DIAPASON: Differentiable Allocation, Partitioning and Fusion of Neural Networks for Distributed Inference

Federico Nicolás Peccia*,Alexander Viehl†,Oliver Bringmann‡
FZI Research Center for Information Technology*†‡, University of Tübingen*‡
Germany
peccia@fzi.de*,viehl@fzi.de†,oliver.bringman@uni-tuebingen.de‡

***Abstract*—Concerns in areas such as privacy, energy consumption, climate gas emissions, and costs, push the trend of migrating neural network inference from being executed on the cloud to embedded edge devices. We present our novel approach DIAPASON to overcome restrictions brought on by the computing and application requirements that impede their execution on resource-constrained embedded devices. Our approach addresses these challenges by distributing the inference across multiple computing instances, which could be anything ranging from a multi-CPU configuration on the same SoC to geographically distributed devices. In contrast to recent efforts which tend to apply heuristics to reduce the search space of their problem definition to solve it in a timely fashion, our novel problem definition applies the concept of continuous relaxation to the categorical selection of partitioning, layer fusion, and allocation opportunities. This approach overcomes the problem of the poor exploration of the actual search space that arises when removing potential distribution opportunities during problem simplifications. We conduct numerical simulations and ablation experiments on each one of the configuration parameters of our algorithm by distributing several widely used neural networks. Finally, we compare DIAPASON against the commonly used MoDNN baseline and a state-of-the-art approach, CoopAI, achieving a 44 % and 12 % mean speed-up respectively.**

***Index Terms*—Distributed Inference, Edge Computing, Neural Networks, Deep Learning**

I. Introduction

During the *inference* phase of artificial neural networks (ANN), the already trained model is presented with new inputs, processes them, and returns the corresponding outputs. Although this phase can be executed on the cloud, application-specific requirements and current cloud drawbacks force researchers and industry leaders to consider alternative solutions. Given the huge energy consumption including cooling of the current cloud hardware, the CO_2 equivalent emissions for a system running 24/7 becomes a major concern that needs to be addressed [1]. Another issue of cloud-only solutions is privacy, given that the raw data of a user is being uploaded to an external system, usually managed by a third party. Finally, given the uplink and downlink times provided by cloud systems, it can be challenging to achieve real-time responses.

Those, together with application-specific constraints, are some of the reasons that make inference at the edge using embedded devices so attractive. By using edge devices, the equivalent CO_2 emissions can be significantly improved as these are optimized for low energy consumption. Privacy concerns are also taken into account, as the raw data is processed physically near the point it is being generated, and only already processed data, which is encoded in the form of latent spaces or does not contain specific personal information, is shared with external systems.

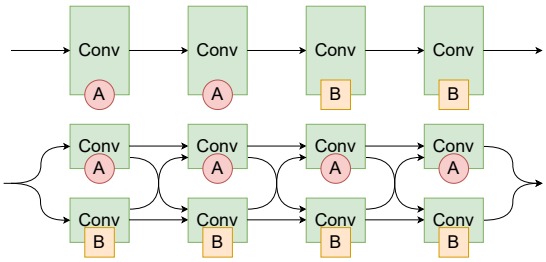

Figure 1: Given two available devices A and B: horizontal/sequential (top) and vertical/parallel (bottom) partitioning.

However, given the memory and computation limits of embedded devices, it becomes challenging to achieve the latency constraints required by different applications. System configurations, like sensors that are physically far away from each other, can also worsen the performance given the high time and energy cost of transmitting the raw data to a centralized edge device. The paradigm known as *distributed inference* (DI) tries to address these issues by a) partitioning the execution of an ANN between multiple devices in a sequential manner, b) parallelising the execution of an ANN across multiple edge devices or c) a combination of both approaches. Fig. 1 presents these two paradigms. A great effort is being put into researching methodologies to find the optimal distribution strategy given a trained ANN, a set of available devices, and the parameters of the communication channels between them [2], [3], [4], [5], [6], [7], [8], [9].

We take inspiration from the concept of *continuous relaxation* proposed by DARTS [10] for Neural Architecture Search (NAS) and apply it to the DI problem. We present DIAPASON: a novel problem definition to represent the distribution problem which can be easily extended to support a wide range of partitioning and fusion opportunities found in the literature. DIAPASON allows us to: 1) encompass multiple

This research was funded by the German Federal Ministry of Education and Research within the project "GreenEdge-FuE", funding no. 16ME0517K.

solutions into the same problem definition, 2) find distribution solutions for both sequential and residual-kind of ANNs, 3) support systems composed of a wide range of devices and communication channels thanks to its generic formulation. To the best of our knowledge, this is the first work to propose continuous relaxation to formulate this problem.

The remainder of this paper is organized as follows. First, Section II gives an overview of current DI approaches. Section III presents the problem definition of DIAPASON and describes how the loss function is built. Then, Section IV gives details about how this problem definition is used to actually obtain the distribution strategy. In Section V, ablation studies of the different parameters of the algorithm are analysed and DIAPASON is compared against the commonly used baseline MoDNN [5] and the state-of-the-art approach CoopAI [6] using numerical simulations. Finally, Section VI concludes the work and presents future expansions for this work.

II. Related work

The first partitioning scheme for DI is the *sequential* partitioning, usually used to improve the throughput of systems composed of two or three devices with very different computing capabilities, like edge-cloud or edge-fog-cloud systems. Auto-Split [2] proposes a nonlinear integer optimization problem for an edge-cloud system, then developed a multi-step search to find a list of potential solutions and select the one that minimizes the latency and satisfies the edge device memory constraints using exhaustive search. AAIoT [3] uses dynamic programming (DP) to minimize the system's response time. JointDNN [4] proposes an approximated solution to optimize latency and energy after formulating the optimization as an integer linear programming (ILP) problem. Given the small complexity of this partitioning scheme, it is usually a common practice to evaluate all partitioning points to find the global optima and report if the proposed algorithm finds this solution.

The second scheme distributes the ANN inference execution by partitioning each layer of the model and executing each partition on a different device in a parallel manner. MoDNN [5] proposes a Biased One-Dimensional Partition (BODP) paradigm to partition the convolutional layers in smaller sublayers, each producing a slice of the original output feature map (*ofm*) inversely proportional to the computing capabilities of each available device. Although this method achieves a good execution load balancing across all devices, it does not include information about the communication channel between them. EdgeFlow [7] uses a progressive model partitioning algorithm to sequentially select the partition decision of each layer approximating it as a linear programming (LP) problem. DistrEdge [11] separates the partitioning decisions into two different modules (the first one using a greedy search and the second one using Deep Reinforcement Learning), so the quality of the distribution strategies found by the second stage is directly related to the initial partitioning points found by the first module. All of these works only consider generating partitions in the height axis of the *ofm* of each layer.

Given the complexity of this partitioning scheme, which can be increased by the number of available devices or techniques like *layer fusion*, it becomes difficult to prove the optimality of the discovered solutions. This is why these works are usually compared against previous baselines (MoDNN is a commonly used one) or against single-device offloading, where the entire ANN is executed on only one device.

Although these methods have been proven successful for a wide range of environment configurations and devices, they still rely heavily on heuristics to limit the search space of their algorithms, or only consider the partition in one axis of the *ofm*, or only support one of both partition schemes.

Additionally, when using parallel schemes, by partitioning previous layers in such a way that each partition generates the exact feature map needed as input for the partitions of the next layer, *chains* or *blocks* of consecutive partitions are generated, thus achieving *layer fusion*. If a complete chain is assigned to the same device, data synchronization between devices can be reduced, with the cost of a greater initial data transmission and a bigger computation load on each device. Several works have used this technique and tried to find the best trade-off between the length of the fused chain and the additional cost inserted by this technique. DeeperThings [12] used Fused Tiled Partitioning (FTP) to divide the *ofm* of convolutional layers in a grid fashion and reduce communication. CoopAI [6], FastCoDNN [8] and DPFP [9] also proposed a fusion problem formulation, and solved it using DP.

DIAPASON was inspired by DARTS [10], which does not focus on DI, but proposes a continuous relaxation of a categorical selection. In DARTS, the goal is to find the best layer configuration for an ANN cell, which can be later used to build an entire ANN. As such, they start with a prototype of their cell, describing the intermediate feature maps of the cell, and leaving the actual layer that transforms these feature maps as unknowns. They propose a set of layers that can be chosen for each transformation and assign a trainable weight to each one of them. During training, the actual *ofm* is a weighted sum of the feature maps generated by each possible layer. At the end of the training, the layer with the greatest weight value is selected as the actual layer for that particular transformation. They have transformed the categorical selection of the layer of each transformation into a continuous relaxation selection.

DIAPASON is not the first to use continuous relaxation to optimize the deployment of ANNs, but it is the first to actually model the allocation, partitioning and fusion of layers using continuous relaxation. [13] used Differential Architecture Search to optimize the latency of the architecture of an ANN for execution on **one** mobile device. [14] used NAS based on DARTS to select the internal structure of the encoder-decoder cells used to compress the data sent between a mobile device and an edge server, but the actual partitioning point is still chosen manually.

III. Problem definition

We define our problem under the assumption that the following inputs are available:

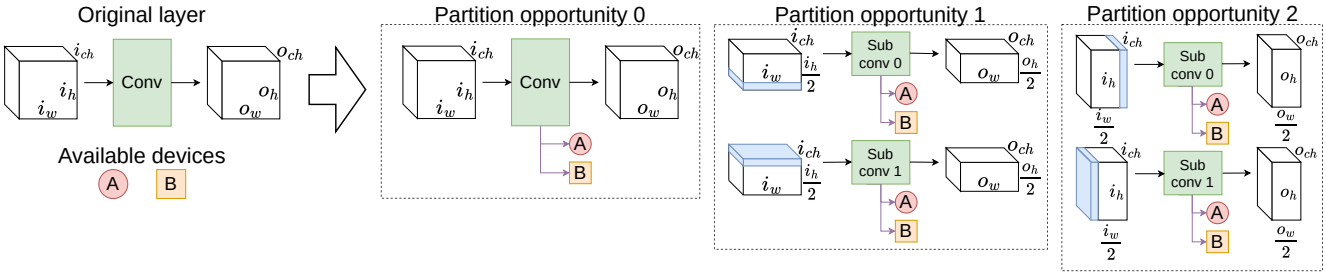

Figure 2: Relationship between an original layer of the model and examples of some of its POs. *ifm* and *ofm* of the original layer and the ones required/generated by the sublayers are also depicted. Blue slices represent the redundant data that is required by both sublayers.

- A Directed Acyclic Graph (DAG) $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ representing the already trained ANN model. Each vertex $v \in \mathcal{V}$ equals a particular layer of the model and each edge $(v_i, v_j) \in \mathcal{E}$ represents data dependencies between layers. Each vertex can have multiple edges that originate from it or end at it, thus enabling the distribution of both sequential and residual kind of ANN architectures.
- A list of available devices $\mathcal{D}$ over which we want to distribute the ANN. Each $d_i \in \mathcal{D}$ provides an estimation function $f : \mathcal{V} \rightarrow \mathcal{R}$ that maps $v \in \mathcal{V}$ to *execution costs* (the term *cost* will be used from now on to talk about latency or energy execution costs). The estimation functions can be obtained by offline profiling [4], simulators [2], [15] or analytical models [6]. In this problem definition, a device can be anything ranging from an embedded device with limited computing capabilities to a powerful cloud server (multiple computation units on the same device, for example, multiple CPU cores on the same SoC, need each to be separately added to $\mathcal{D}$ as an individual device).
- A bandwidth matrix $\mathcal{B} \in \mathcal{R}^{|D|\times|D|}$ representing the cost per element transmission between each one of the available devices in $\mathcal{D}$. We use *cost per element* in order to be agnostic with respect to the bit size used to represent the data in the ANN and enable this approach for both quantized (*int8*) and unquantized (*float32*) networks.
- Both an input and output bandwidth vector $\mathcal{B}_{in/out} \in \mathcal{R}^{|D|}$ representing the cost of moving data from the input sensor and/or to the system that will consume the output.

In DIAPASON, each layer of $\mathcal{V}$ contains a list $\mathcal{P}$ of *partition opportunities* (POs), where $1 \leq |\mathcal{P}| \leq n$ such that $n \geq 1$. Each p contains a list $\mathcal{S}$ of *sublayers*, where $1 \leq |\mathcal{S}| \leq m$ such that $m \geq 1$. Each $s \in \mathcal{S}$ obtains one portion of the original input feature map (*ifm*) as input and calculates only a section of the original layers *ofm*. Each sublayer can be assigned to only one of the devices in $\mathcal{D}$. Fig. 2 presents a representation of the described model.

In order to formulate the problem as a continuous search space, we relax both the categorical choice of the partition for each layer and the device selection of each specific sublayer of each partition in a similar manner to DARTS. In the end, a loss function is generated describing the cost of executing all POs across all devices. This loss function is minimized using gradient descent to obtain the final partition of each layer and the allocation of its sublayers. The generation of the loss function is described in the following sections.

A. Partition opportunity cost

For each layer, $\mathcal{L}_{part_i}$ (1) (2) represents the relaxed cost for PO i. $\mathcal{L}_{sub}$ describes the cost of executing each sublayer on all available devices. $\mathcal{L}_{pen}$ describes a *superposition penalty* to encourage the allocation of sublayers to different devices in order to improve load balancing when optimizing for latency.

$$\mathcal{L}^{lat}_{part_i} = \mathcal{L}^{lat}_{sub} - \mathcal{L}^{lat}_{pen} \tag{1}$$

$$\mathcal{L}^{e}_{part_i} = \mathcal{L}^{e}_{sub} \tag{2}$$

1) Sublayer costs: Each PO has a set of weights $\mathcal{W}_{dev} \in \mathcal{R}^{|\mathcal{S}|\times|\mathcal{D}|}$ which act as device selectors for its sublayers. By applying the *softmax* function to each row of $\mathcal{W}_{dev}$ we obtain values representing the *selection probability* of each available device. $\mathcal{L}^{j}_{dev} \in \mathcal{R}^{|D|}$ is a vector containing the estimated cost of executing sublayer j on each available device, obtained using the estimation function provided for each device. (3) presents the generic sublayer cost $\mathcal{L}_{sub}$.

$$\mathcal{L}_{sub} = \sum_{j=0}^{|\mathcal{S}|} softmax\left(\mathcal{W}^{j}_{dev}\right) \cdot \mathcal{L}^{j}_{dev} \tag{3}$$

2) Superposition penalty: We define a superposition metric between two sublayers i and j as stated in (4), using a scaled euclidean distance as the base comparison function: the similarity function *sim* gets closer to zero the more similar the two weights vectors are, and closer to one the more different they are. V_i is a vector of discount values that represents the superposition between sublayer i and all other sublayers of the partition with $j > i$, where each value V_{ij} is calculated using (5). (6) presents the penalization loss.

$$\mathcal{X}_{ij} = sim\left[softmax\left(\mathcal{W}^{i}_{dev}\right), softmax\left(\mathcal{W}^{j}_{dev}\right)\right] \tag{4}$$

$$V_{ij} = \mathcal{X}_{ij} \cdot min\left(\mathcal{L}_{sub_i}, \mathcal{L}_{sub_j}\right) \tag{5}$$

$$\mathcal{L}_{pen} = \sum_{i=0}^{|\mathcal{S}|-1} minV_i \tag{6}$$

B. Layer costs

The relaxed layer cost for either latency or energy is built by combining the costs of each individual PO as calculated in (1) (or (2)) in a vector $\mathcal{L}_{part} \in \mathcal{R}^{|P|}$. Each layer also has a set of parameters $\mathcal{W}_l \in \mathcal{R}^{|P|}$, which will act as selectors of the

partition with the minimum cost. Similar to (3), (7) present the layer's latency (or energy) cost.

$$\mathcal{L}_l^{lat|e} = softmax\left(\mathcal{W}_l\right) \cdot \mathcal{L}_{part}^{lat|e} \tag{7}$$

C. Data movement costs

To model the data dependencies and cost of moving data between devices, the following costs are defined (generic for both latency and energy costs).

1) Inter-layer costs: For each edge $(v_i, v_j) \in \mathcal{E}$, we need to define a relaxed data movement cost. We first describe the relaxed cost between partitions m of v_i and n of v_j, and then generalize it across all partitions.

For each combination of sublayers of m and n, we generate $\mathcal{W}^{i,j} \in \mathcal{R}^{|\mathcal{D}|\times|\mathcal{D}|}$ (8), which represents the probability of data being sent between any device combination for sublayers i and j. Then, we extract a scalar value representing the overlap between the *ofm* generated by the sublayer of m and the *ifm* needed by the sublayer of n, which is then multiplied by the bandwidth matrix $\mathcal{B}$ obtaining the second term: $\mathcal{L}^{i,j} \in \mathcal{R}^{|D|\times|D|}$ (9). (8) and (9) are multiplied in an element-wise manner to obtain a new matrix representing the relaxed cost for each device combination. By adding together all values in this new matrix, we obtain a relaxed scalar value representing the movement of data from one sublayer to the other. Accumulating these values across all combinations of sublayers of m and n we obtain the final cost to move data between two partitions of different layers (10).

$$\mathcal{W}^{i,j} = softmax\left(\mathcal{W}_{dev_m}^{i}\right) \otimes softmax\left(\mathcal{W}_{dev_n}^{j}\right) \tag{8}$$

$$\mathcal{L}^{i,j} = overlap_{i,j} \cdot \mathcal{B} \tag{9}$$

$$\mathcal{L}_{mov}^{m,n} = \sum_{i=0}^{|\mathcal{S}_n|}\sum_{j=0}^{|\mathcal{S}_m|}\sum_{k=0}^{|\mathcal{D}|}\sum_{l=0}^{|\mathcal{D}|}\left(\mathcal{W}^{i,j} \odot \mathcal{L}^{i,j}\right)_{k,l} \tag{10}$$

The final data movement costs between two consecutive layers a and b can be represented as (11).

$$\mathcal{L}_{mov_{a,b}} = \sum_{m=0}^{|\mathcal{P}_a|}\sum_{n=0}^{|\mathcal{P}_b|} softmax\left(W_l^a\right)_m \cdot softmax\left(W_l^b\right)_n \cdot \mathcal{L}_{mov}^{m,n} \tag{11}$$

2) Input/output costs: We also take into account the cost of moving the data from where it is generated to the devices executing the first layer, and the cost of moving the data from the devices executing the last layer to the system that needs to consume it, proposed by FastCoDNN [8].

The input data movement cost is built similarly to (11). Equation (12) presents the cost of moving data from the input generator to one partition n of the input layer. Equation (13) presents the entire input data movement cost, with l representing the first layer and $\mathcal{L}_{mov_{in}}^{comb} \in \mathcal{R}^{|\mathcal{P}_l|}$ a vector of costs where each element is calculated as stated in (12).

$$\mathcal{L}_{mov_{in}}^{n} = \sum_{j=0}^{|\mathcal{S}_n|} softmax\left(\mathcal{W}_{dev}^{j}\right) \cdot \left(overlap_{in,j} \cdot \mathcal{B}_{in}\right) \tag{12}$$

$$\mathcal{L}_{mov_{in}} = \mathcal{W}_l \cdot \mathcal{L}_{mov_{in}}^{comb} \tag{13}$$

Similarly, equation (14) shows the cost of moving data from one partition n of the output layer to the data consumer, and (15) presents the output data movement cost, with l being the output layer and $\mathcal{L}_{mov_{out}}^{comb} \in \mathcal{R}^{|\mathcal{P}_l|}$ a vector of costs where each element is calculated as stated in (14).

$$\mathcal{L}_{mov_{out}}^{n} = \sum_{j=0}^{|\mathcal{S}_n|} softmax\left(\mathcal{W}_{dev}^{j}\right) \cdot \left(overlap_{j,out} \cdot \mathcal{B}_{out}\right) \tag{14}$$

$$\mathcal{L}_{mov_{out}} = \mathcal{W}_l \cdot \mathcal{L}_{mov_{out}}^{comb} \tag{15}$$

D. Loss functions

(16) describes the continuous loss $\mathcal{L}^{lat|e}$ developed to minimize the execution cost (latency or energy). It is the combination of: 1) the input data movement cost described in Section III-C2; 2) the summation across all layers of the latency cost for a specific layer i (described in Section III-B) added to the cost of moving data from each input layer j from the i layer (described in Section III-C1, with $\mathcal{E}_i \subseteq \mathcal{E}$ representing all edges ending at layer i); 3) the output data movement cost described in Section III-C2.

$$\mathcal{L}^{lat|e} = \mathcal{L}_{mov_{in}}^{lat|e} + \sum_{i=0}^{|\mathcal{V}|}\left(\mathcal{L}_i^{lat|e} + \sum_{j=0}^{|\mathcal{E}_i|}\mathcal{L}_{mov_{i,j}}^{lat|e}\right) + \mathcal{L}_{mov_{out}}^{lat|e} \tag{16}$$

IV. Implementation description

Section III presented how the POs can be used to generate the loss function in a generic way, but which POs are actually modelled is described in this section.

First, each layer is assigned a default partition, which only contains **one** sublayer that generates the entire *ofm*, representing the *non-partitioned* case (first PO in Fig. 2). Then, the second partition is generated, which has as many sublayers as devices are available, to try to parallelize its execution as much as possible. Two ways of generating these sublayers are controlled by a $part_{type}$ flag: 1) naive idea, to partition the *ofm* into equal-height slices and 2) as proposed by [5], partition the *ofm* based on the computing capabilities of the available devices, such that more powerful devices are assigned bigger *ofm* slices in order to improve load balancing.

Each partition, when created, generates a new $\mathcal{W}_{dev}$ set. $weights_{init}$ controls which paradigm is used to initialize these weights: 1) *random initialization*: $\mathcal{W}_{dev}$ of each partition is initialized using a random uniform distribution in range $[0, 1]$; 2) *heuristics initialization*: we assume a heuristic where it is better to distribute each sublayer to a different device to improve parallelism and load balancing, so we initialize $\mathcal{W}_{dev}$ sequentially allocating one sublayer to each available device.

New special partitions are generated to support the fusion of consecutive layers. For each basic PO (from now on called *parent* PO) present in each layer, the fusion POs are recursively generated. In these new special POs, each sublayer generates the slice of the *ofm* needed as input by each corresponding sublayer of the *parent* PO. To enforce the allocation of successive sublayers to the same devices to avoid

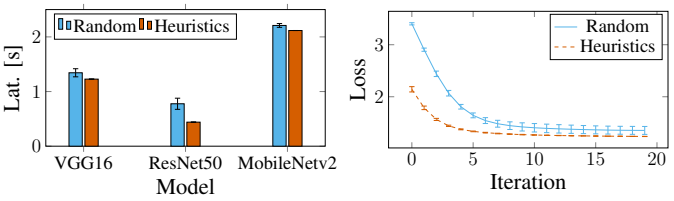

Figure 3: Left: final latency value. Right: loss value during minimization for the VGG16 model (others present a similar behaviour).

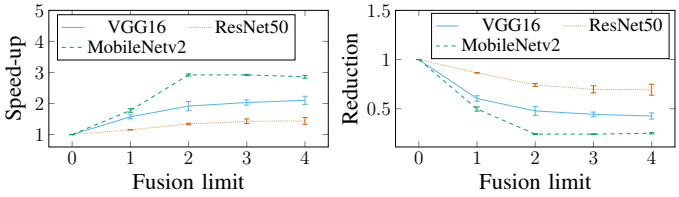

Figure 4: Latency (left) and data movement (right) improvements.

data movement costs between devices, fusion partitions and their parent share their $\mathcal{W}_{dev}$. The $fusion_{limit}$ configuration parameter limits the length of these *fused chains* by breaking the recursive generation. For space reasons the details of this recursive process are omitted.

Only after these POs are generated can the $\mathcal{W}_l$ of each layer be created, and the loss function is returned as described in (16). Once the loss function is generated, gradient descent is used to try to find the best combination of all $\mathcal{W}_l$ and $\mathcal{W}_{dev}$ weights that minimize the loss function. Two parameters are needed: a *learning rate* that controls the update factor of the weights, and an amount of *iterations* describing how many times gradient descent needs to be executed.

To obtain the actual distribution strategy, we choose for each layer the PO with the highest selection probability in $\mathcal{W}_l$. Then, for the selected partition, we assign each sublayer to the device with the highest selection probability in $\mathcal{W}_{dev}$. Last, the non-relaxed time and/or energy cost(s) can be calculated.

V. Experimental results

A. DIAPASON in isolation

Like multiple approaches before us [2], [15], we used a simulator to obtain the latency and energy costs of executing a particular layer on a specific device. We selected the Timeloop project [16] to generate a model of the systolic array-based accelerator Gemmini [17]. We used the linear model for the Raspberry Pi 3 proposed in [6] to model the CPU that controls the Gemmini accelerator from our simulation testbed.

We distribute the well-known VGG16, ResNet50 and MobileNetV2 models provided by the Keras framework. Unless stated otherwise, we used an input image size of $224 \times 224 \times 3$ in all experiments. We chose to use 8-bit quantized networks, as these are the ones normally deployed on embedded devices like the modelled accelerator.

Other parameters used, unless stated otherwise, were: two homogeneous Gemmini devices, a network bandwidth of 100 Mbps (megabits per second), a learning rate $lr = 0.4$, $fusion_{limit} = 2$, $weights_{init} = heuristics$, $part_{type} = naive$ and $iterations = 20$. Each run was repeated 10 times, and we report the mean and standard deviation of the final costs for each experiment.

1) Weights initialization impact: The convergence is improved by using the heuristics method to initialize the device selection weights, as can be seen in Fig. 3 by analysing the final latencies found for each model. The reproducibility is also improved, as seen in the standard deviation bars.

2) Fusion propagation limit impact: By sweeping $fusion_{limit}$ from 0 (no fusion) to 4 (maximum 4 consecutive layers can be fused together on the same device), Fig. 4 shows that the cases where fusion POs are generated ($fusion_{limit} > 0$) the distribution strategies obtained by DIAPASON are faster than the case where fusion is disabled. By adding fusion POs allows DIAPASON to find distribution strategies where less data is exchanged between devices.

3) Latency vs. energy metric optimization: We compare the distribution strategies obtained by DIAPASON when optimizing for different metrics. We used the network bandwidths for the communication channels as reported by [18]. By minimizing the loss function for a specific metric, DIAPASON is able to find better distribution strategies that optimize this metric (Fig. 5).

4) Heterogeneous devices: We configured DIAPASON to use two different Gemmini devices, one running at 100 MHz and one at 500 MHz. We evaluate both partition initialization procedures given by parameter $part_{type}$, as presented in Section IV. Enabling the computing based POs generation allows DIAPASON to find better distribution strategies, by improving load balancing across devices (Fig. 6).

B. DIAPASON compared against other works

In order to demonstrate the power and flexibility of DIAPASON, we compared our approach against the partitioning scheme BODP presented in the commonly used baseline MoDNN [5], and CoopAI [6], a work that implements comparable partition and fusion methods to the ones modelled by DIAPASON (Fig. 7). We configured DIAPASON to use the same device models used by CoopAI and the same network conditions. We executed DIAPASON on VGG16, using a learning rate $lr = 0.8$, a $fusion_{limit} = 4$, $weights_{init} = heuristics$ and $iterations = 20$.

MoDNN is not a good choice when the amount of devices starts to increase because it does not take into account the communication channel properties, and does not apply layer fusion. For RPi 3 devices, DIAPASON is always better than the MoDNN baseline, achieving a maximum speed-up of 64 % for the case with 7 devices, and a mean speed-up of 20 % across all experiments. For RPi 4 devices, DIAPASON achieves a maximum speed-up of 139 % for the case with 7 devices and a mean speed-up of 44 % across all experiments.

DIAPASON is also able to find better distribution strategies than CoopAI for each number of devices. For RPi 3 devices, DIAPASON achieves a maximum speed-up of 12 % for the case with 4 devices and a mean speed-up of 7 % across all experiments. For RPi 4 devices, DIAPASON achieves a

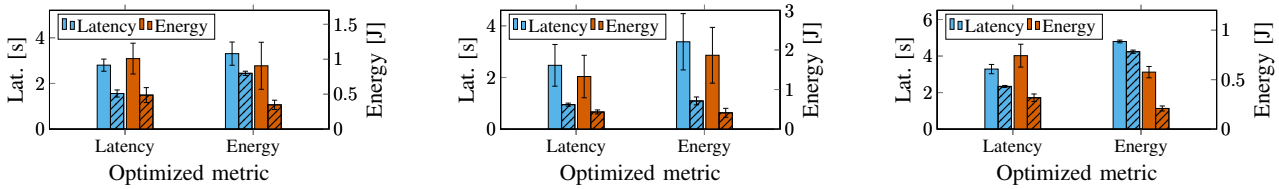

Figure 5: Latencies and energy costs for VGG16 (left), ResNet50 (center) and MobileNetV2 (right) (no pattern=Bluetooth, pattern=WiFi).

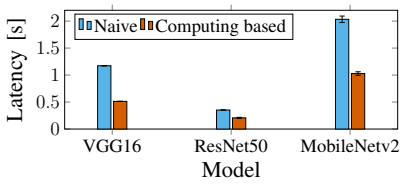

Figure 6: Latencies obtained by changing the $part_{type}$ parameter.

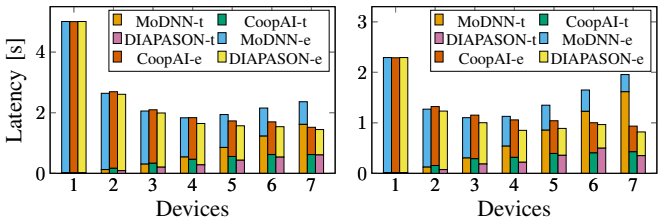

Figure 7: Transmission (t) and execution (e) costs using MoDNN, CoopAI and DIAPASON for VGG16 (left RPi 3, right RPi 4).

maximum speed-up of 24 % for the case with 4 devices and a mean speed-up of 12 % across all experiments.

VI. Conclusions

This work presented a novel methodology to solve the partition, fusion and allocation of an ANN to multiple devices. We have demonstrated DIAPASON's adaptability to various situations and the efficacy of our suggested approaches for enhancing its convergence, both for homogeneous and heterogeneous devices. When compared against MoDNN and CoopAI, DIAPASON was able to outperform them and find better distribution strategies, with a mean speed-up of 44% and 12% respectively.

In subsequent works, we would like to expand DIAPASON by adding hard requirements like energy, latency or per-device memory constraints. Although this work used a simulator and analytical models to evaluate the convergence of the algorithm and the solutions obtained by it, a future work should focus on deploying the ANNs on real distributed systems and study the gap between the predictions of DIAPASON and the actual measurements. Finally, another promising possibility is the potential to expand the available loss definitions to other metrics and combine them all into a single optimization target, in order to find a Pareto front of solutions.

References

[1] R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni, "Green AI," *CoRR*, vol. abs/1907.10597, 2019. [Online]. Available: http://arxiv.org/abs/1907.10597

[2] A. Banitalebi-Dehkordi, N. Vedula, J. Pei, F. Xia, L. Wang, and Y. Zhang, "Auto-Split: A General Framework of Collaborative Edge-Cloud AI," *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pp. 2543–2553, 2021.

[3] J. Zhou, Y. Wang, K. Ota, and M. Dong, "AAIoT: Accelerating Artificial Intelligence in IoT Systems," *IEEE Wireless Communications Letters*, vol. 8, no. 3, pp. 825–828, Jun. 2019.

[4] A. E. Eshratifar, M. S. Abrishami, and M. Pedram, "JointDNN: An Efficient Training and Inference Engine for Intelligent Mobile Cloud Computing Services," *IEEE Transactions on Mobile Computing*, vol. 20, no. 2, pp. 565–576, Feb. 2021.

[5] 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*, Mar. 2017, pp. 1396–1401.

[6] C.-Y. Yang, J.-J. Kuo, J.-P. Sheu, and K.-J. Zheng, "Cooperative Distributed Deep Neural Network Deployment with Edge Computing," *ICC 2021 - IEEE International Conference on Communications*, pp. 1–6, Jun. 2021.

[7] C. Hu and B. Li, "Distributed Inference with Deep Learning Models across Heterogeneous Edge Devices," *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, pp. 330–339, May 2022.

[8] Z. Fu, Y. Zhou, C. Wu, and Y. Zhang, "Joint Optimization of Data Transfer and Co-Execution for DNN in Edge Computing," *ICC 2021 - IEEE International Conference on Communications*, pp. 1–6, Jun. 2021.

[9] N. Li, A. Iosifidis, ros, and Q. Zhang, "Receptive Field-based Segmentation for Distributed CNN Inference Acceleration in Collaborative Edge Computing," *ICC 2022 - IEEE International Conference on Communications*, pp. 4281–4286, May 2022.

[10] H. Liu, K. Simonyan, and Y. Yang, "DARTS: Differentiable Architecture Search," Apr. 2019.

[11] X. Hou, Y. Guan, T. Han, and N. Zhang, "DistrEdge: Speeding up Convolutional Neural Network Inference on Distributed Edge Devices," *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1097–1107, May 2022.

[12] R. Stahl, A. Hoffman, D. Mueller-Gritschneder, A. Gerstlauer, and U. Schlichtmann, "Deeperthings: Fully distributed cnn inference on resource-constrained edge devices," *Int. J. Parallel Program.*, vol. 49, no. 4, p. 600–624, aug 2021. [Online]. Available: https://doi.org/10.1007/s10766-021-00712-3

[13] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," *CoRR*, vol. abs/1812.03443, 2018. [Online]. Available: http://arxiv.org/abs/1812.03443

[14] K. Lee, L. V. Linh, H. Kim, and C.-H. Youn, "Neural Architecture Search for Computation Offloading of DNNs from Mobile Devices to the Edge Server," *2021 International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 134–139, Oct. 2021.

[15] F. Kreß, J. Hoefer, T. Hotfilter, I. Walter, V. Sidorenko, T. Harbaum, and J. Becker, "Hardware-aware Partitioning of Convolutional Neural Network Inference for Embedded AI Applications," *2022 18th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pp. 133–140, May 2022.

[16] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 304–315.

[17] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 769–774.

[18] Y. Jin, J. Xu, Y. Huan, Y. Yan, L. Zheng, and Z. Zou, "Energy-Aware Workload Allocation for Distributed Deep Neural Networks in Edge-Cloud Continuum," *2019 32nd IEEE International System-on-Chip Conference (SOCC)*, pp. 213–217, Sep. 2019.