNNs from the ONNX models zoo and mapping them on the Jetson MPSoC, a 1.36% to 42% higher throughput is achieved when the CNN inference is executed with our methodology compared to the throughput of the CNN inference, executed by the best-known and state-of-the-art TensorRT DL framework for the Jetson MPSoC.