LO-SC: Local-only Split Computing for Accurate Deep Learning on Edge Devices

Luigi Capogrosso*, Enrico Fraccaroli*†, Marco Cristani*, Franco Fummi*, Samarjit Chakraborty†
*University of Verona, Verona, Italy †University of North Carolina at Chapel Hill, USA
Email:*{name.surname}@univr.it, †{enrifrac, samarjit}@cs.unc.edu

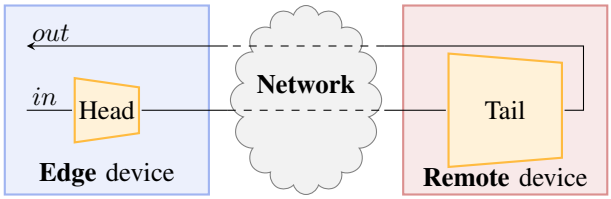
***Abstract*—Split Computing (SC) enables deploying a Deep Neural Network (DNN) on edge devices with limited resources by splitting the workload between the edge device and a remote server. However, relying on a server can be expensive, requires a reliable network, and introduces unpredictable latency. Existing solutions for on-device DNNs deployment often sacrifice accuracy for efficiency. In this paper, we study how to use the concepts from SC to split a DNN for executing on the same device without compromising accuracy. In other words, we propose Local-Only Split Computing (LO-SC), a new approach to split a DNN for execution entirely on the edge device while maintaining high accuracy and predictable latency. We formalize LO-SC as a Mixed-Integer Linear Problem (MILP) problem and solve it using a multi-constrained ordered knapsack algorithm. The proposed method achieves promising results on both synthetic and real-world data, offering a viable alternative for accurately deploying DNNs on resource-constrained edge devices. The source code is available at https://github.com/intelligolabs/LO-SC.**

***Index Terms*—Split Computing, Knapsack Problem, Deep Neural Networks, Edge Devices.**

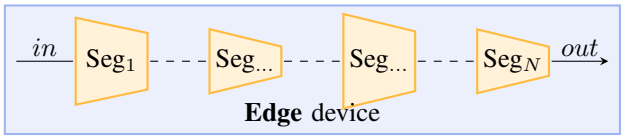
I. Introduction

In the last decade, Deep Neural Networks (DNNs) have made remarkable progress in solving a variety of complex problems. The widespread implementation of DNN-based applications on edge devices has brought about what is commonly referred to as Local-only Computing (LoC) approaches. Unfortunately, the computational demands of DNN models often surpass the capabilities of resource-constrained edge devices available today [1]. Solutions to this problem involve using simplified models, which inevitably result in lower overall accuracy. Consequently, the most common deployment approach of DNN-based applications is the Remote-only Computing (RoC) approach. In this paradigm, the DNN runs on a server, and the input is directly transferred from the acquisition device to the server through a network connection. The server computes the inferences and transmits the output to the device.

As a compromise between the LoC and the RoC approaches, *Split Computing (SC)* frameworks [2] propose to split DNN models into a head and a tail, deployed on an edge device and a remote server, respectively. Figure 1(a) shows how SC works. Early implementations of SC, like the work in [3], select a layer and divide the model to define the head and tail sub-models. Instead, more recent SC frameworks introduce partitions intending to minimize the overall latency time between the edge and the cloud [4].



(a) Existing SC architecture, where the DNN is divided between a head (deployed on an edge device) and a tail (on a remote server), based on the edge device capabilities, network reliability, and delay.



(b) Proposed LO-SC, where the DNN is divided into multiple segments, all deployed only on the edge device.

Figure 1. Traditional SC approach versus the proposed LO-SC architecture.

Motivations for this paper: Current state-of-the-art approaches in different Machine Learning (ML) applications often face severe latency constraints and require predictable latency for safety-critical applications [5]. They cannot afford potential disruptions from network limitations, such as traffic congestion, preventing their seamless execution on cloud-based platforms. Moreover, maintaining the maximum model accuracy is crucial for ensuring reliable and precise outcomes.

An example may be found in the industrial domain, where there is often a precise and enforced segmentation between corporate Information Technology (IT) networks and safety-critical Operational Technology (OT) networks. Certain pieces of equipment are segregated in the OT network. Here, the computation should occur near the machine, on the edge system, maintaining localized and secure data and avoiding risks associated with edge-to-cloud transfers.

Innovations in this paper: We explore the application of SC to systematically split a DNN to execute on a single edge device, introducing a new architecture called *Local-Only Split Computing (LO-SC)*. Among the optimization techniques used to address this problem, we formalize and solve a Mixed-Integer Linear Problem (MILP). Specifically, we adapted the knapsack optimization problem [6] (see Section III for more

This study was conducted within the MICS (Made in Italy – Circular and Sustainable) Extended Partnership and received funding from Next-Generation EU (Italian PNRR – M4 C2, Invest 1.3 – D.D. 1551.11-10-2022, PE00000004). CUP MICS D43C22003120001 - Cascade funding project CollaborICE, and by the European Union's Horizon Europe research and innovation programme under the Marie Sklodowska-Curie grant agreement No. 101109243. This work was also partially supported by the US NSF grant 2038960.

details), resulting in a *multi-constrained ordered knapsack* problem. This adaptation aims for a fast procedure to design a generic DNN architecture that, for the first time, can operate on edge devices without relying on specific optimization techniques, marking a significant advancement in the field. Thus, our models maintain the maximum accuracy possible with a predictable latency time.

Figure 1 summarizes the differences between SC architectures in the literature and our LO-SC. As shown in Figure 1(a), the typical SC scenario focuses on splitting a network in two, between an edge device and a remote server; the model's overall accuracy can be lower, and latencies are typically unpredictable. While Figure 1(b) depicts the LO-SC architecture proposed in this paper. The power of this formalization is the ability to define a set of segments with heterogeneous computation resources that will be executed on the same edge device, *e.g.*, different memory sizes, or maximum execution times. This approach ensures we do not compromise model performance while attaining a predictable latency.

In summary, the main contributions of this paper are:

- A new architecture that, for the first time, to the best of our knowledge, enables systematically running large DNNs entirely on edge devices without relying on additional optimization techniques.
- We formulate the proposed LO-SC partitioning as a MILP problem and solve it using a multi-constrained ordered knapsack algorithm.
- As a result, LO-SC preserves the highest accuracy levels of DNNs, and guarantees *predictable* latency times, which could potentially be higher than solutions using traditional SC that rely on cloud resources.

II. Related Work

We focus on architectures operating through a DNN model $\mathcal{M}(\cdot)$, whose task is to produce the inference output y from an input x. We can identify three major types of architectures used for distributed deep learning applications in the literature: LoC, RoC, and SC.

Local-only Computing (LoC): Under this policy, the entire computation is performed on the edge devices. Therefore, the edge device entirely executes the function $\mathcal{M}(x)$. Its advantage lies in offering low latency due to the proximity of the computing element to the sensor [1]. However, it may not be compatible with DNN-based architectures that demand robust hardware capabilities. Usually, simpler DNN models $\bar{\mathcal{M}}(x)$ that use specific techniques (*e.g.*, depth-wise separable convolutions) are used to build lightweight networks, such as MobileNetV3 [7].

Remote-only Computing (RoC): The input x is transferred through the communication network and then is processed at the remote system through the function $\mathcal{M}(x)$. This architecture preserves full accuracy considering the higher power budget of the remote system, but it leads to high latency and bandwidth consumption due to the input transfer.

Split Computing (SC): The general structure of SC is shown in Figure 1(a), which shows how the SC paradigm divides the DNN model into a head, executed by the edge sensing device, and a tail, executed by the remote system. It combines the advantages of both LoC and RoC thanks to the lower latency and, more importantly, drastically reduces the required transmission bandwidth by compressing the input to be sent x through the use of an autoencoder [4]. Formally, the encoder and decoder models are defined as $z_l = \mathcal{F}(x)$ and $\bar{x} = \mathcal{G}(z_l)$, which are executed at the edge, and remotely, respectively. The distance $d(x, \bar{x})$ defines the performance of the encoding-decoding process.

One of the earliest works on SC is the study by Kang *et al.* [3], which shows that the initial layers of a DNN are the most suitable candidates for partitioning, as they optimize both latency and energy consumption. Additionally, latency reduction is usually achieved through quantization, as explored in [8], and the utilization of lossy compression techniques prior to data transmission, as investigated in [9]. The work in [10] explores lossless techniques to encode intermediate results without modifying the ML model. The concept of employing autoencoders to compress the data further is discussed in various studies, such as [11]. Moreover, in [12], the effect of predefined sparsity within the SC paradigm is presented. This approach reduces computational, storage, and energy demands, regardless of the hardware platform.

The prevalent methods for identifying potential splitting points have evolved from architecture-based to more refined neuron-based techniques. Specifically, architecture-based methods, such as those described in [13], identified candidate split points at layers where the network's size decreased. The underlying assumption is that compressing information at these points is more efficient due to the reduced computational complexity. Instead, regarding neuron-based techniques, [14] and [15] show that not only the architecture of the layers but also the saliency of individual layers is a crucial factor when deciding where to split, where the neuron's saliency is determined by its gradient in relation to the accurate decision.

However, current state-of-the-art approaches in different ML applications rely on advanced learning procedures, such as Multi-Task Learning (MTL), *i.e.*, a paradigm in which multiple related tasks are jointly learned to improve the generalizability of a model by using shared knowledge across different aspects of the input. In [16], the first method to partition multi-tasking DNNs in a SC scenario is presented. The proposed design handles multiple tasks concurrently, shifting the focus from Single-Task Learning (STL) in SC, and through MTL, increases task performances, overcoming the challenge of preserving only the performance of the main task.

From a different context, our work is also related to scheduling and partitioning of streaming tasks [17] to satisfy memory/buffer constraints [18], [19]. Techniques from this domain may be borrowed to further optimize LO-SC. While existing SC approaches focus on splitting a network in two between an edge device and a remote server, our proposal in contrast *i*) defines a set of segments to be executed on the same embedded device, *ii*) maintains model accuracy without any losses, and *iii*) provides predictable latencies.

III. Methodology

Our goal is to optimally split the neural network based on the computation capabilities of the edge device while minimizing a desired metric.

The splitting is akin to the well-known *knapsack problem*, where we want to pack a set of items, with given sizes, into containers with a specified capacity. In our case, the items are the *layers* of the neural network, the size is the *memory* each layer occupies, and the capacity of the containers is the *available memory* of our edge device. We refer to containers because if the neural network cannot be fully executed due to constrained resources, we must split it into segments. We can execute each segment one after the other sequentially to obtain the desired result from the DNN. For the remainder of the paper, we call *segment* a group of sequential layers executed together.

A. Optimization Problem Formulation

There are several metrics we could be interested in minimizing. Still, the objective we discuss in this paper is minimizing the output size of the last layer executed in each segment. This objective has a positive impact, regardless of the scenario. On the one hand, one could decide to maintain the output of the last layer of each segment in memory. However, occupying too much memory on a device with constrained memory is not advisable. On the other hand, one could decide to store the output of each segment somewhere else (from GPU to RAM or from RAM to disk). However, moving large quantities of data directly impacts the overall execution time of the DNN. Nonetheless, the proposed formalization allows the exploration of other metrics in the splitting problem.

Setting and notation: Given L layers in the DNN, we know the following information for each layer $l \in [0 \dots L]$:

- $m(l)$: memory occupied by layer l;
- $t(l)$: worst-case execution time of layer l;
- $o(l)$: output memory size of layer l.

Given a total of S segments, we know the following information for each segment $s \in [0 \dots S]$:

- $M(s)$: the available memory of segment s;
- $T(s)$: the available execution time of segment s.

Decision variables: Let us start by defining the decision variables of our problem. The first variable we define is x, which keeps track of where the layers are placed as follows:

$$x[l,s] = \begin{cases} 1, & \text{if layer } l \text{ is placed in segment } s \\ 0, & \text{otherwise} \end{cases}$$
$$\forall l \in [0 \dots L], \forall s \in [0 \dots S] \, .$$

Once the solver has found a solution, reading the assignment it gave to these variables gives you the complete placement of layers in each segment. The second variable we introduce is u, which keeps track of whether a segment is in use as follows:

$$u[s] = \begin{cases} 1, & \text{if segment } s \text{ contains at least one layer} \\ 0, & \text{otherwise} \end{cases}$$
$$\forall s \in [0 \dots S] \, .$$

Then, we define i, which contains the index of the last layer placed inside a segment as follows:

$$0 \leq i[s] \leq L, \qquad \forall s \in [0 \dots S] \, .$$

Finally, y keeps track of the output size of the last layer placed inside a segment as follows:

$$0 \leq y[s], \qquad \forall s \in [0 \dots S] \, .$$

Problem constraints: Following is the set of constraints that will ensure that the solutions abide by our resource constraints. The first constraint ensures that we place each layer in only one segment:

$$\sum_{s=0}^{S} x[l,s] = 1 \quad \forall l \in [0 \dots L] \, . \tag{1}$$

The next constraint ensures that the total memory occupied by the layers placed inside a segment is lower than its capacity:

$$\sum_{l=0}^{L} x[l,s] \cdot m(l) \leq M(s), \quad \forall s \in [0 \dots S] \, . \tag{2}$$

A similar constraint ensures that the total execution time for all the layers placed inside a segment is lower than its total available time:

$$\sum_{l=0}^{L} x[l,s] \cdot t(l) \leq T(s), \quad \forall s \in [0 \dots S] \, . \tag{3}$$

While the focus of the paper centers around memory constraint edge devices, placing a limit on the total execution time for each segment serves its purpose. This constraint ensures that the scheduler, responsible for executing layers based on the outcome of the MILP, maintains a consistent overall runtime. Producing a consistent execution time and delay due to segment swapping improves the practical utility of this approach for real-time applications.

With the following constraint, u becomes $true$ only if there are layers placed inside the given segment:

$$u[s] = \max_{l \in [0 \dots L]} x[l,s], \quad \forall s \in [0 \dots S] \, . \tag{4}$$

Since x is a Boolean, the maximum of a set of x results in a Boolean that reflects the presence of even just one layer in a segment. The next constraint ensures that if we do not use a segment, we are not going to use the subsequent ones as well:

$$\neg u[s] \implies \neg u[s+1], \quad \forall s \in [0 \dots S-1] \, . \tag{5}$$

This ensures that the segments are incrementally used, from the first to the last, without empty ones in the middle. The next constraint ensures that we execute the layers in an ordered fashion:

$$x[l_1, s_1] \implies \neg x[l_2, s_2],$$
$$\forall s_1 \in [0 \dots S], \forall s_2 \in [s_1 + 1 \dots S], \tag{6}$$
$$\forall l_1 \in [0 \dots L], \forall l_2 \in [0 \dots l_1] \, .$$

To better understand this constraint, let us take the example with four layers and four segments as shown in Figure 2.

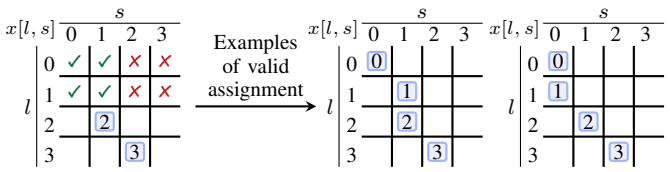


Figure 2. Two examples of valid assignments of layers to segments. We can place layers 0 and 1 in the same segment of layer 2 or the previous ones but not in the following ones.

Layers 2 and 3 are placed in segments 1 and 2, respectively. As such, layers 0 and 1 can only be placed in the same segment where layer 2 is or in those before it. Figure 2 also shows two examples of valid assignments of layers to segments on the right-hand side.

The following constraints are specific for keeping track of the last layer executed inside each segment. We start by storing the index of the last layer placed inside a segment as follows:

$$i[s] = \max_{l \in [0 \dots L]} \quad x[l,s] \cdot l, \quad \forall s \in [0 \dots S] \; . \tag{7}$$

Now, let us take again the example of Figure 2, where the values inside the blue boxes result from multiplying the decision variable $x[l,s]$ by the index l. The variable $i[s]$ is the maximum value inside column s. Finally, we use the previously computed index to get the actual output size of that layer, if and only if the execution segment is actually in use, as follows:

$$y[s] = o(i[s]) \cdot u[s], \quad \forall s \in [0 \dots S] \; . \tag{8}$$

Optimization function: We aim to minimize the sum of the output size of the last layers placed inside each segment. The function we use is the following:

$$\text{minimize} \quad \sum_{s \in [0 \dots S]} y[s] \; . \tag{9}$$

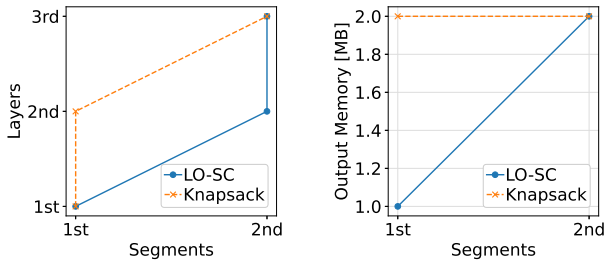
This minimization objective is crucial for ensuring efficient computational resource utilization or reducing potential overhead due to transferring the output at the end of each segment.

The optimization output is an assignment to the decision variable $x[l,s]$, where truth values identify when layers should be executed, akin to a schedule. The schedule is then read by a scheduler that takes care of *i*) loading the group of layers specified by a segment, *ii*) running the inference, *iii*) unloading the batch of layers, and *iv*) passing the partial output to the next batch of layers.

B. Computational Complexity

To get a rough idea of the size of the instances this approach can address, we should consider the asymptotic growth of the variables allocated for the MILP formulation. Since we do not know the number of segments we require apriori, we consider creating a number of segments S that can accommodate double the amount of memory required by the layers.

The worst-case upper bound on the size of variable $x[l,s]$ is $O(L \cdot S)$, while that of variables $u[s]$, $i[s]$, and $y[s]$, is $O(S)$.



(a) Allocation of layers in segments. (b) Output size for each segment.

Figure 3. Comparing our implementation of the multi-constrained ordered knapsack for LO-SC and knapsack.

The worst-case upper bound on constraints can be similarly computed. For Equations (1), (2) and (3) it is $O(S \cdot L)$, while for Equations (4), (5), (7) and (8) it is $O(S)$. Finally, for Equation (6), we can consider a worst-case upper bound of $O(S^2 \cdot L^2)$. Considering all these upper bounds, the setup phase takes approximately $O(S^2 \cdot L^2)$ in the worst case.

IV. Experiments

Models details: In our experiments, we use two well-known DNNs, *i.e.*, VGG16 [20], MobileNetV1 [21]. We chose the VGG16 because it is a well-established and widely used architecture in many image-processing tasks. MobileNets, instead, represent cutting-edge DNNs for embedded vision applications. Besides being very popular architectures, they are also interesting for LO-SC since they exhibit a large number of parameters and a good depth in terms of layers.

Implementation details: All the code regarding the DNN is implemented in PyTorch Lightning, and the pre-trained network used corresponds to the implementations contained in PyTorch [22]. The code for solving the MILP problem is implemented starting from the CP-SAT constraint programming solver of the OR-Tools open-source software suite [23].

Used edge devices: We run our experiments on an NVIDIA Jetson Nano with 4 GB of memory, except for the VGG16 with inputs of 1920px and 2048px, which we run on an NVIDIA Jetson Xavier NX with 8 GB of memory. For brevity, when we write 1280px (or other resolution), we intend an image of 1280 pixels in width and 1280 pixels in height.

A. Demonstrative Example

Let us start with a demonstrative example that highlights the differences between a vanilla knapsack and our multi-constrained ordered knapsack implementation, focusing only on the layers occupied memory and disregarding runtime. We take a simple DNN composed of just three layers, each of which occupies 3 MB (*i.e.*, a total size of 9 MB). However, the output size of the first layer is 1 MB, while for the others is 2 MB. Due to device limitations, we are limited to segments of maximum 6 MB of memory.

Figure 3 compares the solutions found by the proposed LO-SC approach and by a knapsack implementation that aims to minimize the number of occupied segments. Figure 3(a)

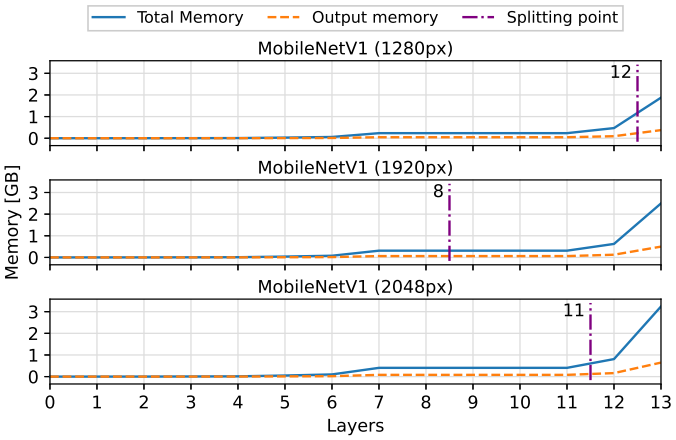


Figure 4. Splitting points determined by LO-SC when applied to the MobileNetV1 [21] model. The task is to process images of various resolutions, *i.e.*, 1280px, 1920px, and 2048px, on an NVIDIA Jetson Nano.

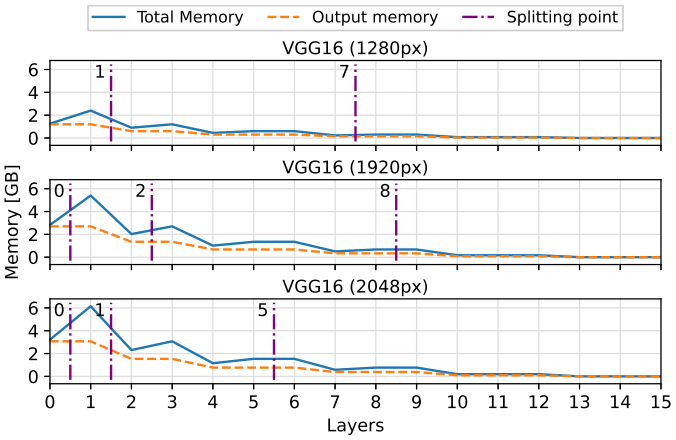


Figure 5. Splitting points determined by LO-SC when applied to the VGG16 [20] model. The task is to process images with a resolution of 1280px on an NVIDIA Jetson Nano and with 1920px and 2048px on an NVIDIA Jetson Xavier NX.

shows the allocation between layers (y-axis) and segments (x-axis). The figure shows that LO-SC performs a split after the first layer while knapsack after the second layer. As a result, the output size of the last layer placed inside those segments changes as shown in Figure 3(b). The segmentation proposed by LO-SC has an output size at the end of the first segment of 1 MB, while with a knapsack, it is 2 MB. Inevitably, the output size of the last segment is 2 MB in both cases.

B. LO-SC Quantitative Results

In these experiments, we focus on the classification task using images of different resolutions, *i.e.*, 1280px, 1920px, and 2048px, depicting objects passing on a conveyor belt within a real Industry 4.0 scenario: the Industrial Engineering Laboratory (ICE Lab) at the University of Verona. In particular, Figure 4, and Figure 5 show the results of the splitting points determined by LO-SC when applied to the MobileNetV1, and VGG16, respectively.

Despite being a lightweight architecture, MobileNetV1 still exceeds the memory constraints of 4 GB of the Jetson Nano for all the inputs. Therefore, LO-SC splits it into two different segments, each of which respects the memory constraint of the board. These results demonstrate the capability of LO-SC to enable the execution of a DNN (in which the memory demands exceed the device's available resources) on an edge device without relying on optimization techniques or the cloud.

Table I
ACCURACY, PRECISION, RECALL, AND F1 SCORE COMPARING THE VGG16: VANILLA, ON WHICH QUANTIZATION AND PRUNING WERE APPLIED, AND ON WHICH LO-SC WAS APPLIED.

VGG16	Accuracy ↑	Precision ↑	Recall ↑	F1 ↑
Vanilla	85.55 %	0.86	0.86	0.86
Quantization and Pruning	77.23 %	0.79	0.77	0.77
LO-SC (ours)	85.55 %	0.86	0.86	0.86

On the other hand, the VGG16 should be split into three segments in the case of 1280px and four in the case of 1920px and 2048px. In particular, the experiments with the resolutions of 1920px and 2048px must be conducted on the Jetson Xavier NX, as some segments, such as the first, reach a total memory of 5.52 GB for 1920px, and 6.28 GB for 2048px, both exceeding the 4 GB of the Jetson Nano. This reveals another outcome of LO-SC: it delineates the specific characteristics of your DNN and aids in selecting the most appropriate hardware device.

It is worth noting how the design output of LO-SC varies not only based on the processed DNN but also on the input being handled. This demonstrates the dynamic nature of splitting strategies when accommodating different input dimensions within the network.

C. Comparing LO-SC Against LoC

LO-SC can outperforms traditional LoC for deploying DNNs on resource-constrained edge devices. We demonstrate this by comparing a fine-tuned VGG16 on CIFAR-10 using quantization and pruning vs. LO-SC. CIFAR-10 has to be considered as a placeholder for bigger datasets (*e.g.*, ImageNet [24]); indeed, the focus is to show the comparison between most used LoC methods and LO-SC, and not beating the state-of-the-art in image classification.

In particular, we combine a uniform scalar quantization, where floating-point values are linearly compressed and rounded to low-precision quantized types, and magnitude-based pruning, following [25].

The results in Table I show that VGG16 can be deployed on Jetson using the LO-SC technique without losing accuracy. Since no memory optimizations are applied, the output remains uncompressed, maintaining the highest accuracy. However, applying quantization and pruning significantly reduces performance.

D. Comparing LO-SC Against RoC

With LO-SC, the latency for the switch between segments is entirely predictable, provided by prior knowledge about the number of segments and the transfer rate of the target hardware.

For instance, consider the switch operation between segments 1 and 2 of VGG16 with a 1920px input. The amount of data to be transferred is $5.5\,\mathrm{GB}$. Assuming a gigabit channel, the time required to transfer the data is $\approx 44\,\mathrm{s}$ (this is only a conservative estimate since we don't consider potential network congestion and delays). Instead, the switch between segments on the considered architectures takes $\approx 15.73\,\mathrm{s}$.

The ability to provide an inference with a deterministic delay could make LO-SC suitable for real-time scenarios.

E. Comparing LO-SC Against SC

This section delves into a comparative analysis between LO-SC and SC. Specifically, we compare the splitting points identified by LO-SC and those identified by the two state-of-the-art methods of SC, namely CDE [13] and I-SPLIT [14] (see Section II). We specify that [15] adopts the basic methodology of [14] with some engineering-specific modifications. Therefore, the evaluation conducted against [14] inherently includes a comparison with [15] as well. Both papers use a VGG16 for experimental purposes, so our comparison is based on this model. In particular, CDE suggests potential splitting points at layers 5, 9, and 13, whereas I-SPLIT recommends splitting the model at layers 5, 9, 11, 13, and 15.

Several facts do emerge. Firstly, none of the potential configurations suggested by the two methods would be feasible in a LO-SC scenario, as the two segments do not meet the constraints of both the Jetson Nano and the Jetson NX. Moreover, in contrast to LO-SC, none of them allows partitioning the DNN into more than two segments.

Secondly, unlike LO-SC, the final accuracy after the split operation might not necessarily be the highest. In the case of I-SPLIT, this occurs because the authors insert a bottleneck that employs lossy data compression to reduce the transmission data on the network channel. On the other hand, CDE utilizes advanced learning techniques, like the teacher-student approaches [26], to craft an architecture that optimizes edge performance post-split. However, these don't guarantee successful outcomes.

Finally, the overall latency time of the system cannot be reliably estimated due to potential network congestion and its unpredictable impact. Conversely, in LO-SC, the latency can be easily estimated.

V. Concluding Remarks

This paper presented LO-SC, to enable generic DNNs to be implemented on resource-constrained embedded devices without relying on specific optimization techniques and eliminating reliance on cloud-based computation while preserving inference accuracy. We formalized LO-SC as a MILP problem and solved it using a multi-constrained ordered knapsack algorithm. Results on real-world data, along with a comparison against the best state-of-the-art methods in SC, confirm the utility of LO-SC. As future work, we plan to deploy LO-SC in multiple learning-enabled cyber-physical systems domains [27] and fine-tune its latency and memory requirements [28].

References

[1] L. Capogrosso *et al.*, "A Machine Learning-Oriented Survey on Tiny Machine Learning," *IEEE Access*, pp. 23 406–23 426, 2024.

[2] Y. Matsubara *et al.*, "Split Computing and Early Exiting for Deep Learning Applications: Survey and Research Challenges," *ACM Computing Surveys*, vol. 55, no. 5, pp. 1–30, 2022.

[3] Y. Kang *et al.*, "Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 615–629, 2017.

[4] Y. Matsubara *et al.*, "Distilled Split Deep Neural Networks for Edge-Assisted Real-Time Systems," in *Workshop on Hot Topics in Video Analytics and Intelligent Edges*, 2019.

[5] D. Roy *et al.*, "Multi-Objective Co-Optimization of FlexRay-Based Distributed Control Systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[6] H. M. Salkin *et al.*, "The knapsack problem: A survey," *Naval Research Logistics Quarterly*, vol. 22, no. 1, pp. 127–144, 1975.

[7] A. Howard *et al.*, "Searching for MobileNetV3," in *Intl. Conf. on Computer Vision (ICCV)*, 2019.

[8] G. Li *et al.*, "Auto-tuning Neural Network Quantization Framework for Collaborative Inference Between the Cloud and Edge," in *Artificial Neural Networks and Machine Learning (ICANN)*, 2018.

[9] H. Choi *et al.*, "Deep Feature Compression for Collaborative Object Detection," in *25th Intl. Conf. on Image Processing (ICIP)*, 2018.

[10] D. Carra *et al.*, "DNN Split Computing: Quantization and Run-Length Coding are Enough," in *Global Communications Conf. (GLOBECOM)*, 2023.

[11] Y. Matsubara *et al.*, "BottleFit: Learning Compressed Representations in Deep Neural Networks for Effective and Efficient Split Computing," in *23rd Intl. Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2022.

[12] L. Capogrosso *et al.*, "Enhancing Split Computing and Early Exit Applications through Predefined Sparsity," in *Forum on Specification & Design Languages (FDL)*, 2024.

[13] M. Sbai *et al.*, "Cut, Distil and Encode (CDE): Split Cloud-Edge Deep Inference," in *18th Intl. Conf. on Sensing, Communication, and Networking (SECON)*, 2021.

[14] F. Cunico *et al.*, "I-SPLIT: Deep Network Interpretability for Split Computing," in *26th Intl. Conf. on Pattern Recognition (ICPR)*, 2022.

[15] L. Capogrosso *et al.*, "Split-Et-Impera: A Framework for the Design of Distributed Deep Learning Applications," in *26th Intl. Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, 2023.

[16] L. Capogrosso *et al.*, "MTL-Split: Multi-Task Learning for Edge Devices using Split Computing," in *61st DAC*, 2024.

[17] S. Chakraborty *et al.*, "A New Task Model for Streaming Applications and its Schedulability Analysis," in *DATE*, 2005.

[18] A. Maxiaguine *et al.*, "Rate Analysis for Streaming Applications with On-chip Buffer Constraints," in *ASP-DAC*, 2004.

[19] M. Alexander *et al.*, "DVS for Buffer-Constrained Architectures with Predictable QoS-Energy Tradeoffs," in *CODES+ISSS*, 2005.

[20] K. Simonyan *et al.*, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[21] A. G. Howard *et al.*, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv preprint arXiv:1704.04861*, 2017.

[22] A. Paszke *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[23] L. Perron *et al.*, "CP-SAT," Google. [Online]. Available: https://developers.google.com/optimization/cp/cp_solver/

[24] J. Deng *et al.*, "ImageNet: A large-scale hierarchical image database," in *Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[25] S. Han *et al.*, "Learning both Weights and Connections for Efficient Neural Networks," *Advances in Neural Information Processing Systems (NeurIPS)*, 2015.

[26] J. Gou *et al.*, "Knowledge Distillation: A Survey," *Intl. Journal of Computer Vision*, vol. 129, no. 6, pp. 1789–1819, 2021.

[27] G. Tibba *et al.*, "Testing Automotive Embedded Systems under X-in-the-Loop Setups," in *ICCAD*, 2016.

[28] W. Chang *et al.*, "Memory-Aware Embedded Control Systems Design," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 36, no. 4, pp. 586–599, 2017.