Combining task- and data-level parallelism for high-throughput CNN inference on embedded CPUs-GPUs MPSoCs

Minakova, S.; Tang, E.; Stefanov, T.P.; Orailoglu, A.; Jung, M.; Reichenbach, M.

Combining Task- and Data-Level Parallelism for High-Throughput CNN Inference on Embedded CPUs-GPUs MPSoCs

Svetlana Minakova(✉), Erqian Tang, and Todor Stefanov

LIACS, Leiden University, Leiden, The Netherlands
{s.minakova,e.tang,t.p.stefanov}@liacs.leidenuniv.nl

Abstract. Nowadays Convolutional Neural Networks (CNNs) are widely used to perform various tasks in areas such as computer vision or natural language processing. Some of the CNN applications require high-throughput execution of the CNN inference, on embedded devices, and many modern embedded devices are based on CPUs-GPUs multiprocessor systems-on-chip (MPSoCs). Ensuring high-throughput execution of the CNN inference on embedded CPUs-GPUs MPSoCs is a complex task, which requires efficient utilization of both task-level (pipeline) and data-level parallelism, available in a CNN. However, the existing Deep Learning frameworks utilize only task-level (pipeline) or only data-level parallelism, available in a CNN, and do not take full advantage of all embedded MPSoC computational resources. Therefore, in this paper, we propose a novel methodology for efficient execution of the CNN inference on embedded CPUs-GPUs MPSoCs. In our methodology, we ensure efficient utilization of both task-level (pipeline) and data-level parallelism, available in a CNN, to achieve high-throughput execution of the CNN inference on embedded CPUs-GPUs MPSoCs.

Keywords: Convolutional Neural Networks · Dataflow models · SDF · CSDF · Mapping · Hight throughput

1 Introduction

Convolutional Neural Networks (CNNs) are biologically inspired graph computational models, characterized by high degree of available parallelism. Due to their ability to handle large, unstructured data, CNNs are widely used to perform various tasks in areas such as computer vision and natural language processing [1]. The CNNs execution typically includes two phases: training and inference [1]. At the training phase the optimal CNN parameters are established. At the inference phase, a trained CNN is applied to the actual data and performs the task for which the CNN is designed. Due to the high complexity of state-of-the-art CNNs, their training and inference phases are usually performed by high-performance

A. Orailoglu et al. (Eds.): SAMOS 2020, LNCS 12471, pp. 18–35, 2020.
https://doi.org/10.1007/978-3-030-60939-9_2

platforms, and provided as cloud services. However, some applications, e.g. [2–4], require high-throughput execution of the CNNs inference, which cannot be provided as a cloud service. These applications are typically deployed on embedded devices.

Many modern embedded devices are based on multi-processor systems-on-chip (MPSoCs) [5]: complex integrated circuits, that consist of processing elements with specific functionalities. Due to their specific design, MPSoCs offer energy-efficient and high-performance solutions for applications running on embedded devices. In addition to hosting various processing elements, capable of running the CNN inference, such as central processing units (CPUs), embedded graphics processing units (embedded GPUs), and field-programmable gate arrays (FPGAs), MPSoCs integrate many other components, such as communication network components and video accelerators, that allow to deploy the entire embedded application on a single chip. Therefore, MPSoCs seem to be a promising solution for the deployment of the CNN inference phase on embedded devices.

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, a high-throughput CNN inference execution requires effective utilization of the parallelism, available in a CNN. The parallelism available in a CNN can be divided into two different types: task-level (pipeline) and data-level parallelism. The parallelism, available among CNN layers, hereinafter referred as task-level parallelism [6], involves execution of several CNN layers in a parallel pipelined fashion, where each layer may perform computations, different from the computations, performed by other CNN layers. Utilization of this type of parallelism allows to reduce the overall computation time, and increase the overall CNN inference throughput, compared to sequential execution of CNN layers [7]. The parallelism, available within a CNN layer, hereinafter referred as data-level parallelism [6], involves the same computation, e.g., Convolution, performed by a CNN layer over the CNN layer input data partitions. Utilization of this type of parallelism allows to improve the CNN inference throughput by accelerating the execution of individual CNN layers [8–12].

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. 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 [13]. 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 [14]. 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 [7–12,15–18], 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 [7] and [18], 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 [8–12,15–17], 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 [7–12,15–18] 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 [7–12,15–18] do not satisfy requirement (4), mentioned above.

Therefore, in this paper, we propose a novel methodology for efficient execution of the CNN inference on embedded CPUs-GPUs MPSoCs. Our methodology consists of three main steps. In Step 1 (Sect. 4.1), we convert a CNN model into a functionally equivalent Synchronous Dataflow (SDF) model [19]. 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 is necessary for efficient mapping and execution of a CNN on an embedded CPUs-GPUs MPSoC. In Step 2 (Sect. 4.2), we propose to utilize a Genetic Algorithm [20], to find an efficient mapping of the SDF model, obtained on Step 1, on an embedded CPUs-GPUs MPSoC. The mapping, obtained by the Genetic Algorithm, describes the distribution of the CNN inference computational workload on an embedded MPSoC, that satisfies requirements (1) to (3), mentioned above. In Step 3 (Sect. 4.3), we use the mapping, obtained in Step 2, to convert a CNN model into a final platform-aware executable Cyclo-Static Dataflow (CSDF) application model [21]. 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. Thus, our methodology satisfies all requirements (1) to (4), mentioned above, to take full advantage of the CPU and GPU resources, available in an MPSoC. Moreover, as we show by experimental results (Sect. 5), our methodology enables high-throughput execution of the CNN inference on embedded CPUs-GPUs MPSoCs.

Paper Contributions

In this paper, we propose a novel methodology for execution of the CNN inference on embedded CPUs-GPUs MPSoCs (Sect. 4), which takes full advantage of all CPU and GPU resources, available in an MPSoC, and ensures high-throughput CNN inference execution on CPUs-GPUs MPSoCs. The exploitation of task-level (pipeline) parallelism, available among CNN layers, together with data-level parallelism, available within CNN layers, for high-throughput execution of the CNN inference on embedded MPSoCs, is our main novel contribution. Other important novel contributions are: (1) the automated conversion of a CNN model into a SDF model, suitable for searching for an efficient mapping of a CNN onto an embedded MPSoC (Sect. 4.1); (2) the automated conversion of a CNN model into a functionally equivalent platform-aware executable CSDF model, which efficiently utilizes CPUs-GPUs embedded MPSoC computational resources (Sect. 4.3); (3) taking state-of-the-art CNNs from the ONNX models zoo [22] and mapping them on a Nvidia Jetson MPSoC [23], we achieve a 20% 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 [12] for Nvidia Jetson MPSoCs (Sect. 5).

2 Related Work

The well-known Deep Learning (DL) frameworks, such as TensorFlow [8], Caffe2 [9] and others [10] and some of the Deep Learning frameworks for embedded devices such as [11,12,15–17] 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 [2–4]. 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 [8–12,15–17].

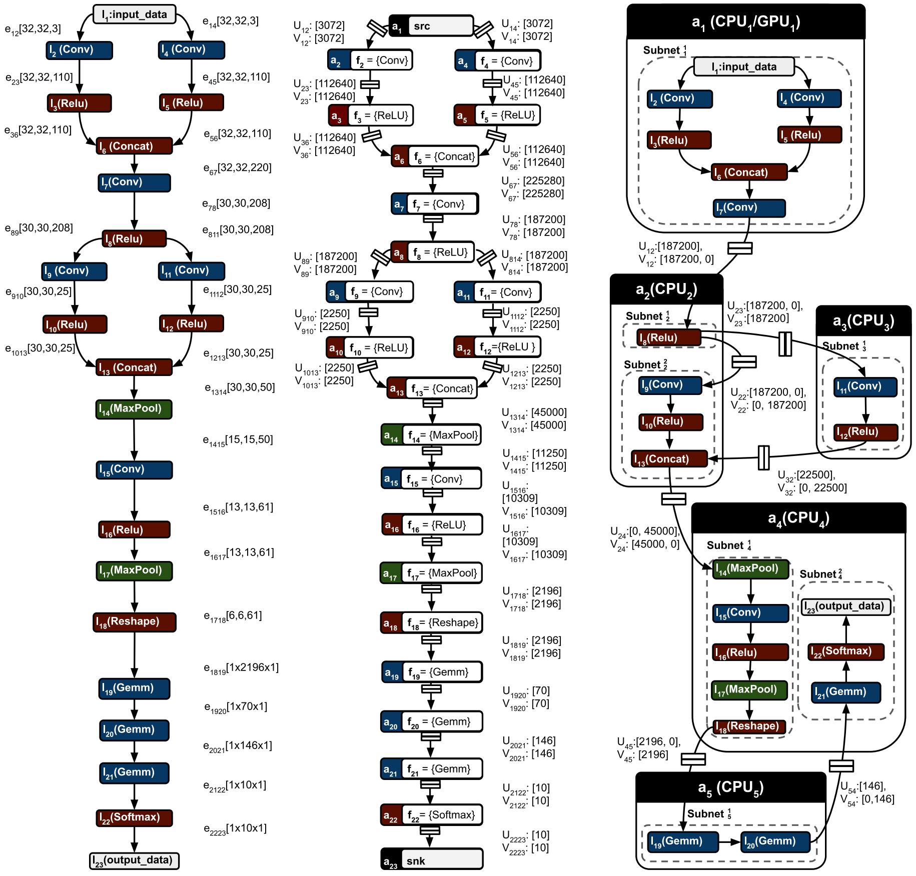
The frameworks, presented in [7] and [18], exploit task-level parallelism, available among CNN layers, for efficient execution of the CNN inference on an embedded MPSoC. In these frameworks, CNN layers are distributed on the embedded CPUs and executed in parallel (pipeline) fashion, which provides higher 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 [7] and [18].

3 Background

In this section, we describe the Convolutional Neural Network (CNN) model, the Synchronous Dataflow (SDF) model, the Cyclo-Static Dataflow (CSDF) model, and specific features of embedded CPUs-GPUs MPSoCs, essential for understanding the proposed methodology.

3.1 CNN Model

The CNN is a computational model [24], commonly represented as a directed acyclic computational graph $CNN(L, E)$ with a set of nodes L, also called layers,



(a) CNN model (b) SDF model (c) CSDF model

Fig. 1. CNN, SDF and CSDF computational models

and a set of edges E. An example of a CNN model with $|L| = 23$ layers and $|E| = 24$ edges is given in Fig. 1(a). The CNN model specifies the transformations over the CNN input data, e.g. an image, required to obtain the CNN output data, e.g. an image classification result. Every layer $l_i \in L$ specifies a part of these transformations. It has a layer input data X_i, a layer output data Y_i, and an operator op_i, so that $Y_i = op_i(X_i)$. The operator op_i determines the main difference between the CNN layer types. The most common layer types [1] are:

- convolutional with $op_i = conv$;
- pooling with $op_i \in \{maxpool, avgpool, globalmaxpool, globalavgpool\}$;
- activation with $op_i \in \{relu, thn, sigm\}$;
- Fully Connected (FC) with $op_i \in \{matmul, gemm\}$;
- normalization with $op_i \in$ {Batch Normalization (BN), Local Response Normalization (LRN)};

- data with $op_i \in \{input, output\}$;
- loss with $op_i = softmax$;

The CNN layers input and output data are stored in multidimensional arrays, called tensors [24]. In this paper, every data tensor T has the format $T^{[W^T,H^T,C^T]}$, where W^T is the tensor width, H^T is the tensor height, C^T is the number of channels. An example of layer l_2 is given in Fig. 1(a). Layer l_2 is a convolutional layer. It applies operator *conv* to its input data tensor $X_2^{[32,32,3]}$ and produces output data tensor $Y_2^{[32,32,110]}$. The input data tensor X_i comes to layer l_i from other CNN layers, as specified by the CNN edges $e_{ji} \in E$. Each CNN edge $e_{ji} \in E$, represents a data dependency between layers l_j and l_i, such that $Y_j \subseteq X_i$. An example of a CNN edge is edge e_{12}, shown in Fig. 1(a). Edge e_{12} specifies, that the output data of layer l_1 is the input data of layer l_2, i.e., $Y_1^{[32,32,3]} = X_2^{[32,32,3]}$.

A CNN is characterized with a large amount of available task-level parallelism and data-level parallelism. However, this parallelism is not explicitly specified in the CNN computational model. Therefore, 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 [8–10] represent the functionality of every CNN layer l_i as multiple tasks, where the total number of tasks depends on the layer mapping. The frameworks [7,18] 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 available data-level parallelism, is not explicitly specified in the CNN model. The data-level parallelism, available within CNN layer l_i can be explicitly expressed by decomposition of the layer output data tensor Y_i into a set of K output sub-tensors $\{Y_{i1}, Y_{i2}, ..., Y_{iK}\}$, where (1) every Y_{ik} can be computed in parallel by operator op_i, applied to the input data sub-tensor X_{ik}; (2) data elements within every Y_{ik} and every X_{ik} can be processed in parallel. However, the CNN model does not specify the data-level parallelism explicitly, i.e., the number of 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, and can vary for different frameworks. For example, the Caffe2 [9] framework internally represents the *conv* operator as the *gemm* operator, so for every Convolutional layer $K = 1$; The TensorFlow framework [10] uses a convolutional operator directly, and computes $K \geq 1$ from the *conv* operator parameters.

3.2 SDF Model

The SDF model [19] is a well-known dataflow model of computation, widely used in the embedded systems community for efficient mapping of applications on embedded devices [25], including embedded CPUs-GPUs MPSoCs. An application, modeled as a SDF, is a directed graph $G(A, C)$, which consists of a set of nodes A, also called actors, communicating through a set of FIFO channels C. An example of a SDF model with $|A| = 23$ actors and $|C| = 24$ FIFO channels is given in Fig. 1(b). Every actor $a_i \in A$ is a task, which performs certain application functionality, represented as a function f_i. An example of SDF actor a_3 is given in Fig. 1(b). Actor a_3 performs function $f_3 = \{ReLU\}$. Every FIFO channel $c_{ij} \in C$ represents data dependency and transfers data in tokens between actors a_i and a_j. c_{ij} has data production rate U_{ij} and data consumption rate V_{ij}. U_{ij} specifies the production of data tokens into channel c_{ij} by actor a_i. V_{ij} specifies the consumption of data tokens from channel c_{ij} by actor a_j. An example of a communication FIFO channel c_{36} is given in Fig. 1(b). Channel c_{36} transfers data between actors a_3 and a_6. It has production rate $U_{36} = [112640]$, specifying, that, at each firing, actor a_3 produces 112640 data tokens into channel c_{36} and consumption rate $V_{36} = [112640]$, specifying, that, at each firing, actor a_6 consumes 112640 data tokens from channel c_{36}.

3.3 CSDF Model

The CSDF model [21] is a generalization of the SDF model, briefly introduced in Sect. 3.2. Unlike in the SDF model, in the CSDF model, actors have cyclically changing firing rules, and channels have cyclically changing production and consumption rates. An example of a CSDF model with $|A| = 5$ actors and $|C| = 7$ FIFO channels is given in Fig. 1(c). Every actor $a_i \in A$ in the CSDF model performs an execution sequence F_i of length P_i, where $p \in [1, P_i]$ is called a phase of actor a_i. At every phase p, actor a_i executes function $f_i(((p-1) mod P_i) + 1)$. An example of CSDF actor a_2 is given in Fig. 1(c). Actor a_2 performs execution sequence $F_2 = \{Subnet_2^1, Subnet_2^2\}$, where $Subnet_2^1$ and $Subnbet_2^2$ are functions. Actor a_2 has $P_2 = 2$ phases. At phase $p = 1$ actor a_3 performs function $f(1) = Subnet_2^1$, and at phase $p = 2$ actor a_2 performs function $f(2) = Subnet_2^2$.

Instead of fixed production and consumption rates, utilized in the SDF model, in the CSDF model every FIFO channel $c_{ij} \in C$ has data production sequence U_{ij} of length P_i and data consumption sequence V_{ij} of length P_j. Every element $u_{ij}(p) \in U_{ij}$ of the production sequence specifies the amount of tokens, produced by actor a_i into channel c_{ij} at actors phase $p \in [1, P_i]$. Analogously, every element $v_{ij}(p) \in V_{ij}$ of the consumption sequence specifies the amount of tokens, consumed by actor a_j from channel c_{ij} at actors phase $p \in [1, P_j]$. An example of a CSDF communication channel c_{12} is given in Fig. 1(c). Channel c_{12} transfers data between actors a_1 and a_2. It has production sequence $U_{12} = [187200]$, specifying, that actor a_1 produces 187200 tokens into channel c_{12} at its single phase $p = 1$, and consumption sequence $V_{12} = [187200, 0]$, specifying, that actor a_2 consumes 187200 tokens from channel c_{12} at phase $p = 1$ and 0 tokens at phase $p = 2$.

3.4 Embedded CPUs-GPUs MPSoC

We define an embedded MPSoC as a tuple $MPSoC(cpu, gpu)$, where $cpu = \{cpu_1, cpu_2, ..., cpu_n\}$ is a set of all CPU cores, available in the MPSoC; $gpu =$

$\{gpu_1, gpu_2, ..., gpu_m\}$ is a set of all GPU devices, available in the MPSoC, and typically $m \leq n$. An example of an embedded CPUs-GPUs MPSoC with $n = 5$ CPU cores and $m = 1$ GPU device is shown in Fig. 2.

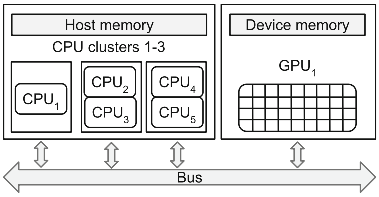


Fig. 2. MPSoC

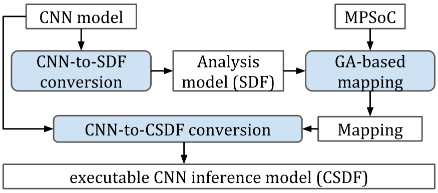


Fig. 3. Our methodology

4 Our Methodology

In this section, we present our three-step methodology for high-throughput execution of a CNN inference on embedded CPUs-GPUs MPSoCs. Our methodology is shown in Fig. 3. In Step 1 (Sect. 4.1), we convert a CNN model into a functionally equivalent SDF model, suitable for efficient mapping of a CNN onto an embedded CPUs-GPUs MPSoC. In Step 2 (Sect. 4.2), we utilize a Genetic Algorithm to find an efficient mapping of the SDF model, obtained in Step 1, onto the MPSoC. In Step 3 (Sect. 4.3), we use the mapping, obtained in Step 2, to convert the CNN model into a CSDF model, representing the final platform-aware executable CNN inference application, which takes full advantage of all CPU and GPU resources, available in an MPSoC for high-throughput execution of the CNN inference on the MPSoC.

Algorithm 1: CNN-to-SDF conversion

```
Input: $CNN(L, E)$
Result: $G(A, C)$
$A, C \leftarrow \emptyset$; $G(A, C) \leftarrow$ SDF model $(A, C)$;
for $l_i \in L$ do
    $f_i = op_i$;
    $a_i \leftarrow$ actor $(f_i)$;
    $A \leftarrow A + a_i$;
for $e_{ij} \in E$ do
    $c_{ij} \leftarrow$ FIFO channel $(a_i, a_j)$;
    $U_{ij} = W^{Y_i} * H^{Y_i} * C^{Y_i}$;
    $V_{ij} = W^{X_j} * H^{X_j} * C^{X_j}$;
    $C \leftarrow C + c_{ij}$;
return $G(A, C)$
```

4.1 CNN to SDF Model Conversion

In this section, we show how we automatically convert a CNN model, introduced in Sect. 3.1, into a functionally equivalent SDF model, introduced in Sect. 3.2. The conversion is given in Algorithm 1. It 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, Algorithm 1 creates an empty SDF model. In Lines 2–5, Algorithm 1 converts every CNN layer l_i into a functionally equivalent actor a_i. Function f_i, executed by actor a_i, is the operator op_i, performed by layer l_i over its input data tensor X_i to produce its output data tensor Y_i. In Lines 6–10, Algorithm 1 converts every CNN edge e_{ij} into FIFO channel c_{ij}. The production rate U_{ij} of channel c_{ij} is computed in Line 8 of Algorithm 1, and is equal to the number of data elements in tensor Y_i. The consumption rate V_{ij} of channel c_{ij} is computed in Line 9 of Algorithm 1, and is equal to the number of data elements in tensor X_j. An example of the CNN-to-SDF conversion, performed by Algorithm 1, is shown in Fig. 1, where the CNN model, shown in Fig. 1(a), is automatically converted into the SDF model, shown in Fig. 1(b).

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 [19]. 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 Fig. 1(b) and explained in Sect. 3.2, 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.

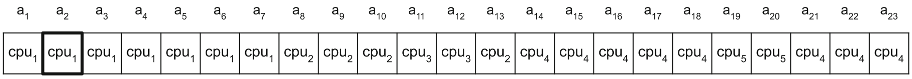
The SDF explicit specification of the tasks, that can be potentially performed during the CNN inference, and the SDF explicit specification of the communication and synchronization between the tasks, allow to perform a search for efficient mappings of the CNN onto an embedded CPUs-GPUs MPSoC.

4.2 Efficient 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

Table 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

Fig. 4. Mapping chromosome example

$MPSoC(cpu, gpu)$, defined in Sect. 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 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 $MPSoC(cpu, gpu)$, 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 Fig. 1(b) and explained in Sect. 3.2, on the embedded MPSoC, shown in Figure 2 and explained in Sect. 3.4, is given in Table 1. Every Column in Table 1 corresponds to a subset ${}^5A_i, i \in [1, 5]$. For example, Column 1 in Table 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 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 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 [14] 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 Genetic Algorithm (GA) [20]: a well-known heuristic approach, widely used for finding optimal solutions for complex DSE problems. We use a simple GA with standard two-parent crossover, a single-gene mutation, and standard user-defined GA parameters, such as initial offspring size, number of epochs, mutation and crossover probabilities [20]. To utilize such a GA for searching of an efficient mapping ${}^nA$, we have to specify two problem-specific GA attributes, namely chromosome and fitness function [20]. A chromosome is a representation of a GA solution (in our methodology a solution is a mapping) as a set of parameters (genes), joined into a string [20]. We represent mapping ${}^nA$, as 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 1, is given in Fig. 4.

The fitness-function is a special function, which measures the quality of the solutions and guides the GA-based search. During the search, the fitness function should be minimized or maximized. In our methodology, we search for a mapping, in which the workload is balanced among all CPU cores, available in $MPSoC(cpu, gpu)$, 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 ϕ to be minimized during the GA-based search as:

$$\phi = \sum_{\forall (cpu_i, cpu_j) \in cpu^2} |\tau_{cpu_i} - \tau_{cpu_j}| \tag{1}$$

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:

$$\tau_{cpu_i} = \tau^t_{cpu_i} + \tau^{com}_{cpu_i} \tag{2}$$

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:

$$\tau^t_{cpu_i} = \sum_{a_k \in {}^nA_i} \tau_{(f_k, cpu_i)} \tag{3}$$

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

$$\tau^{com}_{cpu_i} = \sum_{a_k \in {}^nA_i} (\tau_w * \sum_{c_{kj} \in C} U_{kj} + \tau_r * \sum_{c_{jk} \in C} V_{jk}) \tag{4}$$

where ${}^nA_i$ is the set of all actors, mapped on cpu_i; $c_{kj} \in C$ is an output channel of actor $a_k \in {}^nA_i$, to where, at each firing, actor a_k produces U_{kj} tokens; $c_{jk} \in C$ is an input channel of actor a_k, from where, at each firing, actor a_k consumes V_{jk} 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.

4.3 CNN to CSDF Model Conversion

In this section, we show how we automatically convert a CNN model, introduced in Sect. 3.1, into a final executable platform-aware application, represented as a CSDF model, introduced in Sect. 3.3. The conversion is given in Algorithm 2. Algorithm 2 accepts as inputs a CNN model $CNN(L, E)$ and an efficient mapping ${}^{n}A$, obtained in Sect. 4.2, 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 ${}^{n}A$. An example of the CSDF model $G(A, C)$, generated by Algorithm 2, using as inputs the CNN model $CNN(L, E)$, shown in Fig. 1(a) and explained in Sect. 3.1, and mapping ${}^{5}A$, shown in Table 1 and explained in Sect. 4.2, is given in Fig. 1(c). 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 ${}^{n}A$, where for $\forall l_k \in L$, executed on cpu_i, $\exists a_k \in {}^{n}A_i$. 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 Fig. 1(c), 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 [12]. 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 communication and synchronization between actors, executing layers l_j and l_s. For example, edge e_{811}, shown in Fig. 1(a), 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 Fig. 1(c), where c_{23} explicitly specifies and implements 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 Fig. 1(c), 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 [21], 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.

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     $u_{kr}(p) = \begin{cases} W^{Y_i} * H^{Y_i} * C^{Y_i}, \text{ if p = g} \\ 0, \text{ otherwise} \end{cases}$
30     $v_{kr}(p) = \begin{cases} W^{X_j} * H^{X_j} * C^{X_j}, \text{ if p = z} \\ 0, \text{ otherwise} \end{cases}$
31 return $G(A, C)$
```

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 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 Fig. 1(a), Algorithm 2 creates FIFO channel c_{12}, shown in Fig. 1(c), where actor a_1 executes layer l_7, and actor a_2 executes layer l_8.

5 Experimental Results

In this section, we present our results from an experiment, where real-world CNNs from the ONNX models zoo [22], are mapped and executed on the NVIDIA Jetson TX2 embedded CPUs-GPUs MPSoC [23]. 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 [27], 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 TX2 MPSoC, Tensorrt DL framework [12], which exploits only data-level parallelism, available in the CNN; (3) our methodology, explained in Sect. 4, which exploits both task- and data-level parallelism, and uses the ARM CL framework to implement CNN layers on embedded CPUs, and 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) original (float32) data precision is utilized, so 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 Sect. 4.2) 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 2.

Column 1 in Table 2 lists the CNNs. Columns 2–4 in Table 2 show the CNN inference throughput in frames per second (fps) for ARM CL, Tensorrt, and our methodology, respectively. Columns 2 and 4 in Table 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, and, therefore, does not efficiently exploit the data-level parallelism, available in the CNN. Columns 3 and 4 in Table 2 show, that our methodology achieves up to 20% 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.

Table 2. Experimental results, average over 100 runs

CNN	CNN inference throughput (fps)		
	ARM CL	Tensorrt	Our methodology
bvlc alexnet	3.2	106	112
VGG 19	1.2	13	18
bvlc googlenet	5	115	140
tiny yolo v2	2.6	38	45
inception v1	3.2	115	136
resnet18	7.7	138	143
densenet121	3	52	72
Emotion FER	55	325	401

6 Conclusions

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 a NVIDIA Jetson embedded CPUs-GPUs MPSoC. The evaluation results show, that taking real-world CNNs from the ONNX models zoo and mapping them on a NVIDIA Jetson MPSoC, a 20% 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 NVIDIA Jetson MPSoCs.

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

References

1. Alom, Md.Z., et al.: The history began from AlexNet: a comprehensive survey on deep learning approaches. CoRR, abs/1803.01164 (2018)
2. Diamant, A., et al.: Deep learning in head and neck cancer outcome prediction. Sci. Rep. **9**, 27–64 (2019)
3. Do, T., et al.: Real-time self-driving car navigation using deep neural network. In: GTSD, pp. 7–12 (2018)
4. Shvets, A., et al.: Automatic instrument segmentation in robot-assisted surgery using deep learning. bioRxiv (2018)
5. Grant, M.: Overview of the MPSoC design challenge. In: DAC (2006)
6. Reinders, J.: Intel Threading Building Blocks. O'Reilly & Associates Inc., Sebastopol (2007)
7. Siqi, W., et al.: High-throughput CNN inference on embedded ARM big. LITTLE multi-core processors. IEEE TCAD **39**, 225–2267 (2019)
8. Abadi, M., et al.: TensorFlow: large-scale machine learning on heterogeneous systems. http://tensorflow.org/ (2015)
9. Jia, Y., et al.: Caffe: convolutional architecture for fast feature embedding. In: MM. ACM (2014)
10. Parvat, A., et al.: A survey of deep-learning frameworks. In: ICISC (2017)
11. Song, L., et al.: HyPar: towards hybrid parallelism for deep learning accelerator array. In: HPCA, pp. 56–68 (2019)
12. NVIDIA TensorRT framework. https://developer.nvidia.com/tensorrt
13. Singh, A., et al.: Energy-efficient run-time mapping and thread partitioning of concurrent OpenCL applications on CPU-GPU MPSoCs. ACM Trans. Embed. Comput. Syst. **16**, 147:1–147:22 (2017)
14. Ando, Y., Shibata, S., Honda, S., Tomiyama, H., Takada, H.: Automated identification of performance bottleneck on embedded systems for design space exploration. In: Schirner, G., Götz, M., Rettberg, A., Zanella, M.C., Rammig, F.J. (eds.) IESS 2013. IAICT, vol. 403, pp. 171–180. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38853-8_16
15. Kang, D., et al.: C-GOOD: C-code generation framework for optimized on-device deep learning. In: ICCAD (2018)
16. Huynh, L.N., et al.: DeepSense: a GPU-based deep convolutional neural network framework on commodity mobile devices. In: WearSys@MobiSys (2016)
17. Huynh, L., et al.: DeepMon: mobile GPU-based deep learning framework for continuous vision applications. In: MobiSys (2017)
18. Tang, L., et al.: Scheduling computation graphs of deep learning models on manycore CPUs. arXiv:abs/1807.09667 (2018)
19. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. Proc. IEEE **75**, 1235–1245 (1987)
20. Sastry, K., et al.: Genetic algorithms. In: Burke, E.K., Kendall, G. (eds.) Search Methodologies, pp. 97–125. Springer, Boston (2005). https://doi.org/10.1007/0-387-28356-0_4
21. Bilsen, G., et al.: Cyclo-static dataflow. IEEE Trans. Sig. Process. **44**, 397–408 (1996)
22. ONNX models zoo. https://github.com/onnx/models
23. NVIDIA Jetson TX2. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2

24. Abadi, M., et al.: A computational model for TensorFlow: an introduction. In: MAPL. ACM (2017)
25. Ha, S., Teich, J.: Handbook of Hardware/Software Codesign. Springer, Dordrecht (2017). https://doi.org/10.1007/978-94-017-7358-4
26. Even, S.: Graph Algorithms, 2nd edn. Cambridge University Press, Cambridge (2011)
27. ARM compute library. https://github.com/ARM-software/ComputeLibrary