Automated Exploration and Implementation of Distributed CNN Inference at the Edge

Guo, X.; Pimentel, A.D.; Stefanov, T.

DOI
10.48550/arXiv.2207.12113
10.1109/JIOT.2023.3237572

Publication date
2023

Document Version
Submitted manuscript

Published in
IEEE Internet of Things Journal

License
CC BY

Link to publication

Citation for published version (APA):
Guo, X., Pimentel, A. D., & Stefanov, T. (2023). Automated Exploration and Implementation of Distributed CNN Inference at the Edge. *IEEE Internet of Things Journal*, *10*(7), 5843-5858. https://doi.org/10.48550/arXiv.2207.12113, https://doi.org/10.1109/JIOT.2023.3237572

AutoDiCE: Fully Automated Distributed CNN Inference at the Edge

Xiaotian Guo
University of Amsterdam, Leiden University
x.guo3@uva.nl

Andy D.Pimentel
University of Amsterdam
a.d.pimentel@uva.nl

Todor Stefanov
Leiden University
t.p.stefanov@liacs.leidenuniv.nl

***Abstract*—Deep Learning approaches based on Convolutional Neural Networks (CNNs) are extensively utilized and very successful in a wide range of application areas, including image classification and speech recognition. For the execution of trained CNNs, i.e. model inference, we nowadays witness a shift from the Cloud to the Edge. Unfortunately, deploying and inferring large, compute- and memory-intensive CNNs on edge devices is challenging because these devices typically have limited power budgets and compute/memory resources. One approach to address this challenge is to leverage all available resources across multiple edge devices to deploy and execute a large CNN by properly partitioning the CNN and running each CNN-partition on a separate edge device. Although such distribution, deployment, and execution of large CNNs on multiple edge devices is a desirable and beneficial approach, there currently does not exist a design and programming framework that takes a trained CNN model, together with a CNN partitioning specification, and *fully automates* the CNN model splitting and deployment on multiple edge devices to facilitate distributed CNN inference at the Edge. Therefore, in this paper, we propose a novel framework, called *AutoDiCE*, for 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, while supporting the exploitation of parallelism *among* and *within* the edge devices. Our experimental results show that AutoDiCE can deliver distributed CNN inference with reduced energy consumption and memory usage per edge device, and improved overall system throughput at the same time.**

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].

The execution of a CNN typically includes two phases: training and inference. During the training phase the optimal CNN parameters (i.e., weights and biases) are established. During the inference phase, a trained CNN is applied to the actual data and performs the task for which the CNN is designed. Due to the high complexity of state-of-the-art CNNs, the training phase is performed mainly on high-performance platforms, while the inference phase is usually provided as a cloud service, allowing less powerful compute devices at the Edge to use such services. Utilizing CNN inference as cloud services requires an edge device to send a substantial amount of data to a cloud server. Subsequently, the cloud server processes the data through the CNN and returns back the CNN result to the device. Such data communication and cloud-based computation increases the risk of data leakage from the edge device and, additionally, may cause low device responsiveness due to data transmission delays or temporal unavailability of the cloud service [5]. Evidently, this is highly undesirable for those CNN-based applications that are particularly sensitive to compute response delays or 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. These delays can lead to incorrect navigation of the car and, subsequently, endanger the life of passengers. Another example is applications in medicine [7] that use CNNs on edge devices to analyse private data of patients. Such applications 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 edge device is challenging because many edge devices have limited power 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, an alternative approach to address the challenge is to leverage all available resources *horizontally* along multiple edge devices to deploy and execute a large CNN by properly partitioning the CNN model and running 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 alternative 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, possibly heterogeneous, 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. More specifically, at this moment, no design and programming framework exists that takes a trained CNN model, together with a CNN partitioning specification, and **fully automates** the CNN model splitting and deployment on multiple edge devices in order to facilitate distributed CNN inference at the Edge. Therefore, in this paper, we propose a flexible framework, called **AutoDiCE**, for 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, while supporting the exploitation of parallelism *among* and *within* the edge devices. To the best of our knowledge this is the first framework that offers the following features:

- a unified interface for specifying a CNN model with Open Neural Network Exchange (ONNX) support [12], the model partitioning, and the target edge devices;
- easy and flexible changing of the CNN model partitioning as well as mapping of partitions onto resources of edge devices;
- automated code generation to adapt to user changes, targeting heterogeneous edge platforms;
- hybrid OpenMP and MPI code generation to support the exploitation of parallelism among and within the edge devices (i.e., exploiting multi-core execution);
- a cross-platform inference engine library that supports GPU acceleration via, e.g., VULKAN and CUDA APIs.

Our AutoDiCE framework is open-source and will be made available to the public at [13].

The remainder of the paper is organized as follows. Section II discusses related work, after which Section III presents our AutoDiCE framework. Section IV describes a range of experiments, demonstrating that AutoDiCE can easily and rapidly realize a wide variety of distributed CNN inference implementations with diverse trade-offs regarding energy consumption, memory usage and system throughput. Section V provides a short discussion on the current version of AutoDiCE and how it could be further improved in the future. Finally, Section VI 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 [14] reaching the highest 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 [15] 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 [16] 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) [17], [18]. 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], [19], [20]), 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.

In *horizontally* distributed inference (e.g., [21]–[24]), 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 [22] 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. 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 [23]. For example, MoDNN [21] splits convolution layers and fully connect layers in the VGG-16 model. In [24], 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.

The aforementioned efforts for horizontally distributed inference focus on performance optimization through partitioning, scheduling and exploiting parallelism, whereas our work is complementary to these efforts as it targets the actual automation of partitioning, code generation, and model deployment for distributed CNN inference at the Edge. Our framework is flexible for users to easily explore objectives of distributed CNN inference at the Edge such as reducing memory usage and energy consumption per edge device, improving CNN inference latency/throughput, etc.

III. The AutoDiCE framework

In this section, we describe our AutoDiCE framework 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 framework 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.

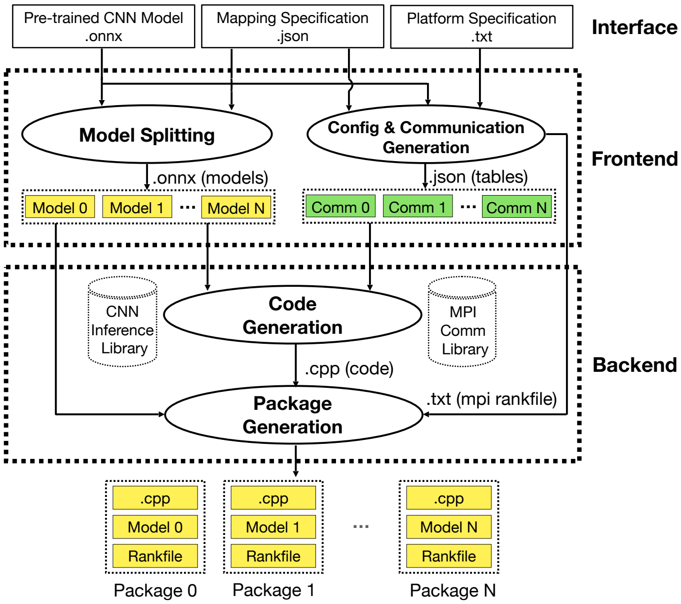


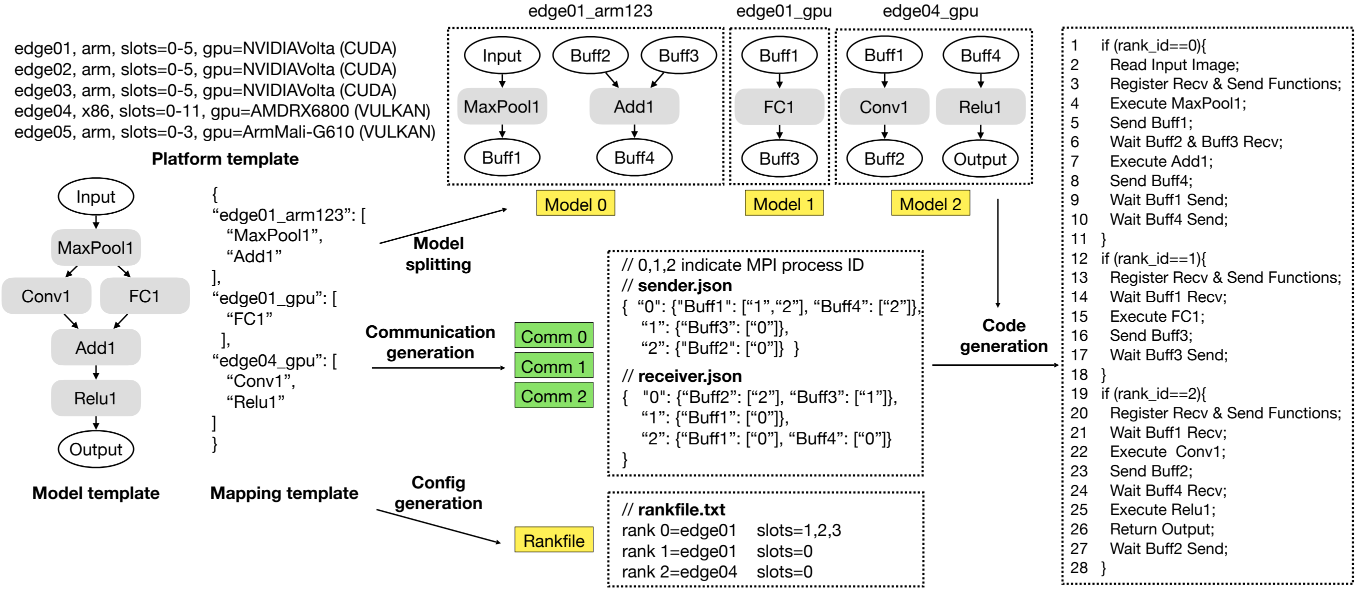
Fig. 1: The AutoDiCE design flow and its user interface

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 [12] are readily available in open-access libraries and can be directly used as an input to our framework.

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 generated by external system-level design-space exploration (DSE) tools or provided manually by the user.

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



```
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

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 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 type 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. In order 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 [25], PyTorch [26], 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 ONNX 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 evaluations, 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. Users can bind CNN layers 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 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 a 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.

D. Back-end

The back-end module constitutes the 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.

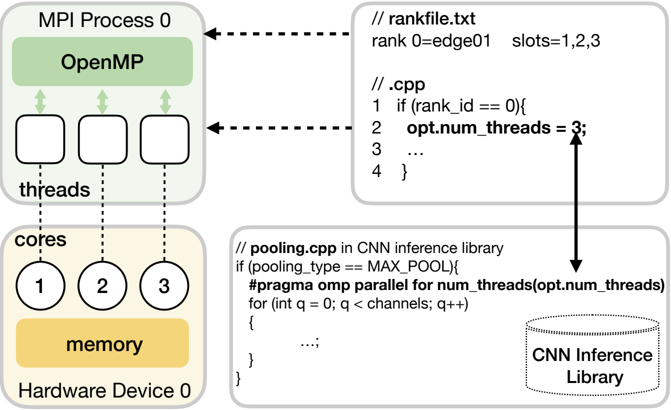
All code blocks have a similar, well-defined structure starting with code that registers all MPI send and receive primitives (e.g., lines 3, 13, and 20 in Figure 2) followed by MPI_Wait primitives that block the code execution until the necessary data to be processed by CNN layers is received (e.g., 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. The used non-blocking MPI_Send primitive returns 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 custom CNN Inference Library. By encapsulating the NCNN [27] and Darknet [28] neural network frameworks 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 [29], 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



```
// 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

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 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 our framework 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, our framework 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 our framework 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 CNN Inference Library. 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. Framework Evaluation

In this section, we present an evaluation of our proposed framework. First, we describe the setup for our experiments in Section IV-A. Then, in Section IV-B, we evaluate the execution time of our framework to show its efficiency. Moreover, we also present a range of experimental results for three representative CNNs to demonstrate that our novel framework 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. Finally, in Section IV-C, we analyze 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.

A. Experimental Setup

The goal of our experiments is to demonstrate that, thanks to our contributions presented in this paper, the AutoDiCE framework can easily and flexibly distribute CNNs over multiple edge devices. Moreover, it can do so with the same or higher CNN inference throughput, with lower per-device energy consumption, and with smaller per-device memory usage as compared to CNN execution on a single edge device. Since state-of-the-art CNNs have deep architectures with many layers, this leads to an immense variety of different CNN mappings on multiple edge devices, each having different characteristics in terms of energy consumption per device, CNN inference throughput, and memory usage per device. Therefore, we have designed a design-space exploration (DSE) experiment, using a Genetic Algorithm (GA), to find Pareto-optimal CNN mappings with respect to CNN inference throughput, energy consumption per device, and memory usage per device.

In our DSE experiment, we use three real-world CNNs, namely VGG-19 [32], Resnet-101 [31], and Densenet-121 [30], from the ONNX models zoo [33] 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 [34] connected over a Gigabit network switch. Each Jetson Xavier NX

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 [30]	910	8.06 million	32	1.93	0.3	21.3
ResNet-101 [31]	344	44.6 million	171	7.30	0.1	23.3
VGG-19 [32]	47	143 million	549	21.50	0.4	26.9

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).

For a given CNN mapping specification, we apply our AutoDiCE framework to generate and distribute a deployment package for every Jetson Xavier NX device. For every implementation generated by AutoDiCE, 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 experiment is 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.

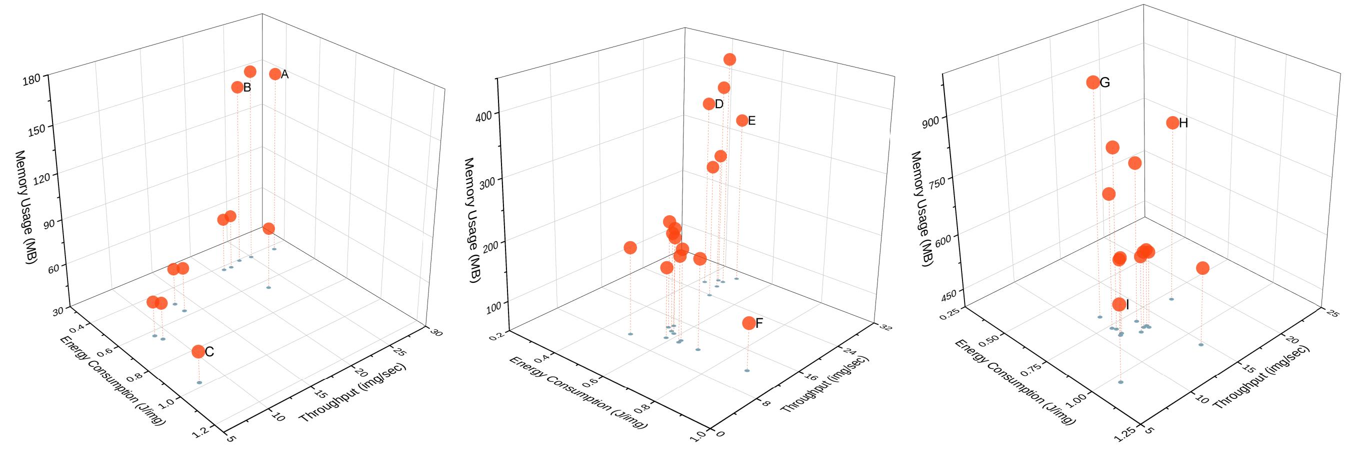
To actually explore the different CNN mappings, while optimizing for the three target objectives (i.e., system throughput, energy consumption per device, and memory usage per device), we apply the well-known Non-dominated Sorting Genetic Algorithm (NSGA-II) [35]. The chromosomes in our NSGA-II multi-objective GA implementation encode how a CNN is split into different segments and how these segments are mapped onto the various edge devices and resources within them. To evaluate the fitness of the encoded CNN mappings using our AutoDiCE framework, the chromosomes are translated to the framework's mapping format described in Section III-B. In our DSE experiment, every CNN layer can be mapped either onto a single CPU core, onto six CPU cores, or onto a GPU inside an edge device. The GA is executed with a population size of 100 individuals, a mutation probability of 0.1, a crossover probability of 0.5, and performs 400 search generations. For all experiments with the three CNNs, the original data precision (i.e., float32) is utilized in order to preserve the original 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 the framework 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 experiment explores a wide range of different CNN mappings and results 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 4a, 4b, and 4c 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 4) for comparison. As a reference, the table also includes the mapping results when using a single edge device with 6 CPUs or 1 GPU. Columns 3 and 5 show the maximum energy consumption per device and maximum memory usage per device for a specific CNN mapping, respectively. Column 4 shows the overall system throughput. Columns 6, 7 and 8 show the hardware configurations of the selected CNN mappings, consisting of the number of deployed edge devices, the total number of used CPU cores and the total number of used GPUs, respectively.

From Figure 4 and Table II, we can see that AutoDiCE 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



(a) DenseNet-121 (910 layers)

(b) ResNet-101 (344 layers)

(c) VGG-19 (47 layers)

Fig. 4: Pareto-optimal CNN mappings from our DSE experiment with three CNNs.

TABLE II: Selected Pareto-optimal Mappings (points) from Figure 4

Network	Points	Max. per-device Energy (J)	System Throughput (FPS)	Max. per-device Memory (MB)	# Edge Devices	# CPU cores	# GPUs
DenseNet-121	1-Device CPU	0.905	7.987	129.984	1	6	0
	1-Device GPU	0.650	12.807	251.172	1	0	1
	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
ResNet-101	1-Device CPU	1.635	5.786	656.527	1	6	0
	1-Device GPU	1.031	21.767	955.012	1	0	1
	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
VGG-19	1-Device CPU	1.471	7.273	1310.91	1	6	0
	1-Device GPU	1.523	11.664	1666.418	1	0	1
	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

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.

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 5, 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 5 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 5 have been normalized, where the results for a configuration with one edge device are taken as the reference (i.e., these represent

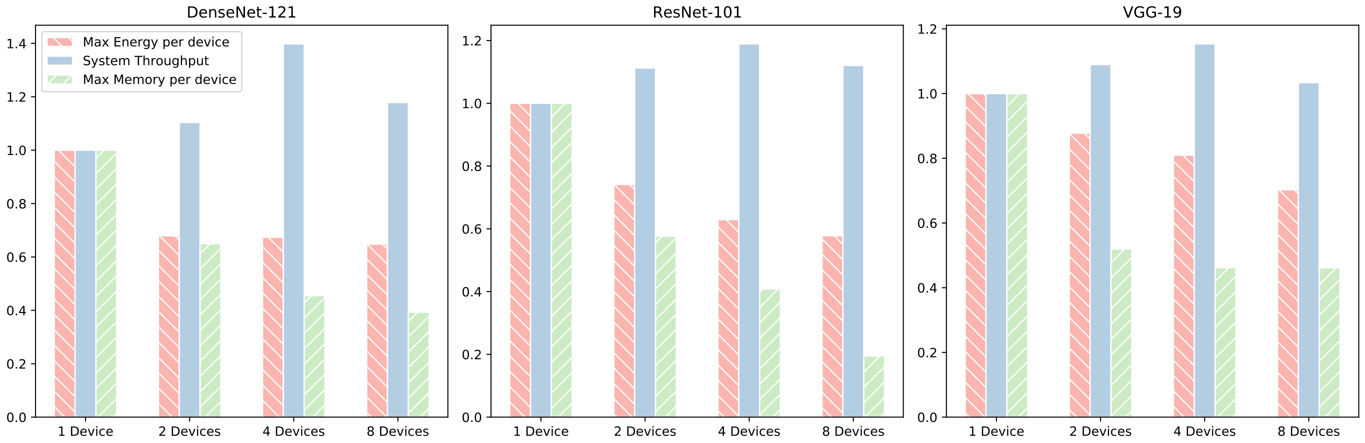


Fig. 5: System throughput and max energy/memory per device when varying the number of edge devices for three CNNs.

the results of the best found mappings when targeting a single edge device).

From Figure 5, 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 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 5 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.

V. Discussion

Our current AutoDiCE framework 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 custom CNN Inference Library (based on the NCNN [27] and Darknet [28] frameworks) that supports CNN implementation and execution on a variety of hardware platforms (e.g., Raspberry Pi, NVIDIA Jetson, etc.). Our own custom 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 framework 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 [36].

VI. Conclusions

In this paper, we have presented AutoDiCE, the first fully automated framework for distributed CNN inference over multiple resource-constrained devices at the Edge. The framework 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. By applying the design flow on three representative CNNs, we have evaluated AutoDiCE in terms of efficiency and usefulness in facilitating fast and accurate Design Space Exploration (DSE). Our DSE experiments and results show that AutoDiCE can easily and rapidly 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 as well as improved system (inference) throughput. 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] J. Bai, F. Lu, K. Zhang *et al.*, "Onnx: Open neural network exchange," 2019. [Online]. Available: https://github.com/onnx/onnx

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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. [Online]. Available: https://doi.org/10.1145/3460427

[19] 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.

[20] 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

[21] 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.

[22] 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. [Online]. Available: https://ieeexplore.ieee.org/document/8493499/

[23] 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.

[24] 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. [Online]. Available: https://ieeexplore.ieee.org/document/8985265/

[25] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[26] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "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. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[27] L. Tencent. (2017) Ncnn. [Online]. Available: https://github.com/Tencent/ncnn

[28] J. Redmon. (2013–2016) Darknet: Open source neural networks in c. [Online]. Available: http://pjreddie.com/darknet/

[29] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "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.

[30] 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.

[31] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.

[32] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations*, 2015.

[33] ONNX. (2022) Onnx model zoo. [Online]. Available: https://github.com/onnx/models

[34] (2020) Nvidia jetson xavier nx. [Online]. Available: https://developer.nvidia.com/embedded/jetson-xavier-nx

[35] 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.

[36] 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.