This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3237572

Automated Exploration and Implementation of Distributed CNN Inference at the Edge

Xiaotian Guo, *Student Member, IEEE,*
Andy D. Pimentel, *Senior Member, IEEE,* and Todor Stefanov, *Member, IEEE*

***Abstract*—For model inference of Convolutional Neural Networks (CNNs), we nowadays witness a shift from the Cloud to the Edge. Unfortunately, deploying and inferring large, compute- and memory-intensive CNNs on Internet-of-Things devices at the Edge is challenging as they typically have limited resources. One approach to address this challenge is to leverage all available resources across multiple edge devices to execute a large CNN by properly partitioning it and running each CNN partition on a separate edge device. However, there currently does not exist a design and programming framework that takes a trained CNN model as input and subsequently allows for *efficiently exploring* and *automatically implementing* a range of different CNN partitions on multiple edge devices to facilitate distributed CNN inference. Therefore, in this paper, we propose a novel framework that automates the splitting of a CNN model into a set of sub-models as well as the code generation needed for the distributed and collaborative execution of these sub-models on multiple, possibly heterogeneous, edge devices, while supporting the exploitation of parallelism *among* and *within* the edge devices. In addition, since the number of different CNN mapping possibilities on multiple edge devices is vast, our framework also features a multi-stage and hierarchical Design Space Exploration methodology to efficiently search for (near-)optimal distributed CNN inference implementations. Our experimental results demonstrate that our work allows for rapidly finding and realizing distributed CNN inference implementations with reduced energy consumption and memory usage per edge device, and under certain conditions, with improved system throughput as well.**

***Keywords*-edge computing; internet of things; deep learning; distributed inference; design space exploration.**

I. Introduction

Deep learning (DL) [1] has become a popular method in AI-based applications in various fields including computer vision, natural language processing, automotive, and many more. Especially, DL approaches based on convolutional neural networks (CNNs) [2] have been extensively utilized because of their huge success in image classification [3] and speech recognition applications [4].

Due to the high complexity of state-of-the-art CNN models, the training of these models is performed mainly on high-performance platforms, while the model inference is usually provided as a cloud service [5], allowing less powerful Internet-of-Things (IoT) devices at the Edge to easily use such services. Realizing CNN inference on edge devices using cloud services, however, requires users to communicate a substantial amount of data between an edge device and a cloud server. Such data communication may cause data privacy concerns as well as low device responsiveness due to data transmission delays or temporal unavailability of cloud services. Evidently, this is highly undesirable for those CNN-based applications that are particularly sensitive to compute response delays or the privacy of the processed data. For example, CNN-based navigation in self-driving cars [6] cannot tolerate variable and large response delays occurring due to the communication between the car and a cloud server. Or, applications in healthcare [7] using CNNs on IoT devices dealing with patient data cannot send their data to the cloud because this could lead to leakages of private data and violation of patients' privacy rights. The aforementioned concerns motivate the shift of the CNN inference from the Cloud to the Edge. When entirely executed at the Edge, a CNN is deployed close to the source of data, and data communication with a cloud server is not required, thereby ensuring high application responsiveness and reducing the risk of private data leakage.

Unfortunately, deploying and inferring a large CNN, which is typically memory/power-hungry and compute-intensive, on an IoT edge device is challenging because many edge devices have limited energy budgets and compute and memory resources. One approach to address this challenge is to construct a lightweight CNN model from a large CNN model by utilizing model compression techniques (e.g., pruning [8], quantization [9], knowledge distillation [10]), thereby reducing the CNN model size to a degree that allows the CNN to be deployed and efficiently executed on a resource-constrained edge device. However, the accuracy of the compressed CNN model is significantly decreased if high compression rates are required. Another approach is to infer only part of a large CNN model on the edge device and the rest on the cloud by efficiently partitioning the model and distributing the partitions *vertically* along the edge-cloud continuum [11]. However, the aforementioned edge device responsiveness and private data leakage issues are still inevitable in such partitioned CNN inference due to the partial involvement of the cloud. Finally, a third approach to address the challenge is to leverage all available resources *horizontally* along multiple, possibly heterogeneous, edge devices to deploy and execute a large CNN by properly partitioning the CNN model and running

Xiaotian Guo is with the Informatics Institute at the University of Amsterdam and the Leiden Institute of Advanced Computer Science at Leiden University, The Netherlands. (e-mail: x.guo3@uva.nl).

Andy D. Pimentel is with the Informatics Institute at the University of Amsterdam, Amsterdam, The Netherlands. (e-mail: a.d.pimentel@uva.nl).

Todor Stefanov is with the Leiden Institute of Advanced Computer Science at Leiden University, Leiden, The Netherlands. (e-mail: t.p.stefanov@liacs.leidenuniv.nl).

Copyright (c) 20xx IEEE. Personal use of this material is permitted. However, permission to use this material for any other purposes must be obtained from the IEEE by sending a request to pubs-permissions@ieee.org.

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Universiteit Leiden. Downloaded on January 18,2023 at 16:01:14 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3237572

each CNN partition on a separate edge device. The size of each CNN partition should match the limited energy, memory, and compute resources of the edge device the partition runs on. Such an approach not only makes it possible to deploy large CNN models without the need of model compression, respectively without loss of accuracy, but it also resolves the aforementioned responsiveness and privacy issues because a cloud server is not involved in the CNN inference. Thus, in this paper, we focus on this last approach, i.e., entirely distributing and executing a large CNN model at the Edge.

Although distributing, deploying, and executing a large CNN model on multiple IoT edge devices is a desirable and beneficial approach, currently, it requires a significant manual design and programming effort involving advanced skills in CNN model design, embedded systems and programming, and parallel programming for (heterogeneous) distributed systems. At this moment, no design and programming framework exists that fully automates these tasks. Moreover, such distributed execution of the CNN model inference often needs to take multiple requirements into account as well, like latency, throughput, resource usage, power/energy consumption, etc. Here, the way how the different CNN layers are distributed and mapped onto the edge devices plays a key role in optimizing/satisfying these requirements. As today's CNN models are becoming increasingly deep and complex, the number of different CNN mapping possibilities when deploying multiple edge devices, and the various compute resources in each of them, is vast. Therefore, efficient Design Space Exploration (DSE) methods are essential to find a set of (near-)optimal CNN mappings subject to one or more design requirements (i.e., objectives).

To address the above needs, this paper presents a novel framework that allows for *efficiently exploring* and *automatically implementing* a, possibly large, range of different CNN partitions/mappings on multiple edge devices to facilitate distributed CNN inference at the Edge. The framework consists of two main components, namely a multi-stage hierarchical DSE methodology for efficient exploration of CNN mappings and the AutoDiCE tool for fully automated implementation of a CNN mapped on multiple edge devices. The multi-stage hierarchical DSE methodology deploys a tailored Genetic Algorithm (GA) as the underlying search engine and also leverages the AutoDiCE tool to assess the quality (in terms of inference throughput, memory footprint, and energy consumption) of particular CNN mapping implementations. At every stage, DSE is performed at two hierarchical levels. In the first level, analytical models are used inside a GA to approximate each objective function (i.e., throughput, memory, and energy consumption) to avoid relatively long evaluation times through real on-device (i.e., on-board) measurements using AutoDiCE. The near-optimal solutions found in the first level together with Pareto-optimal solutions from a previous DSE stage are utilized as a starting point for the second level DSE. In this second level, we further search and evaluate design solutions using real measurements taken from AutoDiCE-generated CNN inference implementations to determine the Pareto front for the next DSE stage. The output of the last DSE stage provides the final Pareto-optimal solutions in the form of AutoDiCE-based distributed CNN implementations. An initial version of this multi-stage hierarchical DSE methodology has been presented in [12]. The AutoDiCE tool used in the DSE methodology takes as input a specific DSE solution candidate, i.e., a trained CNN model and a CNN partitioning/mapping specification, and subsequently performs automated splitting of the CNN model into a set of sub-models and automated code generation for distributed and collaborative execution of these sub-models on multiple, possibly heterogeneous, edge devices. Doing so, it supports the exploitation of parallelism *among* and *within* the edge devices.

Our novel contributions can be summarized as follows:

- A tool, called AutoDiCE, featuring automated splitting of a CNN model into a set of sub-models and automated code generation for distributed and collaborative execution of these sub-models on multiple, possibly heterogeneous, edge devices. AutoDiCE is the first fully automated tool for distributed CNN inference over multiple resource-constrained devices at the Edge. It is open-source and available at [13];
- A hybrid MPI and OpenMP code generation approach in AutoDiCE to support the exploitation of parallelism *among* and *within* the edge devices, i.e., the latter exploiting multi-core execution;
- A highly flexible AutoDiCE implementation that facilitates easy specification and reuse of existing CNNs (via the ONNX format [14]), and can target a range of (heterogeneous) edge devices via a custom inference engine library which supports a variety of CPUs (x86, ARM), GPUs (NVIDIA, Mali, AMDRX) and GPU APIs (VULKAN, CUDA);
- An advanced DSE methodology for efficiently exploring distributed CNN implementations at the edge, using i) analytical models to approximate each objective function and to prune the design space that is evaluated with AutoDiCE implementations and on-board measurements, ii) multiple DSE stages where at each stage only a specific part of the design space is considered of which the Pareto-optimal solutions from a previous DSE stage are used to find Pareto-optimal solutions in a next DSE stage, and iii) a GA with a tailored chromosome encoding method to scale down the search space;
- A range of experiments in which we show that our framework, composed of the multi-stage hierarchical DSE and AutoDiCE, can rapidly explore and realize a wide variety of distributed CNN inference implementations on multiple edge devices, achieving improved (i.e., reduced) per-device energy consumption and per-device memory usage, and under certain conditions, improved system (inference) throughput as well.

The remainder of the paper is organized as follows. Section II discusses related work, after which Section III presents our AutoDiCE tool. Section IV discusses our multi-stage hierarchical DSE methodology for efficient CNN mapping exploration, which leverages the AutoDiCE tool. In Section V, we describe a range of experiments, demonstrating that our framework can rapidly explore and realize a wide variety of distributed CNN inference implementations with diverse trade-

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Universiteit Leiden. Downloaded on January 18,2023 at 16:01:14 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3237572

offs regarding energy consumption, memory usage and system throughput. Section VI provides a discussion on the current version of our framework and how it could be further improved in the future. Moreover, we further clarify, with examples, why distributed CNN inference using our novel framework is beneficial in real-world application scenarios when the CNN memory footprint and energy consumption are a concern. Finally, Section VII concludes the paper.

II. Related Work

Today's convolutional neural network (CNN) models for computer vision tasks are becoming increasingly complex. For example, the CNN-based model CoAtNet-7 [15] reaching the top-1 accuracy of 90.88% for the ImageNet dataset has 2.44 billion parameters (weights and biases) which values have to be determined during the training and stored/used during the inference. To train and deploy such large CNN models, parallel or distributed computing is often required. For model training, a common approach to accelerate the training process is to exploit pipeline parallelism. For example, GPipe [16] applies pipeline parallelism by splitting a mini-batch of training data into smaller micro-batches, where different GPUs train on different micro-batches. Another example is PipeDream [17] which partitions the CNN model for multiple GPUs such that each GPU trains a different part of the model. An alternative distributed training approach, motivated by privacy concerns among multiple devices/machines, is federated learning (FL) [18], [19]. FL aims at training a global centralized model with multiple, local datasets on distributed devices or data centers, thereby preserving local data privacy and improving learning efficiency. All of the aforementioned approaches target efficient, distributed training of large CNN models. In contrast, our work presented in this paper focuses on efficient, distributed inference of large CNNs.

Unlike the parallel or distributed CNN training, discussed above, the inference of large CNN models often needs to take multiple requirements into account, such as latency, throughput, resource usage, power/energy consumption, etc. To satisfy these requirements when executing the inference of large CNNs on edge devices, the following two approaches for distributed CNN model inference are typically used: *vertically* and *horizontally* distributed inference.

In *vertically* distributed inference (e.g., [11], [20], [21]), the workload of a large CNN is distributed along the cloud-edge continuum. Such an approach maximizes the utilization of computing resources on edge devices, reduces the computation workload on the cloud, and usually improves the CNN inference throughput. The most common idea in this approach is to obtain a specific small sub-model from or an early-exit branch of the initial large CNN model that runs on the edge device. Only if the inference result of the deployed sub-model/early-exit branch on the edge device is below a certain confidence threshold, the device has to upload its data on the cloud and the CNN inference has to continue on the cloud. Vertical distribution along the cloud-edge continuum still relies on the quality and stability of network connections between the edge device and the cloud server because intermediate results of the small CNN sub-models or early-exit branches may still need to be uploaded to the cloud. This not only suffers from high communication latency but also there is a risk of information leakage. In contrast, our framework achieves lower inference latency by deploying a large CNN model over edge devices without the cloud, and therefore also preserves both data and model privacy to some extent.

In *horizontally* distributed inference (e.g., [22]–[27]), the workload of a large CNN is fully distributed among multiple edge devices. That is, all CNN computations are collaboratively executed at the Edge and there is no dependency on the cloud. Data partitioning and model partitioning are two common methods to horizontally distribute the CNN inference across multiple edge devices. Data partitioning exploits data parallelism among multiple devices by splitting the input/output data to/from CNN layers into several parts while each device executes all layers of a CNN model using only some parts of the data. For example, DeepThings [23] uses the Fused Tile Partitioning (FTP) method for splitting input data frames of CNN layers in a grid fashion to reduce the CNN memory usage per device. The main drawback of the data partitioning method is that an edge device should still be capable of executing all layers of a CNN model which implies that the edge device should be able to store the weights and biases of the entire CNN model. Alternatively, the model partitioning method splits the CNN layers and/or connections of a large CNN model, thereby creating several smaller sub-models (model partitions) where each sub-model is executed on a different edge device [24]. For example, MoDNN [22] splits convolution layers and fully connect layers in the VGG-16 model. In [25], [27], CNN layer connections are split and each CNN layer is treated as a sub-task. These sub-tasks are then mapped to edge devices through a balanced processing pipeline approach. In addition to using data and CNN model partitioning to map large CNNs on resource-constrained edge devices, researchers try to optimize the CNN mapping to improve the inference performance. For example, the methodologies in [27]–[30] propose efficient algorithms to determine partitioning policies that generate efficient CNN mappings in order to improve the performance of cooperative inference over multiple edge devices. However, all of these methodologies typically optimize and evaluate CNN mappings based on analytical models only and consider a limited number of objectives. In contrast, our framework optimizes more objectives, and besides analytical models, it deploys AutoDiCE to evaluate mappings by real on-device measurements.

Distributed inference of large CNN models typically needs to consider a range of different design requirements, such as latency, throughput, resource usage, power/energy consumption, etc. These requirements/objectives can be conflicting, implying that there usually does not exist a single optimal CNN mapping that satisfies all requirements. Usually, multiple solutions (the so-called Pareto optimal solutions) co-exist and the set of all optimal solutions is called the Pareto front. Finding these Pareto-optimal CNN mappings for a given number of edge devices to perform distributed CNN inference under several requirements is addressed in this paper. A popular approach to perform such a search for Pareto-optimal

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Universiteit Leiden. Downloaded on January 18,2023 at 16:01:14 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3237572

solutions is by using multi-objective evolutionary algorithms [31]. More specifically, in the domain of DSE, multi-objective Genetic Algorithms (GAs), such as the Non-dominated Sorting Genetic Algorithm (NSGA-II) [32], are widely used and have demonstrated to produce good results [33]. For instance, [34], [35] use the NSGA-II GA to explore the design space to find improved neural network architectures for CNN-based applications. Our DSE methodology also employs NSGA-II to explore the Pareto-optimal CNN mapping solutions with respect to systems (inference) throughput, maximum memory usage per device, and maximum energy consumption per device. However, NSGA-II can easily get stuck in so-called dominance resistant solutions [36], which are far away from the true Pareto front. Therefore, how to search the optimal CNN mappings for distributed inference using NSGA-II, and efficiently find the Pareto front in the huge search space, are important research challenges. In this paper, we try to address these challenges by devising and using a multi-stage hierarchical DSE methodology based on NSGA-II with a tailored chromosome encoding method. Although our method does not guarantee to completely solve the problem of dominance-resistant solutions, our experiments in Section V-D demonstrate that our method mitigates this problem.

III. The AutoDiCE tool

In this section, we present our AutoDiCE tool, which is deployed in our DSE methodology for efficiently searching for (near-)optimal distributed CNN inference implementations at the Edge. To this end, we describe AutoDiCE as a design flow and explain the main steps in the flow with the help of an illustrative example. First, we provide a high-level overview of the AutoDiCE design flow. Second, we describe AutoDiCE's unified user interface. Next, we explain in detail the main steps in the front-end of the AutoDiCE design flow. Finally, we do the same for the back-end of the flow.

A. Overview

AutoDiCE is a flexible tool that facilitates distributed inference of a CNN model, embedded in an AI application, at the Edge. More specifically, it allows designers and programmers of such CNN-based AI applications to perform, *in a fully automated manner*, CNN model partitioning, deployment, and execution on multiple resource-constrained edge devices. Figure 1 shows the AutoDiCE user interface and design flow where the main steps in the flow are divided into two modules: front-end and back-end.

The interface is composed of three specifications, namely Pre-trained CNN Model provided as an .onnx file, Mapping Specification provided as a .json file, and Platform Specification provided as a .txt file.

The Pre-trained CNN Model specification includes the CNN topology description with all layers and connections among layers as well as the weights/biases that are associated with the layers and obtained by training on a specific dataset using deep learning frameworks like PyTorch, TensorFlow, etc. Many such CNN model specifications in ONNX format [14] are readily available in open-access libraries and can be directly used as an input to AutoDiCE.

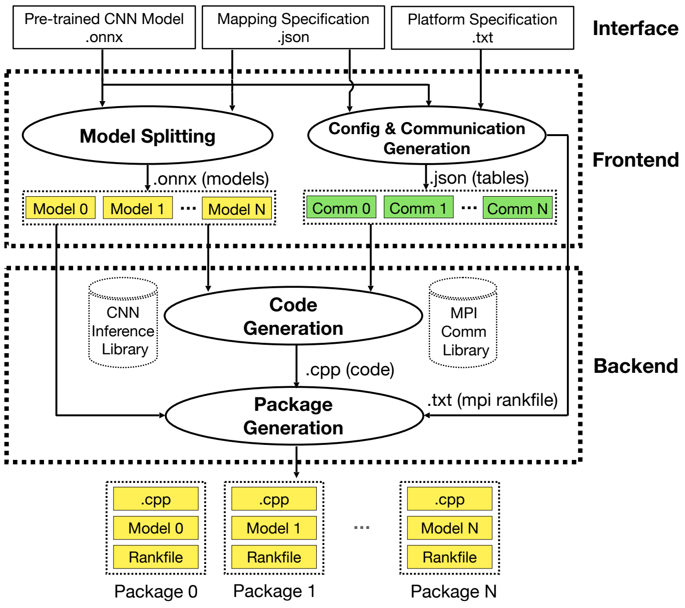


Fig. 1: The AutoDiCE design flow and its user interface

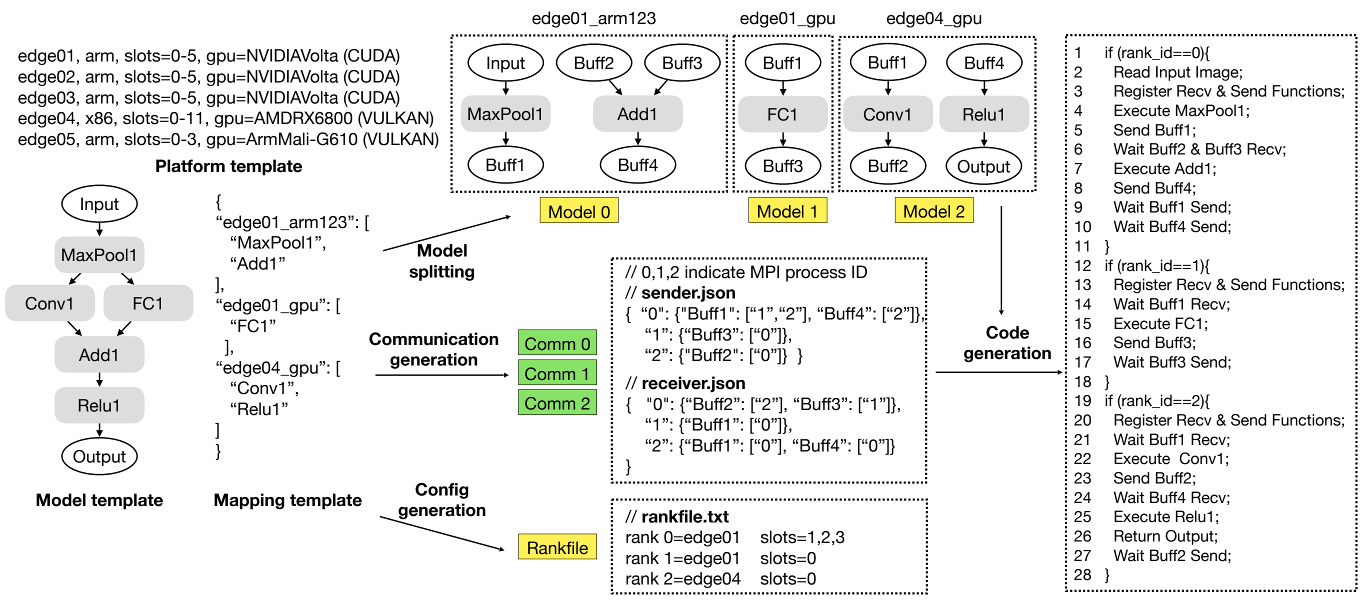
The Platform Specification lists all available edge devices together with their computational hardware resources and specific software libraries associated with these resources. This specification is simple to draw up and can be generated by external tools that query the network connecting the edge devices or provided manually by the user.

The Mapping Specification is a simple list of key-value pairs in JSON format that explicitly shows how all layers described in the Pre-trained CNN Model specification are mapped onto the computational hardware resources listed in the Platform Specification. Every unique key corresponds to an edge device with a selection of its hardware resources to be used for computation. Every value corresponds to a set of CNN layers to be deployed and executed on the edge device resources. Such a Mapping Specification can be provided manually by the user or, like in this paper, generated by a system-level design-space exploration (DSE) tool.

The three aforementioned specifications are given as an input to the front-end module as shown in Figure 1. Two main steps are performed in this module: *Model Splitting* and *Config & Communication Generation*. The Model Splitting takes as an input the Pre-trained CNN Model and Mapping specifications, splits the input CNN model into multiple sub-models, and generates these sub-models in ONNX format. The number of generated sub-models is equal to the number of unique key-value pairs in the Mapping Specification. Each sub-model contains input buffers, output buffers, and the set of CNN layers, specified in the corresponding key-value pair. The Config & Communication Generation step takes all three specification files as an input and generates specific tables in JSON format containing information needed to realize proper communication and synchronization among the sub-models using the well-known MPI interface. In addition, a

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Universiteit Leiden. Downloaded on January 18,2023 at 16:01:14 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3237572



```
edge01, arm, slots=0-5, gpu=NVIDIAVolta (CUDA)
edge02, arm, slots=0-5, gpu=NVIDIAVolta (CUDA)
edge03, arm, slots=0-5, gpu=NVIDIAVolta (CUDA)
edge04, x86, slots=0-11, gpu=AMDRX6800 (VULKAN)
edge05, arm, slots=0-3, gpu=ArmMali-G610 (VULKAN)
```

```
{
"edge01_arm123": [
  "MaxPool1",
  "Add1"
],
"edge01_gpu": [
  "FC1"
 ],
"edge04_gpu": [
  "Conv1",
  "Relu1"
]
}
```

```
// 0,1,2 indicate MPI process ID
// sender.json
{ "0": {"Buff1": ["1","2"], "Buff4": ["2"]},
  "1": {"Buff3": ["0"]},
  "2": {"Buff2": ["0"]} }
// receiver.json
{  "0": {"Buff2": ["2"], "Buff3": ["1"]},
   "1": {"Buff1": ["0"]},
   "2": {"Buff1": ["0"], "Buff4": ["0"]}
}
```

```
// rankfile.txt
rank 0=edge01   slots=1,2,3
rank 1=edge01   slots=0
rank 2=edge04   slots=0
```

```
if (rank_id==0){
  Read Input Image;
  Register Recv & Send Functions;
  Execute MaxPool1;
  Send Buff1;
  Wait Buff2 & Buff3 Recv;
  Execute Add1;
  Send Buff4;
  Wait Buff1 Send;
  Wait Buff4 Send;
}
if (rank_id==1){
  Register Recv & Send Functions;
  Wait Buff1 Recv;
  Execute FC1;
  Send Buff3;
  Wait Buff3 Send;
}
if (rank_id==2){
  Register Recv & Send Functions;
  Wait Buff1 Recv;
  Execute  Conv1;
  Send Buff2;
  Wait Buff4 Recv;
  Execute Relu1;
  Return Output;
  Wait Buff2 Send;
}
```

Fig. 2: AutoDiCE in action: a detailed example

configuration text file (MPI rankfile) is generated to initialize and run the sub-models as different MPI processes.

As shown in Figure 1, the generated configuration file, sub-models, and tables are used in the back-end module for code and deployment package generation. During the *Code Generation* step in this module, efficient C++ code is generated for every edge device based on the input sub-models and tables. In the generated code, primitives from the standard MPI library are used for data communication and synchronization among sub-models as well as primitives from our customized CNN Inference Library are used for implementation of the CNN layers belonging to every sub-model. Both libraries enable the generation of cross-platform code that can be compiled for and executed on multiple heterogeneous edge devices. Finally, the *Package Generation* step packs the generated cross-platform C++ code, the MPI rankfile, and a sub-model together to generate a specific deployment package for every edge device. All packages contain the same C++ code and the same MPI rankfile but different sub-models. When a package is compiled, deployed, and executed on an edge device, the specific sub-model in the package will be loaded and only the part of the code that corresponds to the loaded sub-model will run as an MPI process as specified in the MPI configuration rankfile.

In the following subsections, the interface and the main steps of the AutoDiCE design flow, introduced above, are explained in more detail with the example in Figure 2.

B. Interface

In the left-most part of Figure 2, we show three templates (examples) representing the three specifications of the user interface introduced in Section III-A. By using these example templates, we comprehensively reveal and explain the flexibility of and heterogeneity support in AutoDiCE.

In general, the Platform Specification lists all available edge devices with their computational resources. Every line in the list specifies the name of the edge device, the CPU architecture, the number of CPU cores, and (optionally) a GPU device with its architecture and programming library. For instance, the first line of the platform template in Figure 2 specifies that the name of the device is *"edge01"* with an ARM processor architecture including six cores in total (slots=0-5) and one GPU device with NVIDIAVolta architecture supported by the CUDA library. Through the Platform Specification, a user can easily and flexibly specify alternative heterogeneous hardware platforms including different numbers of edge devices and types of resources. As shown in Figure 2, the user can select different CPU architectures per edge device such as ARM, x86, etc. with different numbers of cores as well as different GPU architectures per edge device such as NVIDIA, Mali, AMDRX, etc. with different GPU programming APIs such as CUDA, VULKAN, etc.

The model template in Figure 2 is an example of a part of a Pre-trained CNN Model specification that visualizes the CNN model topology only. It contains an input layer, five hidden layers (i.e., MaxPool1, Conv1, FC1, Add1, and Relu1), and an output layer. Every hidden layer stores its own parameters (such as weights, bias, etc.) that are not shown in Figure 2. To support interoperability of AutoDiCE with other DL frameworks, we adopt ONNX as the standard format to represent/specify a pre-trained CNN model in the AutoDiCE interface. The choice of ONNX allows users to provide a CNN model designed, trained, and verified in well-known and widely-used frameworks such as TensorFlow [37], PyTorch [38], etc. A large variety of trained CNN models are already available in ONNX format that can be readily utilized by AutoDiCE, allowing easy deployment of these models over multiple edge devices. In addition, the use of the ONNX interface facilitates reproducibility in terms of CNN designs (e.g., CNN topology, used parameters, etc.) and in CNN evaluations (for CNN model accuracy and non-functional characteristics). For example, in experimental evalu-

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Universiteit Leiden. Downloaded on January 18,2023 at 16:01:14 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3237572

ations, users can confidently and reliably compare CNN model characteristics such as accuracy, memory usage, performance, and power/energy consumption, obtained by AutoDiCE, with the same characteristics obtained by other frameworks and approaches, applied on exactly the same CNNs.

As mentioned in Section III-A, the Mapping specification lists several different key-value pairs to describe a distribution of the layers in a CNN model over different computational platform resources. The Mapping template in Figure 2 is an example of such specification. It lists three different key-value pairs. For example, the unique key *"edge01_arm123"* specifies that three ARM CPU cores (i.e., cores 1, 2, and 3) of device *edge01*, described in the Platform specification, are allocated for CNN layers execution. The corresponding value [*"MaxPool1", "Add1"*] specifies that layers MaxPool1 and Add1, described in the Pre-trained CNN model specification, are executed on the allocated three cores. All valid keys must be generated from the Platform Specification to ensure the availability of chosen computational resources. CNN layers can be bound to a single GPU, a single CPU core, or multiple CPU cores. Specifically, if all keys use computational resources of the same device, the distributed inference turns into a multi-threaded execution on a single device. All valid values must be selected from layers of the Pre-trained CNN model, and all CNN layers in that model should be assigned to at least one hardware processing unit (CPU or GPU) to ensure the mapping consistency. The mapping example in Figure 2 is a vertical partitioning, which means that every CNN layer is mapped to a single unique key (device). If a CNN layer is mapped to multiple unique keys, then the layer will be horizontally distributed over multiple computational resources. Users can realize different approaches for splitting (and parallel execution of) a CNN model, namely vertical, horizontal, and using data parallelism (the latter two are not shown in Figure 2). This is done by changing the layer distribution in the Mapping Specification. However, in this paper, we will only focus on vertical partitioning. It is easy and flexible for users (or DSE and other tools, for that matter) to change the CNN model partitioning as well as mapping of partitions to edge devices through selecting different combinations of key-value pairs in the Mapping Specification.

C. Front-end

The front-end module is designed to parse, check, and pre-process all user specifications through its two main steps: Model Splitting and Config & Communication Generation. Model Splitting splits the input CNN model according to the mapping specification and generates several CNN sub-models. Each sub-model will be implemented and executed as an MPI process. Config & Communication Generation generates an MPI-specific configuration file and communication tables based on the three input specification files. At the top center of Figure 2, the model splitting step is illustrated. Based on the three key-value pairs in the Mapping template (specification), the CNN model template is vertically partitioned into three sub-models (*Model 0, Model 1, and Model 2*). The layers of the CNN model mapped on the same edge device resource will be grouped into a single sub-model. For example, the two layers *MaxPool1* and *Add1* are grouped together to form sub-model *Model 0*.

The output of a CNN layer in the initial Model template is the input of its next connected CNN layers. If two connected CNN layers are mapped onto different edge devices or different compute resources (CPU or GPU) within an edge device, i.e., the two layers belong to two different sub-models, the direct connection between these two layers is replaced by one output buffer belonging to one of the sub-models and one input buffer belonging to the other sub-model. These two buffers are used to store and communicate intermediate results between the two CNN layers. For example, the directly connected CNN layers *MaxPool1* and *Conv1* of the Model template in Figure 2 are mapped onto two different edge devices according to the Mapping template. Thus, layer *MaxPool1* belongs to sub-model *Model 0* and layer *Conv1* belongs to sub-model *Model 2*. As a consequence, the direct connection between *MaxPool1* and *Conv1* is replaced by output buffer *Buff1* in *Model 0* and input buffer *Buff1* in *Model 2*.

The Config Generation step is illustrated in the bottom center of Figure 2. It generates an MPI-specific Rankfile which provides detailed information about how the individual MPI processes, corresponding to the generated sub-models, should be mapped onto edge devices, and to which processor/core(s) of an edge device an MPI process should be bound to. In the example in Figure 2, we have three sub-models *Model 0, Model 1, and Model 2* that will be implemented and executed as three different MPI processes 0, 1, and 2, respectively. Based on the Mapping template, the example Rankfile in Figure 2 specifies that the MPI processes 0 and 1 should be mapped onto edge device *edge01* and the MPI process 2 should be mapped onto edge device *edge04*. In addition, each line of the Rankfile specifies the physical processors/cores allocated to the corresponding MPI process. In our example Rankfile, the first line specifies that MPI process 0 should be mapped on edge device *edge01* and slots 1, 2, and 3 are allocated to this process on this device. This means that this process will run on three ARM CPU cores (i.e., core 1, 2, and 3) of device *edge01*.

The Communication Generation step is illustrated in the center of Figure 2. It generates a sender table and a receiver table as .json files. These two communication tables specify the necessary communications between individual MPI processes to ensure that the input/output buffers of the corresponding sub-models are synchronized through the MPI interface. For example, the first line in the sender table specifies that MPI process 0 needs to send the contents of *Buff1* to MPI processes 1 and 2, and the contents of *Buff4* to MPI process 2. Correspondingly, the third line in the receiver table specifies that MPI process 2 needs to receive the contents of *Buff1* and *Buff4*, both from MPI process 0. The communication and synchronization information in the sender and receiver tables ensure that the initial input CNN model is correctly executed after the model splitting.

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Universiteit Leiden. Downloaded on January 18,2023 at 16:01:14 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3237572

D. Back-end

The back-end module constitutes AutoDiCE's final stage to create a CNN-based application for deployment over multiple edge devices. It contains two main steps: Code Generation and Package Generation.

The first step, Code Generation, turns all intermediately generated files (all sub-models and communication tables) by the front-end module into efficient C++ code. The output of this step is a single .cpp file which has a very specific and well-defined code structure, making calls to specific primitives and functions located in two libraries: a standard MPI Library and our customized CNN Inference Library. The code structure contains several code blocks. Each code block is surrounded by an `if` statement and implements one CNN sub-model. The sub-models are executed as individual MPI processes mapped on different edge device resources, meaning that every MPI process runs only the code block implementing the corresponding sub-model. The code block is uniquely identified by a rank ID checked in the `if` statements surrounding the code blocks. Unique rank IDs are assigned according to the Rankfile, explained in Section III-C, during the MPI initialization stage. The pseudo-code template in the right-most part of Figure 2 illustrates the specific code structure of the generated .cpp file. It contains three code blocks, i.e., Lines 1-11, Lines 12-18, and Lines 19-28, that implement sub-models *Model 0, Model 1, and Model 2*, respectively. *Model 0, Model 1, and Model 2* will be executed as three MPI processes 0, 1, and 2, respectively. Every MPI process contains the aforementioned code template but the MPI process 0 corresponding to sub-model *Model 0* will run only the code block between lines 1 and 11. Similarly, the MPI process 1 will run only the code block between lines 12 and 18, etc.

The code blocks themselves all have a similar, well-defined structure starting with code that registers all MPI send and receive primitives (e.g., lines 3, 13, and 20 in Figure 2) followed by MPI_Wait primitives that block the code execution until the necessary data to be processed by CNN layers is received (e.g., lines 6, 14, 21, and 24). Then, code implementing the CNN layers is executed followed by MPI_Send primitives that communicate the output data from a layer to other layers executing in different MPI processes mapped on different edge devices/resources (e.g., lines 7-8, 15-16, 22-23). Finally, MPI_Wait primitives are used to block the code execution until the sent data arrives at the destination (e.g., lines 9, 10, 17, and 27).

Some code blocks have to implement and execute more than one CNN layer because the corresponding CNN sub-models contain multiple CNN layers. Every code block implementing multiple CNN layers has to execute the layers in the order specified by the data dependencies in the input CNN Model template to preserve the functional correctness of the distributed CNN model. For example, the CNN sub-model *Model 0* in Figure 2 is implemented by the code block between lines 1 and 11 in Figure 2. Line 2 reads an image file to prepare the input data for the CNN model. The code in line 3 registers all non-blocking MPI send and receive primitive calls according to the first lines in the sender and receiver tables, explained in Section III-C. In lines 4 and 7, the *MaxPool1* and *Add1* layers are executed one after the other, thereby preserving the order specified in the CNN Model template given in Figure 2. After executing each layer, they store their output data in *Buff1* and *Buff4*, respectively. Line 5 sends the content of *Buff1* to MPI process 1 and MPI process 2 according to the sender table. To allow for overlapping communication with computation, the generated code uses non-blocking MPI_Send primitives that return immediately and will not block the execution. A layer within a code block is executed once its input data is available, i.e., layers are executed in a data-driven fashion. For those layers that read their input data from communication buffers (i.e., data generated by another sub-model, possibly running on a different edge device), MPI synchronization (wait) primitives enforce that layers cannot start execution before their input data is available. For example, this data-driven based execution of layers enforces that the *Add1* layer in *Model 0* can only be executed after the input data in *Buff2* and *Buff3* is available. Such synchronization is realized by the MPI_Wait primitives in line 6 of Figure 2. Line 8 uses the non-blocking MPI_Send primitive again to transfer the content of *Buff4* to MPI process 2. Finally, at the end of the code block, in lines 9-10, two synchronization MPI_Wait primitives are called that are associated with the two asynchronous send requests in lines 5 and 8. All such synchronization primitives are always called at the end of a code block in order to stop the code execution until the corresponding send requests (in this example the requests to send the contents of *Buff1* and *Buff4*) are completed.

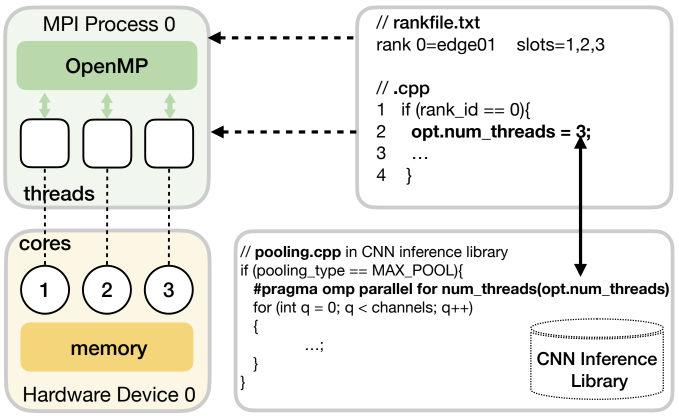
In every code block, the implementation and execution of the CNN layers is realized by calling functions and primitives located in our customized CNN Inference Library. By encapsulating the NCNN [39] and Darknet [40] neural network engines into a uniform wrapper, our custom inference library supports CNN layer implementation and execution on a variety of hardware platforms (e.g., Raspberry Pi with a quad-core ARM v8 SoC, NVIDIA Jetson AGX Xavier series, etc.).

The used MPI primitives in the code blocks are part of the Open MPI library [41], which is an open-source implementation of the standard MPI interface for high performance message passing. It enables parallel execution on both homogeneous and heterogeneous platforms without drastic modifications to the device-specific code.

Besides facilitating the C++ code generation and distributed execution of CNN models (using MPI), our customized CNN Inference Library also integrates and provides OpenMP support. This means that if a CNN layer is mapped onto multiple CPU cores in an edge device, the actual execution of such layer will be multi-threaded using OpenMP in order to efficiently utilize the multiple CPU cores by exploiting data parallelism available within the layer. For example, the *MaxPool1* layer in Figure 2 is implemented and executed as multiple threads within MPI process 0 which is mapped onto the three ARM CPU cores 1, 2 and 3 in edge device *edge01*. More specifically, in Figure 3, we show some details about how the multiple threads bound to the three CPU cores 1, 2 and 3 are executed within MPI process 0. A thread number variable, called *num_threads*, is set to 3 in the code block

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Universiteit Leiden. Downloaded on January 18,2023 at 16:01:14 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3237572



```
// rankfile.txt
rank 0=edge01    slots=1,2,3

// .cpp
if (rank_id == 0){
  opt.num_threads = 3;
  ...
}
```

```
// pooling.cpp in CNN inference library
if (pooling_type == MAX_POOL){
  #pragma omp parallel for num_threads(opt.num_threads)
  for (int q = 0; q < channels; q++)
  {
        ...;
  }
}
```

Fig. 3: MPI process 0 with OpenMP

implementing MPI process 0 during the code generation step. In our customized CNN Inference Library, this variable is used in the implementation code of all types of layers (i.e., convolution, pooling, etc.), and it configures the OpenMP macro line *#pragma omp parallel for* shown in Figure 3. This macro line spawns a group of multiple threads and divides the loop iterations (the `for` loop in Figure 3) that follow this macro line between the spawned threads during the execution. So, during the execution, layer *MaxPool1* is executed as three threads running on CPU cores 1, 2, and 3.

The above discussion on the first step (Code Generation) of the back-end module clearly indicates that AutoDiCE employs a hybrid MPI+OpenMP programming model. OpenMP is used for parallel execution of a CNN layer within an edge device and MPI is used for communication and synchronization among CNN sub-models running on different edge devices or on different compute resources (e.g., CPUs and GPUs) within an edge device. By doing so, AutoDiCE provides extreme flexibility in terms of many alternative ways to distribute the CNN inference within and across edge devices by treating every CPU core or GPU unit in edge devices as a separate entity with its own address space. This allows AutoDiCE to be used in very complex IoT scenarios that may contain a lot of heterogeneous devices.

The second step of the back-end module, i.e. Package Generation, packs the generated .cpp code, sub-models, and Rankfile together into a deployment package for every edge device utilized in the distributed CNN inference. As it is essential to identify the individual MPI process running on an edge device, this step must put the Rankfile in every package. The Rankfile provides detailed information about the MPI processes' binding, which constrains each MPI process to run on specific compute resources of different edge devices. The executable binary (to be deployed on an edge device) will be generated when the corresponding .cpp code in a package is compiled together with the aforementioned customized CNN Inference Library we have developed. As all packages contain the same .cpp code (i.e., we use the Single Program Multiple Data paradigm in this sense), the same binary can be deployed and executed on the same type of edge devices where each edge device will load the corresponding CNN sub-model from its own package before the execution of the binary. For different types of edge devices, we can generate an executable binary for every type.

IV. Multi-stage Hierarchical DSE

In this section, we first describe the set of analytical models, we have devised, to approximate the objectives (throughput, memory usage, and energy consumption) of distributed CNN inference implementations. We use these models in the first level of our multi-stage hierarchical (two-level) DSE methodology to reduce the number of solutions that need to be evaluated using (more costly) measurements on AutoDiCE-generated implementations, which takes place at the second level of DSE. After describing our analytical models, we present the details of all the steps in our multi-stage hierarchical DSE methodology.

A. Analytical Models

We use t_{l_j}, M_{l_j}, E_{l_j} to represent the execution time, the memory usage, and the energy consumption of layer l_j in a CNN model, respectively. A CNN mapping $\mathbf{x}$ is denoted as $\mathbf{x} = [x_1, x_2, \cdots, x_L]$, where L is the number of layers in the CNN model and $x_j = PE_i$ means that layer l_j is mapped on processing element PE_i, which could, e.g., be a CPU or GPU inside an edge device. For a given mapping $\mathbf{x}$, the three objectives of the distributed system can be computed as follows.

1) Throughput: The overall system throughput T_{system} is defined as the images processed per second (img/sec) over multiple PEs:

$$T_{system} = \frac{1}{\max_{1 \le i \le N}(t_i)}$$

$$t_i = \sum_{\forall j: 1 \le j \le L \wedge x_j = PE_i} t_{l_j} + t_i^{comm}$$

where t_i is the time to process one image on PE_i, N is the total number of deployed PEs in the distributed system, and t_i^{comm} is the time needed for data communication related to PE_i. We assume that the size of input images is already determined as well as the input and output tensor shapes of every CNN layer are also fixed and known. Then, we can estimate the total number of operations in every layer and the total size of communicated data related to PE_i. The execution time t_{l_j} is estimated through the number of multiply–accumulate operations (MACs). A proper estimation of communication time t_i^{comm} depends on different data transfers associated with the corresponding PE_i inside an edge device, and involves intra-device data communication via shared memory, intra-device data communication between CPU and GPU, and/or inter-device communication over the network connecting the edge devices in the distributed system.

2) Memory: Every PE_i allocates memory M_i which consists of three parts: memory for CNN coefficients (i.e. weights, bias, and parameters), memory for output buffers to store

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Universiteit Leiden. Downloaded on January 18,2023 at 16:01:14 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3237572

intermediate results of layers, and memory for input buffers of some layers to receive data from other PEs:

$$M_i = \sum_{\forall j: 1 \le j \le L \wedge x_j = PE_i} (M_{l_j}^{coeffs} + M_{l_j}^{outbuffs} + M_{l_j}^{inbuffs})$$

where $M_{l_j}^{coeffs}$, $M_{l_j}^{outbuffs}$, and $M_{l_j}^{inbuffs}$ denote the sizes of the aforementioned memory parts associated with layer l_j mapped on PE_i. These sizes (in number of elements) are estimated based on the type of CNN layer l_j. For example, given a convolutional layer l_j, the memory sizes are calculated as follows:

$$M_{l_j}^{coeffs} = w_{l_j}^{k} * h_{l_j}^{k} * C_{l_j}^{in} * C_{l_j}^{out} + C_{l_j}^{out}$$

$$M_{l_j}^{outbuffs} = w_{l_j}^{out} * h_{l_j}^{out} * C_{l_j}^{out}$$

$$M_{l_j}^{inbuffs} = w_{l_j}^{in} * h_{l_j}^{in} * C_{l_j}^{in}$$

where $w_{l_j}^{k}$ and $h_{l_j}^{k}$ are the width and height of the convolution kernel, $C_{l_j}^{in}$ and $C_{l_j}^{out}$ are the number of input and output channels of layer l_j, and $w_{l_j}^{in}$, $h_{l_j}^{in}$, $w_{l_j}^{out}$, $h_{l_j}^{out}$ are the width and height of the input and output tensors of layer l_j. If layer l_j mapped on PE_i does not receive data from layers that are mapped on other PEs then $M_{l_j}^{inbuffs} = 0$.

3) Energy: Every PE_i consumes energy E_i to execute the CNN layers mapped on PE_i. In our energy consumption analytical model, E_i includes the energy consumed for inference computation and data communication with other PEs:

$$E_i = \sum_{\forall j: 1 \le j \le L \wedge x_j = PE_i} E_{l_j}^{comp} + \sum_{\forall j: 1 \le j \le L \wedge x_j = PE_i} E_{l_j}^{comm}$$

where $E_{l_j}^{comp}$ and $E_{l_j}^{comm}$ denote the computation and communication energy consumption for layer l_j, respectively. Here, $E_{l_j}^{comm}$ has a non-zero value only when layer l_j actually communicates with another PE. We calculate $E_{l_j}^{comp}$ and $E_{l_j}^{comm}$ as follows:

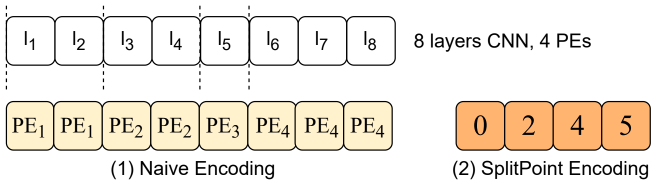
$$E_{l_j}^{comp} = \int_0^{t_{l_j}} P_{l_j}^{comp}(t)\, dt$$

$$E_{l_j}^{comm} = \int_0^{t_{l_j}^{comm}} P_{l_j}^{comm}(t)\, dt$$

where $P_{l_j}^{comp}(t)$ is the power consumption during the execution time t_{l_j} of layer l_j, and $P_{l_j}^{comm}(t)$ is the power consumption during the data communication time $t_{l_j}^{comm}$ of layer l_j with another PE. $P_{l_j}^{comp}(t)$ and $P_{l_j}^{comm}(t)$ are acquired by real measurements during CNN layer profiling on an edge device.

B. DSE Methodology

Our DSE methodology utilizes a Genetic Algorithm (GA), namely the NSGA-II algorithm [32], to search for optimal mappings of (complete) CNN layers to different, distributed edge devices. We assume that each edge device contains a number of internal compute resources (i.e. PEs), like a CPU and GPU, and we map CNN layers directly to these specific PEs within an edge device.



l_1 l_2 l_3 l_4 l_5 l_6 l_7 l_8
8 layers CNN, 4 PEs
PE_1 PE_1 PE_2 PE_2 PE_3 PE_4 PE_4 PE_4
0 2 4 5
(1) Naive Encoding
(2) SplitPoint Encoding

Fig. 4: Two Chromosome Encoding Methods

Given a trained CNN model with L layers, a layer l_j performs a computation operation in the CNN model such as a convolution (Conv), a matrix multiplication (FC), etc. As mentioned in Section IV-A, a mapping $\mathbf{x}$ of the CNN layers onto a total of N PEs is denoted as $\mathbf{x} = [x_1, x_2, \cdots, x_L]$. Such mapping notation $\mathbf{x}$ is typically encoded with the GA's chromosome where $PE_i, i \in [1..N]$ define the gene types in the chromosome. An example of such encoding, called Naive Encoding (NE), is shown in Figure 4. The GA chromosome $[PE_1$,PE_1,PE_2,PE_2,PE_3,PE_4,PE_4,$PE_4]$ encodes an 8-layer CNN ($L = 8$) mapped onto four PEs ($N = 4$), where layers l_1 and l_2 are mapped on PE_1, l_3 and l_4 on PE_2, l_5 on PE_3, and l_6, l_7, l_8 on PE_4. Such naive encoding for CNN mappings is simple and intuitive but it may require exploration of a huge design space because the space size depends exponentially on the number of layers L in a CNN model and L is typically large. Therefore, in our DSE methodology, we propose and utilize a tailored chromosome encoding method, called Split Point Encoding (SPE). It encodes points in a CNN model that partition the model into N groups of CNN layers, where each group consists of consecutive layers and is mapped on one PE. In Figure 4, the Split Point Encoding example encodes the same mapping as the Naive Encoding example. It can be seen that the 8-layer CNN has four split points, visualized with the vertical dashed lines, at positions 0, 2, 4, and 5 determined by the layer index j. Therefore, the GA chromosome using our SPE method is [0, 2, 4, 5] and it encodes four groups of layers each mapped on one PE as follows: 1) for $j \in (0..2]$, l_j is mapped on PE_1; 2) for $j \in (2..4]$, l_j is mapped on PE_2; 3) for $j \in (4..5]$, l_j is mapped on PE_3; 4) for $j > 5$, l_j is mapped on PE_4. The length of our SPE chromosome is equal to the number of PEs which is N, thus SPE requires exploration of a design space which size depends exponentially on N. Since N is typically much smaller than the number of CNN layers L, our SPE method largely scales down the design space and improves the search efficiency compared to the NE method.

Given a trained CNN model and all edge devices with in total N PEs, our DSE methodology searches for Pareto CNN mappings to optimize the three objectives, mentioned in Section IV-A. In Figure 5, we present the general structure of our multi-stage hierarchical DSE methodology. On the left, the K stages in our DSE workflow are depicted, and on the right a zoomed-in view of each stage is provided with the two rectangular boxes showing the two hierarchical levels per stage. We accelerate our DSE process by splitting it into K different stages, where K is the ceiling value of $log_2(N)$. At each stage, we perform a two-level DSE. At both levels,

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Universiteit Leiden. Downloaded on January 18,2023 at 16:01:14 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3237572

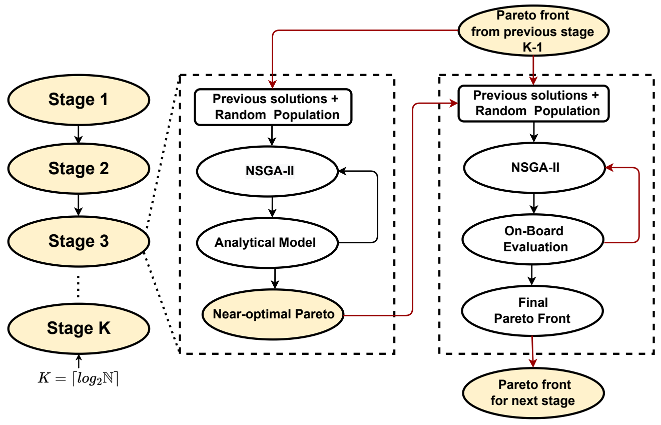

Pareto front from previous stage K-1
Stage 1
Stage 2
Stage 3
Stage K
$K = \lceil log_2 \mathbb{N} \rceil$
Previous solutions + Random Population
NSGA-II
Analytical Model
Near-optimal Pareto
Previous solutions + Random Population
NSGA-II
On-Board Evaluation
Final Pareto Front
Pareto front for next stage

Fig. 5: The DSE Methodology workflow

the NSGA-II GA is deployed to evolve a population of CNN mappings over multiple generations to search for a Pareto front in terms of the targeted objectives. In the first DSE level, we use the analytical models, introduced in Section IV-A, inside the GA to approximate each objective function. In the second DSE level, we use real distributed CNN inference implementations generated by AutoDiCE (see Figure 1) for evaluation, thereby producing more accurate Pareto solutions as they are based on real (on-board) measurements.

At every DSE stage $k \in [1..K-1]$, we search for optimal CNN mappings on 2^k target PEs. Figure 5 shows that to initialize the GA population at stage k, with $k > 1$, the Pareto optimal results found by the previous stage $k-1$ are used. By doing so, we can retain the information of Pareto CNN mappings in previous stages to improve the DSE convergence. Moreover, the second level DSE at each stage also uses the results from the first level of DSE to initialize its population. Finally, the output of the last DSE stage ($k = K$) provides the final Pareto-optimal solutions for N PEs.

V. Framework Evaluation

In this section, we present an evaluation of our proposed framework. First, we describe the setup for our experiments in Section V-A. Then, in Section V-B, we evaluate the execution time of our AutoDiCE tool to show its efficiency. Moreover, we also present a range of experimental results for three representative CNNs to demonstrate that our novel framework, using multi-stage hierarchical DSE and AutoDiCE, can rapidly realize a wide variety of distributed CNN inference implementations with diverse trade-offs regarding energy consumption per device, memory usage per device, and overall system throughput. Subsequently, Section V-C analyzes the effects on the energy consumption per device, the memory usage per device and the overall system throughput when scaling the distributed CNN inference to a varying number of deployed edge devices. Finally, in Section V-D, we evaluate the efficiency of our multi-stage hierarchical DSE methodology by comparing it against one-stage non-hierarchical DSE featuring our Split Point Encoding as well as to more traditional GA-based DSE, i.e., one-stage non-hierarchical DSE with the naive encoding.

A. Experimental Setup

The goal of our experiments is to demonstrate that, thanks to our novel contributions presented in this paper, our framework can rapidly explore and automatically implement CNN partitions over multiple edge devices to realize distributed CNN inference. Moreover, it can do so with lower per-device energy consumption, with smaller per-device memory usage, and under certain conditions, with the same or higher CNN inference throughput, as compared to CNN execution on a single edge device.

In our experiments, we use three real-world CNNs, namely VGG-19 [42], Resnet-101 [43], and Densenet-121 [44], from the ONNX models zoo [45] that take images as an input for CNN inference. These CNNs are used in image classification and are diverse in terms of types and number of layers, and memory requirements to store parameters (weights and biases). The first four columns in Table I list the details of the used CNN models. As these CNNs provide a good layer and parameter diversity, we believe that they are representative and good targets for our evaluations to demonstrate the merits of our framework. The aforementioned CNN models are mapped and executed on a set of up to eight edge devices where all devices are NVIDIA Jetson Xavier NX development boards [46] connected over a Gigabit network switch. Each Jetson Xavier NX device has an embedded MPSoC featuring six CPUs (6-core NVIDIA Carmel ARMv8) plus one Volta GPU (384 NVIDIA CUDA cores and 48 Tensor cores, with a theoretical maximum performance of 844.8 GFLOPS). In our DSE experiments, every CNN layer can be mapped either onto a single CPU core, onto six CPU cores, or onto a GPU inside an NVIDIA Jetson Xavier NX edge device.

As explained in Section IV, the second level in our DSE methodology uses AutoDiCE to evaluate CNN mappings. This means that for the CNN mapping specifications in that DSE level, we apply AutoDiCE to generate and distribute a deployment package for every Jetson Xavier NX device. Subsequently, we measure and collect energy consumption per device, CNN inference throughput, and memory usage per device results, as an average value over 20 CNN inference executions. As the experiments are targeted to embedded devices, the batch size of CNN inference is 1. The inference throughput (measured by instrumenting the code with appropriate timers) and the memory usage per device are reported directly by the code itself during the CNN execution. To measure the energy consumption per device, a special sampling program reads power values from the integrated power monitors on each NVIDIA Jetson Xavier NX board during the CNN execution period, where the power consumption involves the whole board including CPUs, GPU, SoC, etc.

To evaluate the fitness of CNN mappings during DSE using our AutoDiCE tool, the chromosomes inside our GA are translated to the AutoDiCE mapping format described in Section III-B. The GA is executed with a population size of 100 individuals, a mutation probability of 0.2, a crossover probability of 0.5, and performs 400 search generations. For all experiments with the three aforementioned CNNs, the original data precision (i.e., float32) is utilized to preserve the original

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Universiteit Leiden. Downloaded on January 18,2023 at 16:01:14 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3237572

TABLE I: Used CNN models and AutoDiCE execution time breakdown

Network	Total # Layers	Total # Parameters	Memory for Parameters (MB)	AutoDiCE Execution Time (seconds)		
				Front-end	Back-end	Package deployment
DenseNet-121 [44]	910	8.06 million	32	1.93	0.3	21.3
ResNet-101 [43]	344	44.6 million	171	7.30	0.1	23.3
VGG-19 [42]	47	143 million	549	21.50	0.4	26.9

model accuracy of classification.

B. *Efficiency of AutoDiCE and DSE Results*

We start with evaluating the execution time of AutoDiCE itself, to provide insight on how long this tool generally takes to split a CNN model (front-end), to generate the code for the distributed CNN execution (back-end), and to deploy the generated packages to the edge devices for actual execution. To this end, we have measured the required time for each of these phases using the 'worst-case scenario' in the scope of our experiments: using the maximum number of splits in our CNNs to generate sub-models (24 splits/sub-models of a CNN in our experiments), and mapping and deploying the generated sub-models to the maximum number of edge devices (8 in our experiments). These measurements were done on a system equipped with an Intel Core i7-9850H processor, running Ubuntu 20.04.3 LTS. The last three columns in Table I provide a breakdown of the execution time (in seconds) of AutoDiCE for the three CNNs in these worst-case scenarios. From the results in Table I, we can see that AutoDiCE is able to produce executable, distributed CNNs and deploy them on the various edge devices in a relatively short time frame, i.e., in less than a minute for any of the three used CNNs in our worst-case scenario. The comparatively larger execution time of the front-end for VGG-19 is due to the high number of parameters in this model, and the resulting overheads in AutoDiCE of copying these parameters to the large number of sub-models. In any case, these results demonstrate that AutoDiCE allows for rapidly splitting CNNs and deploying them for distributed execution on multiple edge devices.

Our DSE experiments explore a wide range of different CNN mappings and these experiments result in a Pareto front with several Pareto-optimal mappings. In such a set of Pareto-optimal mappings, none of the targeted objectives (energy consumption, throughput, and memory usage) can be further improved without worsening some of the other objectives. More specifically, we consider the *maximum* energy consumption *per device*, *maximum* memory usage *per device*, and total system (CNN inference) throughput as our target objectives. Figures 6a, 6b, and 6c show the Pareto-optimal CNN mappings found by our DSE for DenseNet-121, ResNet-101, and VGG-19, respectively. To better illustrate (the diversity of) these Pareto-optimal mappings, Table II shows more details about a selection of these mappings (points A to I in Figure 6) for comparison. As a reference, the table also includes the mapping results when using a single edge device with 6 CPUs or 1 GPU.

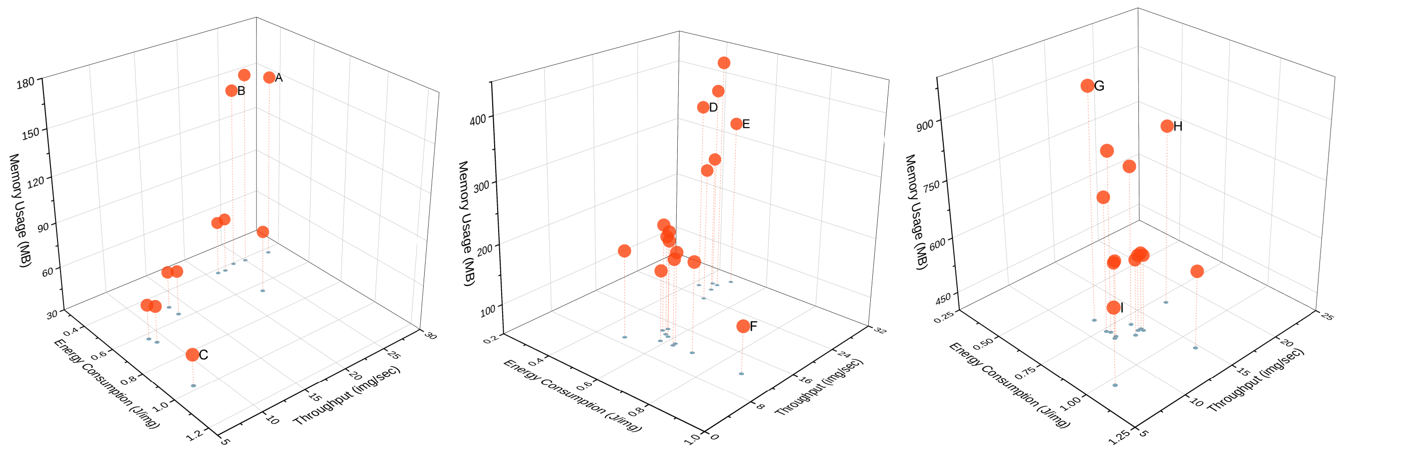
Moreover, to provide a feeling of how the distributed CNN execution on resource-constrained edge devices compares to CNN execution on a (centralized) powerful server, Table II also includes throughput and GPU memory results from an experiment on an NVIDIA GeForce RTX2080 Ti card (4352 NVIDIA CUDA cores and 544 Tensor cores, with a theoretical maximum performance of 13.45 TFLOPS) with Pytorch to mimic a cloud server based execution of the CNNs. Here, we would like to stress that the mimicked cloud server results do not include any latencies required for sending data to and from the cloud server, which would be the case in reality. To make a fair comparison with our experimental edge devices, the inference batch size when using the aforementioned NVIDIA GPU card is also set to 1. We note that it is not possible to precisely measure the energy consumption of the GPU card, thus its energy consumption is not given in Table II. However, its energy consumption is definitely much higher compared to our experimental edge devices. For memory usage, we have taken the peak memory usage of the GPU card because it is influenced by the CNN model and its execution.

Columns 3 and 5 in Table II show the maximum energy consumption per device (in Joules per image) and maximum memory usage per device (in MegaBytes) for a specific CNN mapping, respectively. Column 4 shows the overall system throughput (in images per second). Columns 6, 7 and 8 show the hardware configurations of the selected CNN mappings, consisting of the number of deployed edge devices, and total of CPU cores and GPUs used in these devices, respectively.

From Figure 6 and Table II, we can see that our novel framework allows for easily and rapidly realizing a wide variety of distributed CNN inference implementations with diverse trade-offs regarding per-device energy consumption, per-device memory usage, and overall system throughput. Taking point A as an example, a distributed execution of DenseNet-121 on four devices utilizing only GPUs can reduce the maximum energy consumption per device by 52.5% and 33.8% as compared to the 1-Device CPU and 1-Device GPU hardware configurations, respectively. The system throughput of DenseNet-121 on four devices achieves a 3.5x and 2.2x performance improvement compared to the 1-Device CPU and 1-Device GPU configurations, respectively. In terms of per-device memory usage, the CNN mapping A with four devices consumes 39.3% less memory than the 1-Device GPU implementation, but consumes 17.2% more memory as compared to the 1-Device CPU configuration. Moreover, the distributed CNN inference results in Table II show that for the CNNs with many layers (DenseNet-121 and ResNet-101) comparable performance (throughput) can be obtained as the mimicked powerful cloud server (NVIDIA GeForce

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Universiteit Leiden. Downloaded on January 18,2023 at 16:01:14 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3237572



(a) DenseNet-121 (910 layers) (b) ResNet-101 (344 layers) (c) VGG-19 (47 layers)

Fig. 6: Pareto-optimal CNN mappings from our DSE experiment with three CNNs.

TABLE II: Selected Pareto-optimal Mappings (points) from Figure 6

Network	Points	Max. per-device Energy (J/img)	System Throughput (img/sec)	Max. per-device Memory (MB)	# Edge Devices	# CPU cores	# GPUs
	NVIDIA GeForce RTX 2080 Ti	-	21.339	286.854	-	-	1
	1-Device CPU	0.905	7.987	129.984	1	6	0
	1-Device GPU	0.650	12.807	251.172	1	0	1
DenseNet-121	A	0.430	**27.941**	152.336	4	0	4
	B	**0.408**	23.551	149.941	6	6	5
	C	0.977	7.546	**51.066**	8	38	0
	NVIDIA GeForce RTX 2080 Ti	-	30.823	437.446	-	-	1
	1-Device CPU	1.635	5.786	656.527	1	6	0
	1-Device GPU	1.031	21.767	955.012	1	0	1
ResNet-101	D	**0.425**	26.406	360.766	7	0	7
	E	0.488	**30.048**	329.641	7	12	5
	F	0.886	12.123	**127.883**	8	48	0
	NVIDIA GeForce RTX 2080 Ti	-	166.820	822.902	-	-	1
	1-Device CPU	1.471	7.273	1310.91	1	6	0
	1-Device GPU	1.523	11.664	1666.418	1	0	1
VGG-19	G	**0.680**	11.651	998.273	6	0	6
	H	0.791	**17.385**	868.496	6	6	5
	I	1.035	7.194	**604.504**	7	30	2

RTX2080).

An observation that can be made in general from our DSE results is that by increasing the number of utilized devices, the per-device memory usage is not always reduced if GPUs are deployed within (some of) the devices. In Table II, this is clearly illustrated by, for example, CNN mappings A and B. These mappings have even higher per-device memory usage when distributing the CNN over, respectively, four and six devices as compared to a 1-Device CPU configuration. The higher memory usage when deploying GPUs is due to the fact that an NVIDIA Jetson Xavier NX device has 8GB memory that is shared between CPU and GPU programs. During the loading phase of CNN models, there will typically be at least two copies of the CNN weights when using the GPU: those from the original model file in the host memory, and those initialized as part of the GPU engine.

C. *Varying the Number of Edge Devices*

In Figure 7, we show the effects on the maximum per-device energy consumption, maximum per-device memory usage, and system throughput when scaling the number of deployed edge devices in the distributed CNN execution. Every bar in Figure 7 reflects the best value (energy consumption, memory usage, or throughput) found among all the evaluated mappings, during our DSE experiment, with a specific number of deployed edge devices. This implies that the value reflected by each bar may come from a different Pareto-optimal mapping. For better visualization, all results in Figure 7 have been normalized, where the results for a configuration with one edge device are taken as the reference (i.e., these represent the results of the best-found mappings when targeting a single edge device).

From Figure 7, we can see that, in general, both the per-device energy consumption and the per-device memory usage can be improved (i.e., reduced) when increasing the number of deployed edge devices. Evidently, this is due to the fact that

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Universiteit Leiden. Downloaded on January 18,2023 at 16:01:14 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3237572

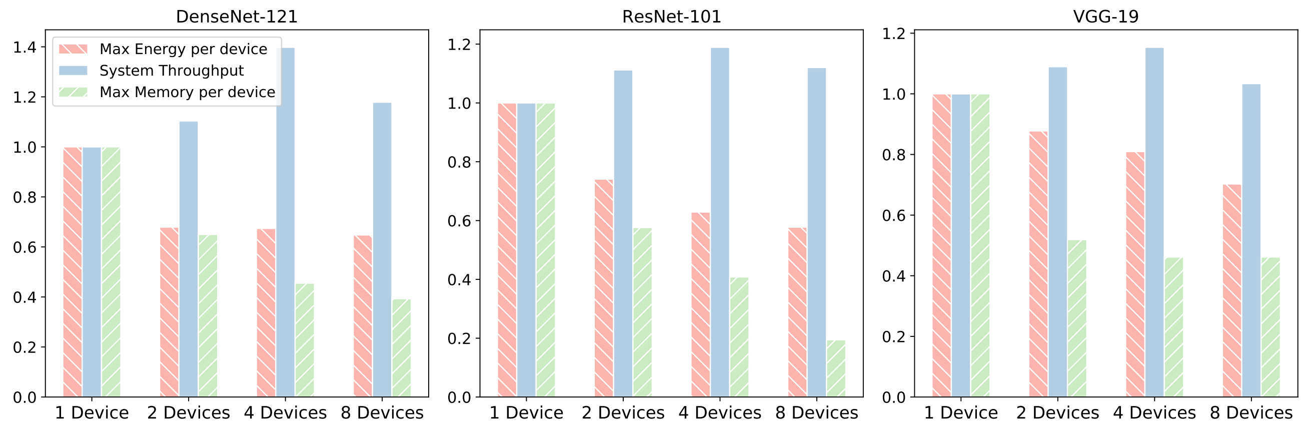


Fig. 7: System throughput and max energy/memory per device when varying the number of edge devices for three CNNs.

the workload (the size and/or the number of executed sub-models) on each participating edge device is reduced when increasing the number of edge devices. Moreover, in some cases, the improvement can be significant. For example, for ResNet-101, the maximum per-device energy consumption and maximum memory usage are reduced by around 40% and 80%, respectively, when distributing the CNN over eight edge devices as compared to execution on a single device. Furthermore, the results in Figure 7 show that the system (CNN inference) throughput can also be improved by means of distributed CNN execution. This is because of the exploitation of pipeline parallelism in the distributed CNN execution. For example, for DenseNet-121, ResNet-101, and VGG-19, the inference throughput increases by up to 38%, 18%, and 18%, respectively when executing the CNN inference on up to four edge devices as compared to a single device. However, the inter-device data communication overheads involved in distributed CNN execution may prevent any further throughput gains, or even cause a slowdown, when scaling the CNN execution to a larger number of edge devices. For example, for all three CNNs, DenseNet-121, ResNet-101, and VGG-19, we see a slowdown in system throughput when scaling the CNN inference from four to eight edge devices.

D. DSE acceleration

To evaluate and demonstrate the search efficiency of our multi-stage hierarchical DSE methodology, we conducted three DSE experiments using the ResNet-101 [43] CNN model and with a slightly smaller cluster of four edge devices. We compare the obtained DSE results in terms of the quality of the found solutions and how this quality changes over time during the DSE process (i.e., the search). In the first DSE experiment, referred as 3s-2l-SPE, we utilize our multi-stage hierarchical DSE methodology as presented in Section IV with 3 stages, 2 levels per stage, and the chromosome is encoded using our SPE method. In the second experiment, referred as 1s-non-SPE, we utilize a classical 1-stage, non-hierarchical DSE methodology based on the NSGA-II algorithm with our AutoDiCE-based on-board evaluation as the fitness function and our SPE as the chromosome encoding method. In the third experiment, referred as 1s-non-NE, we utilize the same DSE methodology as in the second experiment but we replace SPE with the naive encoding (NE) method mentioned in Section IV. In these experiments, every CNN layer can be mapped either onto a 6-core CPU or a GPU present in any of the four edge devices. In each DSE experiment, we run the search for optimal mappings for 70 hours and compare the quality of solutions found within these 70 hours.

Figure 8 shows how the quality of the found mappings in terms of the three targeted objectives improves during the search in the three DSE experiments. The results for each objective are plotted in a separate chart where the X-axis represents the search time in hours and the Y-axis represents the objective value in images per second (img/sec) for the CNN inference throughput, in megabytes (MB) for the maximum memory usage per edge device, and in joules per image (J/img) for the maximum energy consumption per edge device. Every point in a chart represents the best-found mapping with respect to the objective at a given point in time.

The results in Figure 8 clearly indicate that the 1s-non-NE DSE gets easily stuck in dominance resistant solutions, which means that such DSE cannot find high-quality mappings even after hundreds of generations. In contrast, by replacing the common NE encoding method with our tailored SPE method, the search efficiency is significantly improved as shown in Figure 8 where the 1s-non-SPE DSE delivers high-quality mappings for the three objectives after 20 hours. This is because our SPE method ensures that only consecutive CNN layers will be mapped on a PE, thereby scaling down significantly the design space and allowing only exploration of mappings with reduced data communication among PEs. Such mappings are better than less restricted mappings allowed by the NE method.

Finally, comparing the 1s-non-SPE and 3s-2l-SPE results shown in Figure 8, we see that by introducing multiple stages and hierarchy in the DSE process, it further accelerates the finding of high-quality mappings. For example, after 40 hours of search time, our 3s-2l-SPE DSE delivers better mappings for the three objectives than the 1s-non-SPE DSE.

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Universiteit Leiden. Downloaded on January 18,2023 at 16:01:14 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3237572

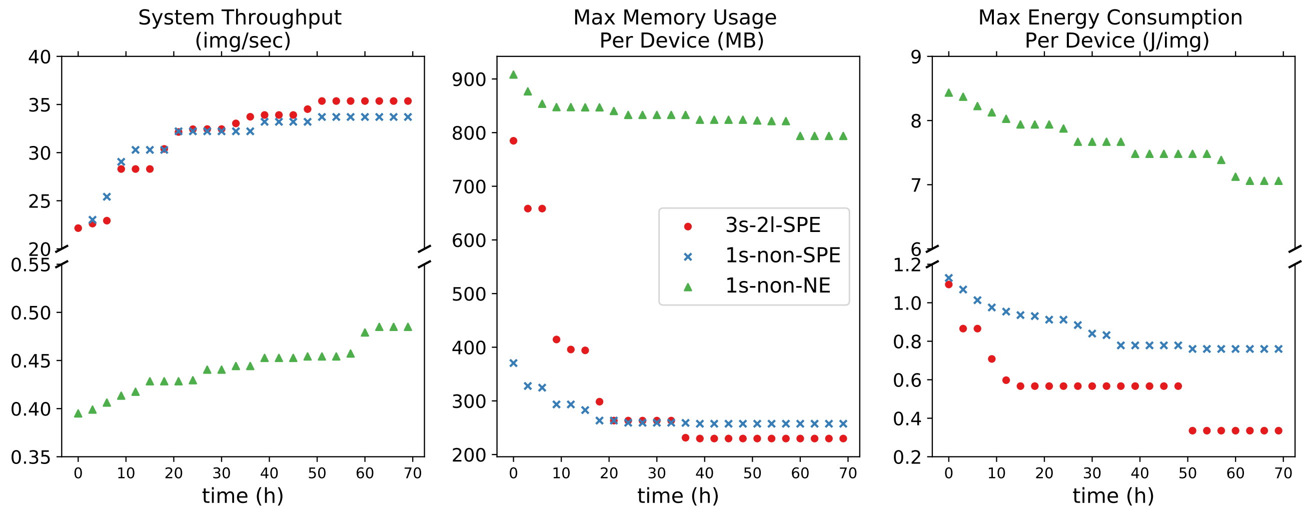


Fig. 8: Quality of found mappings during the three DSE experiments.

VI. Discussion

Our current AutoDiCE tool implementation seeks to provide the greatest flexibility in terms of facilitating distributed execution of CNN models on a wide range of different hardware configurations at the Edge, i.e., configurations different in the number of deployed edge devices as well as in the nature (architecture) of these devices. Therefore, in the current version of AutoDiCE, we have integrated our own customized CNN Inference Library (based on the NCNN [39] and Darknet [40] frameworks) that supports CNN implementation and execution on a variety of hardware platforms (e.g., Raspberry Pi, NVIDIA Jetson, etc.). Our own customized library is not optimized for specific devices in order to provide the greatest possible flexibility. With our focus on flexibility, we have not yet heavily invested in the performance optimization of our AutoDiCE tool when, e.g., targeting specific edge devices. For example, in the future, we plan to integrate the TensorRT framework into AutoDiCE to support very optimized and efficient CNN execution when targeting specific NVIDIA-based devices such as the NVIDIA Jetson series of embedded computing boards because TensorRT has demonstrated to produce superior CNN inference performance on NVIDIA-based devices [47].

Moreover, in our experiments, we have used edge devices that are interconnected using a Gigabit network switch. Evidently, in more realistic edge/IoT settings the connectivity between edge devices might have a lower bandwidth, e.g. using WiFi or other wireless protocols. This would have a detrimental effect on the system throughput objective of distributed CNN inference implementations, possibly leading to more or even purely slowdowns when distributing the inference of a CNN on multiple edge devices. However, we would like to stress that this will not have any impact on the positive effects on (i.e., the reduction of) the per-device energy consumption and per-device memory usage that can always be achieved by distributing CNNs over multiple edge devices.

Finally, since the Jetson NX boards with 16GB of memory used as edge devices in our experiments are sufficiently equipped for executing complete CNNs, one could question why distributed execution would be needed. However, in real-world application scenarios, there are often other running application tasks, besides the CNN execution, on an edge device. In such scenarios, the device memory cannot be fully utilized for the CNN execution, and therefore the available memory may be insufficient for CNN-based applications. If CNN models cannot be mapped on a single device because of memory limitations (either due to memory usage of other application tasks on the device or the fact that the device is less capable than the one we used in our experiments and simply has not enough physical memory), then we have to split the CNN model and execute it on multiple collaborative edge/IoT devices.

Another important reason for distributing CNN execution over multiple edge/IoT devices, even if CNN execution on a single edge/IoT device would be feasible, is when the consumed energy by a single (battery-operated) device does not provide enough ‘lifetime’ for the application mission to be performed. For example, consider an application scenario where a swarm of eight collaborating battery-operated mobile robots has to perform a surveillance mission for 20 hours without recharging the batteries. One of the tasks, among several mission tasks the swarm has to perform, is a continuous on-board CNN-based image processing of a camera-captured video stream using the ResNet-101 CNN model. Every mobile robot in the swarm is equipped with a Jetson NX board (edge device) used for the robot control/navigation and for running tasks related to the mission. Let us assume that the Jetson NX board is powered by a battery with capacity 18000 mAh and output voltage of 19 V. On the one hand, if the CNN-based image processing task of the swarm is assigned to and performed by only one of the robots then, with the aforementioned battery capacity, the execution of the ResNet-101 model on the robot’s Jetson NX edge device can last only for 15.24 hours, thus the swarm will not be able to accomplish the 20-hour mission without battery recharging. This is because the energy consumption per image of ResNet-

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Universiteit Leiden. Downloaded on January 18,2023 at 16:01:14 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3237572

101 executed on Jetson NX is 1.031 J, and after processing 1194181 images with processing time of 45.94 ms per image, the aforementioned battery will be completely discharged. On the other hand, if the CNN-based image processing task of the swarm is assigned to and performed collaboratively by four out of the eight robots in the swarm, i.e., distributing the ResNet-101 CNN model on four Jetson NX edge devices, then the 20-hour mission of the swarm without battery recharging could be accomplished. This is because, according to our results shown in Figure 7 for ResNet-101, the distributed ResNet-101 execution on four edge devices will reduce the energy consumption per device by around 35%, thereby increasing the 'lifetime' of ResNet-101 on a single battery charge with 1.54x to about 23.45 hours.

The real-world application scenarios and example, discussed above, clearly demonstrate the benefits of reducing the per-device memory usage and per-device energy consumption that could be achieved by using our novel framework for distributed CNN inference at the Edge.

VII. Conclusions

In this paper, we have presented a novel framework for efficiently exploring and automatically implementing CNN partitionings on multiple edge devices to facilitate distributed CNN inference at the Edge. To this end, we have introduced a novel multi-stage hierarchical DSE methodology for exploring a wide range of different distributed CNN inference implementations using a variety of edge device resources. To accelerate the DSE process and improve its efficiency, our DSE methodology combines analytical models with real on-board measurements to speedup the evaluations of individual design points and utilizes a tailored chromosome encoding method to effectively scale down the explored design space. To perform the measurement-based evaluations, our DSE methodology leverages the AutoDiCE tool. AutoDiCE is the first fully automated tool for distributed CNN inference over multiple resource-constrained devices at the Edge. It features a unified and flexible user interface, fast CNN model partitioning and code generation, and easy deployment of the CNN partitions on edge devices. We have demonstrated the flexibility of AutoDiCE with a detailed example illustrating all main steps in the AutoDiCE design flow. We have evaluated our novel framework by applying it to three representative CNNs, demonstrating its efficiency and usefulness in facilitating fast and accurate DSE as well as fully automated distributed CNN implementation. Our experiments and results show that our framework, using multi-stage hierarchical DSE and AutoDiCE, can easily and rapidly explore and realize a wide variety of distributed CNN inference implementations on multiple edge devices, achieving improved (i.e., reduced) per-device energy consumption and per-device memory usage, and under certain conditions, improved system (inference) throughput as well. It is worth noting that these improvements are achieved without losing the initial CNN model accuracy because the steps in our framework change neither the CNN layers and their data dependencies nor the values and precision of the CNN parameters (weights and biases).

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[2] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P. Reyes, M.-L. Shyu, S.-C. Chen, and S. S. Iyengar, "A survey on deep learning: Algorithms, techniques, and applications," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–36, 2018.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[4] L. Deng, J. Li, J.-T. Huang, K. Yao, D. Yu, F. Seide, M. Seltzer, G. Zweig, X. He, J. Williams *et al.*, "Recent advances in deep learning for speech research at microsoft," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2013, pp. 8604–8608.

[5] T. Dillon, C. Wu, and E. Chang, "Cloud computing: Issues and challenges," in *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, 2010, pp. 27–33.

[6] K. Patel, K. Rambach, T. Visentin, D. Rusev, M. Pfeiffer, and B. Yang, "Deep learning-based object classification on automotive radar spectra," in *2019 IEEE Radar Conference (RadarConf)*, 2019, pp. 1–6.

[7] Z. Zhou, M. M. R. Siddiquee, N. Tajbakhsh, and J. Liang, "Unet++: A nested u-net architecture for medical image segmentation," in *Deep learning in medical image analysis and multimodal learning for clinical decision support*. Springer, 2018, pp. 3–11.

[8] R. Reed, "Pruning algorithms-a survey," *IEEE transactions on Neural Networks*, vol. 4, no. 5, pp. 740–747, 1993.

[9] Y. Guo, "A survey on methods and theories of quantized neural networks," *arXiv preprint arXiv:1808.04752*, 2018.

[10] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[11] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 615–629, 2017, publisher: ACM New York, NY, USA.

[12] X. Guo, A. D. Pimentel, and T. Stefanov, "Hierarchical Design Space Exploration for Distributed CNN Inference at the Edge," in *Proc. of the Int. Workshop on IoT, Edge, and Mobile for Embedded Machine Learning (ITEM 2022)*, Sept. 2022.

[13] AutoDiCE, "https://github.com/parrotsky/autodice," 2022.

[14] J. Bai, F. Lu, K. Zhang *et al.*, "Onnx: Open neural network exchange," 2019. [Online]. Available: https://github.com/onnx/onnx

[15] Z. Dai, H. Liu, Q. V. Le, and M. Tan, "Coatnet: Marrying convolution and attention for all data sizes," *arXiv preprint arXiv:2106.04803*, 2021.

[16] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems*, vol. 32, pp. 103–112, 2019.

[17] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: Generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–15.

[18] W. Y. B. Lim, N. C. Luong, D. T. Hoang, Y. Jiao, Y.-C. Liang, Q. Yang, D. Niyato, and C. Miao, "Federated learning in mobile edge networks: A comprehensive survey," *IEEE Communications Surveys Tutorials*, vol. 22, no. 3, pp. 2031–2063, 2020.

[19] X. Yin, Y. Zhu, and J. Hu, "A comprehensive survey of privacy-preserving federated learning: A taxonomy, review, and future directions," *ACM Comput. Surv.*, vol. 54, no. 6, jul 2021.

[20] S. Teerapittayanon, B. McDanel, and H.-T. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 328–339.

[21] E. Li, Z. Zhou, and X. Chen, "Edge Intelligence: On-Demand Deep Learning Model Co-Inference with Device-Edge Synergy," *arXiv:1806.07840 [cs]*, Dec. 2018, arXiv: 1806.07840. [Online]. Available: http://arxiv.org/abs/1806.07840

[22] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "Modnn: Local distributed mobile computing system for deep neural network," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1396–1401.

[23] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "DeepThings: Distributed Adaptive Deep Learning Inference on Resource-Constrained IoT Edge Clusters," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2348–2359, Nov. 2018.

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Universiteit Leiden. Downloaded on January 18,2023 at 16:01:14 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3237572

[24] R. Stahl, Z. Zhao, D. Mueller-Gritschneder, A. Gerstlauer, and U. Schlichtmann, "Fully distributed deep learning inference on resource-constrained edge devices," in *International Conference on Embedded Computer Systems*. Springer, 2019, pp. 77–90.
[25] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Toward Collaborative Inferencing of Deep Neural Networks on Internet-of-Things Devices," *IEEE Internet of Things Journal*, vol. 7, no. 6, pp. 4950–4960, Jun. 2020.
[26] R. Stahl *et al.*, "Deeperthings: Fully distributed cnn inference on resource-constrained edge devices," *International Journal of Parallel Programming*, vol. 49, no. 4, pp. 600–624, 2021.
[27] E. Tang and T. Stefanov, "Low-Memory and High-Performance CNN Inference on Distributed Systems at the Edge," in *Proc. of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC)*. ACM, 2021, pp. 1–8.
[28] L. Zhou, M. H. Samavatian, A. Bacha, S. Majumdar, and R. Teodorescu, "Adaptive parallel execution of deep neural networks on heterogeneous edge devices," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 195–208.
[29] L. Zeng, X. Chen, Z. Zhou, L. Yang, and J. Zhang, "Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices," *IEEE/ACM Transactions on Networking*, vol. 29, no. 2, pp. 595–608, 2020.
[30] X. Hou, Y. Guan, T. Han, and N. Zhang, "Distredge: Speeding up convolutional neural network inference on distributed edge devices," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 1097–1107.
[31] K. Deb, *Multi-Objective Evolutionary Algorithms*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 995–1015.
[32] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.
[33] A. Pimentel, "Exploring exploration: A tutorial introduction to embedded systems design space exploration," *IEEE Design & Test*, vol. 34, no. 1, pp. 77–90, 2 2017.
[34] M. Loni, S. Sinaei, A. Zoljodi, M. Daneshtalab, and M. Sjödin, "Deepmaker: A multi-objective optimization framework for deep neural networks in embedded systems," *Microprocessors and Microsystems*, vol. 73, p. 102989, 2020.
[35] S. Minakova, D. Sapra, T. Stefanov, and A. D. Pimentel, "Scenario based run-time switching for adaptive cnn-based applications at the edge," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 21, no. 2, pp. 1–33, 2022.
[36] L. M. Pang, H. Ishibuchi, and K. Shang, "Nsga-ii with simple modification works well on a wide variety of many-objective problems," *IEEE Access*, vol. 8, 2020.
[37] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/
[38] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035.
[39] L. Tencent. (2017) Ncnn. [Online]. Available: https://github.com/Tencent/ncnn
[40] J. Redmon. (2013–2016) Darknet: Open source neural networks in c. [Online]. Available: http://pjreddie.com/darknet/
[41] E. Gabriel *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.
[42] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations*, 2015.
[43] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
[44] G. Huang, Z. Liu, G. Pleiss, L. Van Der Maaten, and K. Weinberger, "Convolutional networks with dense connectivity," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2019.
[45] ONNX. (2022) Onnx model zoo. [Online]. Available: https://github.com/onnx/models
[46] (2020) Nvidia jetson xavier nx. [Online]. Available: https://developer.nvidia.com/embedded/jetson-xavier-nx
[47] B. Ulker, S. Stuijk, H. Corporaal, and R. Wijnhoven, "Reviewing inference performance of state-of-the-art deep learning frameworks," in *Proc. of the 23th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2020, p. 48–53.

Xiaotian Guo received the B.S. degree in applied physics and the M.S. degree in electronic science and technology from the University of Science and Technology of China, Hefei, China, in 2013 and 2016, respectively. He is pursuing the joint Ph.D. degree with the University of Amsterdam and Leiden University. His current research interest includes deep learning at the edge and design space exploration.

Andy D. Pimentel is full professor at the University of Amsterdam where he chairs the Parallel Computing Systems group. His research centers around the design, programming and run-time management of multi-core and multi-processor computer systems. The modeling, analysis and optimization of the extra-functional aspects of these systems, such as performance, power/energy consumption, thermals, reliability but also the degree of productivity to design and program these systems, play a pivotal role in his work. He has an MSc and PhD in computer science from the University of Amsterdam. He is a co-founder of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS). He has (co-)authored more than 130 scientific publications and is an Associate Editor of the Simulation Modelling Practice and Theory journal as well as the Journal of Signal Processing Systems. He is a recipient of the prestigious Test of Time Award from the IEEE/ACM CODE+ISSS 2022 conference. He served as the General Chair of the HIPEAC 2015 conference, as Local Organization Co-Chair of Embedded Systems Week 2015, as Program (Co-)Chair of CODES+ISSS in 2016 and 2017, and as General Chair of DATE in 2024. Furthermore, he has served on the TPC of many leading (embedded) computer systems design conferences, such as DAC, DATE, CODES+ISSS, ICCD, ICCAD, FPL, and LCTES.

Todor Stefanov (S'01–M'05) received the Dipl.Ing. and M.S. degrees in computer engineering from the Technical University of Sofia, Bulgaria, in 1998 and the Ph.D. degree in computer science from Leiden University, The Netherlands, in 2004. He is currently an Associate Professor in the Leiden Institute of Advanced Computer Science at Leiden University and the Head of the Leiden Embedded Research Center (LERC) which is a medium-size research group with a strong track record in the area of system-level modeling and analysis, scheduling and synthesis, programming, and implementation of heterogeneous embedded and cyber-physical systems. He has (co-)authored over 100 scientific papers. His current research interests include several aspects of cyber-physical and embedded systems design, with particular emphasis on system-level design automation for distributed deep learning at the edge, deep learning on heterogeneous resource-constrained embedded systems, multiprocessor systems-on-chip design, and hardware/software co-design. Dr. Stefanov is a recipient of two prestigious awards: The 2022 ACM/IEEE/ESWEEK TEST-OF-TIME AWARD for his CODES+ISSS 2007 conference paper "A Framework for Rapid System-level Exploration, Synthesis, and Programming of Multimedia MP-SoCs" and The 2009 IEEE TCAD DONALD O.PEDERSON BEST PAPER AWARD for his journal article "Systematic and Automated Multi-processor System Design, Programming, and Implementation published in the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD). He has been an editorial board member of the Springer Journal on Embedded Systems and the International Journal of Reconfigurable Computing as well as a guest associate editor of the ACM Transactions on Embedded Computing Systems (2013). He has been General Chair of ESTIMedia 2015 and Local Organization Co-Chair of ESWeek 2015. Moreover, he serves (has served) on the organizational committees of several leading conferences, symposia, and workshops, such as DATE, ACM/IEEE CODES+ISSS, RTSS, IEEE ICCD, FPL, LCTES, IEEE/IFIP VLSI-SoC, ESTIMedia, SAMOS, EUC (as TPC member), and IEEE ESTIMedia, ACM SCOPES (as Program Chair).

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Universiteit Leiden. Downloaded on January 18,2023 at 16:01:14 UTC from IEEE Xplore. Restrictions apply.