Received 22 February 2023, accepted 31 March 2023, date of publication 5 April 2023, date of current version 12 April 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3264828

RESEARCH ARTICLE

A Pipelining-Based Heterogeneous Scheduling and Energy-Throughput Optimization Scheme for CNNs Leveraging Apache TVM

DELIA VELASCO-MONTERO[1], BART GOOSSENS[2], (Member, IEEE), JORGE FERNÁNDEZ-BERNI[1], ÁNGEL RODRÍGUEZ-VÁZQUEZ[1], (Life Fellow, IEEE), AND WILFRIED PHILIPS[2], (Senior Member, IEEE)

[1]Instituto de Microelectrónica de Sevilla, Universidad de Sevilla-CSIC, 41092 Seville, Spain
[2]Image Processing and Interpretation Research Group, imec-Ghent University, 9000 Ghent, Belgium

Corresponding author: Delia Velasco-Montero (delia@imse-cnm.csic.es)

This work was supported in part by the Flemish Government (Belgium) through AI Research Program by Project SEMIoTICS under Grant PID2021-128009OB-C31; in part by MCIN/AEI/10.13039/501100011033 and EU "ERDF: A Way of Making Europe," through Project ULTIMATE under Grant TED2021-131835B-I00; in part by MCIN/AEI/10.13039/501100011033 through the European Union NextGenerationEU/ PRTR, by the U.S. Office of Naval Research under Grant N00014-19-1-2156; and in part by the European Commission and Junta de Andalucía through Erasmus+ Traineeship Program.

ABSTRACT Extracting information of interest from continuous video streams is a strongly demanded computer vision task. For the realization of this task at the edge using the current de-facto standard approach, i.e., deep learning, it is critical to optimize key performance metrics such as throughput and energy consumption according to prescribed application requirements. This allows achieving timely decision-making while extending the battery lifetime as much as possible. In this context, we propose a method to boost neural-network performance based on a co-execution strategy that exploits hardware heterogeneity on edge platforms. The enabling tool is Apache TVM, a highly efficient machine-learning compiler compatible with a diversity of hardware back-ends. The proposed approach solves the problem of network partitioning and distributes the workloads to make concurrent use of all the processors available on the board following a pipeline scheme. We conducted experiments on various popular CNNs compiled with TVM on the Jetson TX2 platform. The experimental results based on measurements show a significant improvement in throughput with respect to a single-processor execution, ranging from 14% to 150% over all tested networks. Power-efficient configurations were also identified, accomplishing energy reductions above 10%.

INDEX TERMS Apache TVM, continuous inference, convolutional neural networks, edge vision, heterogeneous processing, jetson TX2, performance optimization.

I. INTRODUCTION

Edge computing, which brings computation close to the data acquisition system, i.e., sensors, constitutes the reference paradigm to replace cloud-based solutions for a wide set of computer vision applications. It permits to save bandwidth, remove communication latency, and preserve information privacy. However, the execution of computationally demanding convolutional neural networks (CNNs) on resource-constrained embedded devices calls for efficient implementations. This remains a challenge, especially in the scope of edge applications requiring real-time decision-making and/or low-energy inference.

The trade-offs related to the realization of vision at the edge have been addressed from diverse perspectives. Specific and progressively more efficient devices have been developed in the last few years, such as low-power edge GPUs [1], [2], [3], neural-network accelerators [4], embedded CPUs [5], or tensor processor units [6]. Strategies based on multiple devices have also been proposed, including *hybrid cloud-edge*

The associate editor coordinating the review of this manuscript and approving it for publication was Giovanni Merlino.

solutions [7], [8], [9], [10], [11], [12], [13] and *collaborative systems* in the framework of the Internet of Things [14], [15], [16], [17], [18]. These methods are highly dependent on the availability of effective communication networks and require additional algorithms, which increase resource utilization and implementation cost. An intermediate solution consists in *exploiting hardware heterogeneity* (iii). This means properly leveraging the computational resources available on edge platforms for the sake of boosting the inference performance. The present work applies this intermediate solution, which exploits heterogeneous processors included in an edge platform through a co-execution collaborative scheme. The advantages of this approach are as follows:

- There is no need for additional infrastructure, which is particularly relevant given that many edge systems and mobile devices already integrate a variety of processing components: CPU, GPU, TPU, NPU, DSP, etc.
- Compared to cloud-based and multiple-node collaborative solutions, local mobile computing (1) facilitates data privacy and (2) mitigates the dependency on a communication network with enough bandwidth.
- The cost of inter-device data transmission of cloud-dependent solutions is saved. This is important given that data transfer latencies in the cloud are often high – especially on 3G and LTE [8], [19].
- As long as the network model does not change, the model partition algorithm only needs to be performed once; thus, its cost can be amortized over continuous executions.

This manuscript is organized as follows. Next sub-sections of present Sect. I establish the context and scope of our study within the state-of-the-art and emphasizes the relevance of the reported results. Section II describes the proposed approach and mathematically formulates the optimization problem. Experimental set-up and practical implementation details are provided in Sect. III. Corresponding performance results are reported in Sect. IV. Finally, we discuss the strengths of the proposed method in Sect. V, highlighting how it can be extended to allow dynamic runtime scheduling.

A. RELATED WORK

The key question we aimed to answer with this study is whether the hardware resources accessible in heterogeneous edge platforms can be effectively exploited to attain significant performance enhancement. Certainly, previous works on heterogeneous edge computing evidenced the advantages of this approach. For instance, the effectiveness of CNN workload distribution on the CPU-GPU architecture of Google Pixel 2 was evaluated in [20]. In this case, the proposed strategy is based on hardware allocation according to the system runtime state, in contrast to network partition schemes. Neural-network acceleration on the ARM big.LITTLE architecture through heterogeneous processing has also been reported [21], [22], [23]. *Pipe-all* [21] carries out network partitioning to implement a CPU-GPU pipeline schedule with ARM Compute Library (ARM-CL) running on both ARM CPU and Mali GPU. *Pipe-it* [22] leverages big.LITTLE technology – *big* and *small* quad-core CPU clusters – by applying layer-level network distribution in a co-execution scheme also based on ARM-CL. Similarly, the authors in [23] built a pipeline processing scheme on a multi-core platform by splitting the network into processing stages assigned to groups of cores. An heterogeneous two-stage processing pipeline on Jetson AGX Xavier has also been proposed [24]. In this work, the accelerator included in the board performs most of the CNN operations and then offloads the last layers of the network to the CPU. When processing batches of images with that pipeline-based strategy, inference speedups are achieved. A more comprehensive framework is represented by *DeepX* [25], which constitutes a solution to execute neural networks on multiple processors through layer compression and workload distribution. Scheduling of machine-learning (ML) tasks over heterogeneous FPGA-GPU or multi-FPGA embedded systems have also been explored [26], [27], as well as layer-pipelined CNN acceleration schemes tailored for FPGAs [28]. In contrast to these approaches, our work aims to set the basis for generalized heterogeneous hardware exploitation, by employing 1) the TVM library, which is compatible with a great variety of hardware devices; and 2) allowing the scheduler to select any combination of processing devices and number of stages. We also focus on processing continuous image streams, both in a fast-response or energy-saving manner.

On the other hand, cloud-based and edge-only approaches constitute other heterogeneous techniques enabling edge applications. *Neurosurgeon* [8] employs Jetson TX1 and a cloud server to dynamically adapt network partitioning according to estimated runtimes or energy consumption. Low-cost low-energy devices, such as Raspberry Pi and Odroid, have also been employed in edge-cloud hybrid solutions with dynamic network conditions [15], [29]. Multi-node edge-edge solutions can also optimize network execution by applying layer fusion, data parallelism, or network partition over clusters of edge devices [18], [19], [30], [31], [32], [33]. For instance, [34] proposes spatial and channel partitioning to parallelize convolutional layers in multiple devices using dynamic programming-based search. However, in these methods, the need for data transfer between devices in a network increases communication latency and incurs high network traffic and privacy risks. Moreover, an extra economic cost is also frequently incurred owing to the complexity of distributed computing mechanisms.

Finally, an example of approach exploiting heterogeneous computing resources for real-time applications is LoPECS [35], which enables running autonomous driving services on a resource-constrained device.

B. MOTIVATION

A summary of related works reported in the literature is provided in Table 1. Although previous studies showed the potential of heterogeneous execution on specific platforms,

TABLE 1. **Related approaches to accelerate DNN inference at the edge. We propose a single-platform heterogeneous scheme that leverages a general open-source machine-learning compiler (Apache TVM). This scheme has been tested for various CNNs runnning on Jetson TX2. △throughput denotes the performance improvement in terms of that metric with respect to the performance of a single processor. It can be observed that the proposed methodology presents state-of-the-art improvements.**

(i) Hybrid edge-cloud approaches

	Hardware (Software)	**Testing workloads**	**Partitioning method**
[8]	Jetson TX1 (Caffe)	8 DNNs	Dynamic layer-level splitting
[15]	Raspberry Pi 3 (Caffe)	AlexNet, ResNet-18	Dynamic adaptive DNN surgery
[29]	Odroid XU4 (Caffe)	5 CNNs	Graph-based partitioning

(ii) Multi-node edge-edge approaches

	Hardware (Software)	**Testing workloads**	**Partitioning method**	**# devices**
[18]	Raspberry Pi 3B (DarkNet)	YOLOv2	Fused Tile partitioning	1–6
[33]	LG Nexus 5 (MxNet)	VGG-16	Biased one-dimensional partition	2–4
[19]	Raspberry Pi 3B+ (DarkNet)	VGG-16, YOLO	Fused-layer parallelization	1–8
[30]	MinnowBoard, RCC-VE Network Board (PyTorch)	ViT-Base, ViT-Large, ViT-Huge	Layer-level splitting	16
[31]	Raspberry Pi 3B + Jetson TX2 (Keras-TensorFlow)	AlexNet, VGG-16	Intra- & inter-layer partitioning	4–12
[34]	Simulation Virtual Machines (TensorFlow)	VGG-16	Fused-layer parallelization	5
[32]	Nexus 6 / Galaxy S2 + LG Watch Urbane (TensorFlow)	8 CNNs	Dynamic graph partitioning	2

(iii) Single-platform heterogenous co-execution approaches

	Hardware (Software)	**Testing workloads**	Partitioning method	△throughput
[20]	Google Pixel 2 CPU+GPU (TensorFlow Lite)	1–3 CNNs	Reinforcement learning (no partitioning)	1.159 – 1.233
[21]	ARM big.Little CPU + ARM Mali GPU (ARM-CL)	5 CNNs	3-stage layer-level splitting	1.759 on average
[22]	Hi3670 big.LITTLE SoC (ARM-CL)	5 CNNs	Multi-stage layer-level splitting	1.098 – 1.675
[23]	Rock960 big.LITTLE multi-core platform (ARM-CL)	3 CNNs	Iterative graph bi-partition	1.73 on average
[24]	Jetson AGX Xavier CPU+accelerator (Caffe)	5 CNNs	2-stage layer-level splitting	1.439 on average
This work	Jetson TX2 CPU+GPU (TVM)	5 CNNs	2-stage layer-level splitting	1.138 – 2.502

the main challenge is to design a general approach that can combine the specific strengths/weaknesses of the various CPUs and accelerators in a close to optimal way (e.g., power or performance wise). Note that the execution of processing workloads on different components – TPU, VPU, CPU, etc. – could be impossible due to software or hardware incompatibilities. A second challenge is to achieve efficient distribution of CNN layer operation into processing units. The implementation of dynamic scheduling and load balancing constitutes a third remaining challenge (note that some approaches require prohibitive search optimization times [21]). To address these issues, we propose a simple but efficient method based on layer-level network mapping into processors using TVM, a powerful general software compiler.

Apache TVM [36], [37] is an open-source framework that allows compiling ML workflows on a vast set of hardware back-ends. It is compatible with other ML frameworks, such as TensorFlow, Keras, MXNet, PyTorch, and DarkNet. In addition, TVM includes auto-tuner tools – i.e., autoTVM [37] and Ansor [38] – to automatically apply graph-level and operator-level optimizations – e.g., operation fusion or data transformations – to network graphs, generating highly efficient machine code. Another asset of TVM is its so-called RPC interface to remotely cross-compile and run ML tasks on an edge device. In summary, the main reasons for selecting TVM as the enabling tool for leveraging hardware heterogeneity are 1) the generation of highly efficient code, 2) the compatibility with a great deal of hardware devices (thus becoming closer to the aforementioned general approach), and 3) its ease of use.

We evaluated heterogeneous CNN inference on the Nvidia Jetson TX2 board, a powerful edge device. This platform integrates an Nvidia Pascal GPU compatible with CUDA for intensive workloads and ARM processor cores. First, we assessed the performance of TVM on this platform. TVM allows CNN compilation with CUDA as the back-end. Alternatively, for CPU execution, LLVM can be also set as the target back-end of TVM. Inference runtimes for some popular CNNs running entirely either on the GPU or CPU of Jetson TX2 are reported in Fig. 1. Let us highlight two relevant aspects at this point:

- The inference performance on the two hardware back-ends available in Nvidia Jetson TX2 differs significantly.
- When deploying neural networks on edge platforms with TVM, a particular *target* hardware – CPU or GPU in our case – must be specified. However, this single-device approach does not permit optimal exploitation of heterogeneous hardware resources. For example, under some conditions, a neural network model may execute faster on the CPU than on the GPU (SqueezeNet in Fig. 1).

These two important factors motivated us to implement a hybrid co-execution method in order to 1) fully exploit Jetson TX2 resources, in contrast with realizations based on a single processor; 2) overcome the built-in assumption of TVM concerning single-device execution, which had precluded its application for CNN-based acceleration of video-stream processing based on heterogeneous co-execution. All in all, these are the contributions of this study:

- To the best of our knowledge, we propose the first fine-grain multi-back-end method that fully exploits the TVM compiler. It is an easy-to-implement strategy that sets the basis for advanced co-scheduling approaches on the multitude of devices compatible with TVM (in contrast to approaches relying on hardware-specific libraries such as ARM-CL).
- We experimentally demonstrate the benefits from exploiting heterogeneous edge platforms based on the proposed approach, with improvements up to 2.5× in throughput and energy reductions up to 66%.

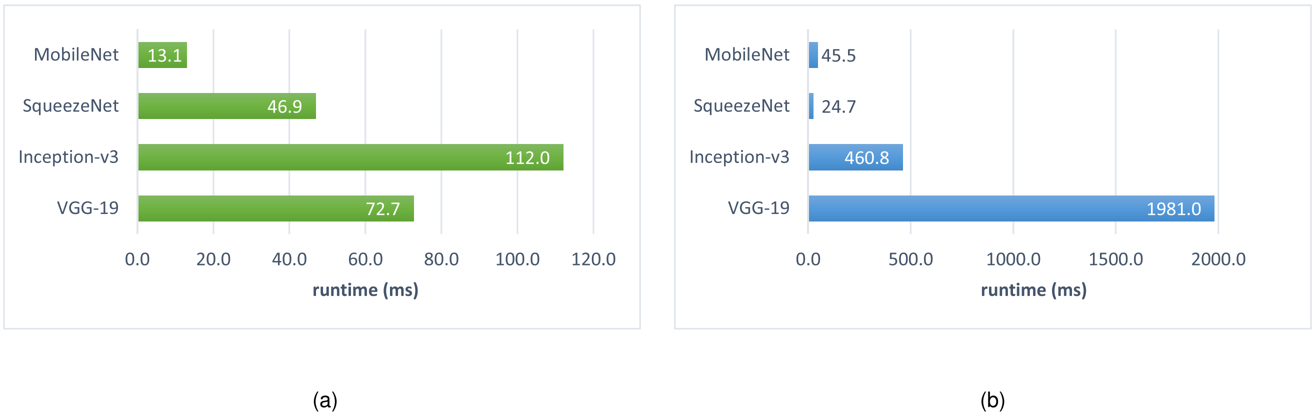


FIGURE 1. Inference performance of CNNs running entirely either on (a) the GPU, or (b) CPU of Jetson TX2.

Our extensive set of experiments encompasses five CNNs under two compiling settings while quantifying both runtime and energy.

- For standalone auto-tuned TVM kernels, we report optimal configurations on Jetson TX2 to achieve high-throughput or low-energy visual inference.
- We also demonstrate, through additional experiments conducted under several system states, the advantages of applying the proposed methodology to dynamic runtime scheduling.

II. PROPOSED APPROACH

In most application scenarios, high-performance visual analysis is required. These days, this means the realization of energy-efficient high-throughput CNN inference on input streams. Neural-network performance depends on both the network topology and its mapping on the hardware back-end, as evidenced by the dissimilar runtimes $t_{\mathcal{M}}$ shown in Fig. 1. Continuous CNN processing on a video stream is illustrated in Fig. 2a. In this case, the system processes the next image as soon as the current frame has been processed, delivering information at a rate of $T = 1/t_{\mathcal{M}}$ (processed images per second, img/s).[1]

In contrast to single-device execution setting, we propose a collaborative distributed scheme to accelerate network workloads, as illustrated in Fig. 2b and explained next.

- An s-stage network partition is built by splitting the CNN computational graph into s separate sub-graphs. Fig. 2b exemplifies two stages that, sequentially executed, are equivalent to the original network. Thus, network splitting does not modify network accuracy.
- Each sub-graph workload is compiled by TVM with a different hardware target. This way, we overcome the limitation of single-device execution on TVM.
- By pipelining sub-network execution on a stream, different hardware components collaboratively and concurrently execute network operators.

This unified scheme of co-execution can simultaneously exploit all processors on a board. Throughput acceleration depends on the slowest sub-graph $T = 1/\max_i\{t_{\mathcal{M}_i}\}$ (imgs/s). Next, we define a generic *graph-level* partitioning problem to optimize inference performance. Additionally, *kernel-level* optimization is automatically achieved using TVM autotuning, which renders highly efficient ML graphs [38].

A. PROBLEM FORMULATION

Deep neural networks are computational dataflows commonly expressed as directed acyclic graphs (DAGs). In DAGs, nodes represent operators/layers, whereas edges represent their dependencies. For instance, nodes in CNNs embody convolutional, fully-connected, or softmax layers, as well as their intrinsic parameters – kernel size, feature-map size, etc.

Given a network model $\mathcal{M}$ and a set of available hardware devices $h_i \in \mathcal{H}$, the objective is to split $\mathcal{M}$ into sub-graphs $\mathcal{M}_i \in \mathcal{M}$ running on heterogeneous devices to optimize particular performance metrics. After network mapping, each sub-graph $\mathcal{M}_i$ will contain a set of nodes from the original graph $\mathcal{M}$ without modifying it, i.e., $\mathcal{M}$ is unambiguously expressed as the concatenation of all $\mathcal{M}_i$.

In this paper we will assume that networks are sequential – or that they contain a limited number of branches with parallelism. This is indeed the case of many widely-used classification networks such as AlexNet [39], ResNets [40], MobileNets [41], [42], and VGG [43]. Many networks containing branches can be expressed as a chain; otherwise, the methodology proposed in this paper can still be applied, although the scheduling algorithm needs to be replaced by a more complex algorithm – scheduling algorithms are well researched [44], [45], [46]. In this work, we will show the advantages of the heterogeneous scheduling of neural network chains (models detailed in Sect. III-B).

[1] In this work, T denotes throughput.

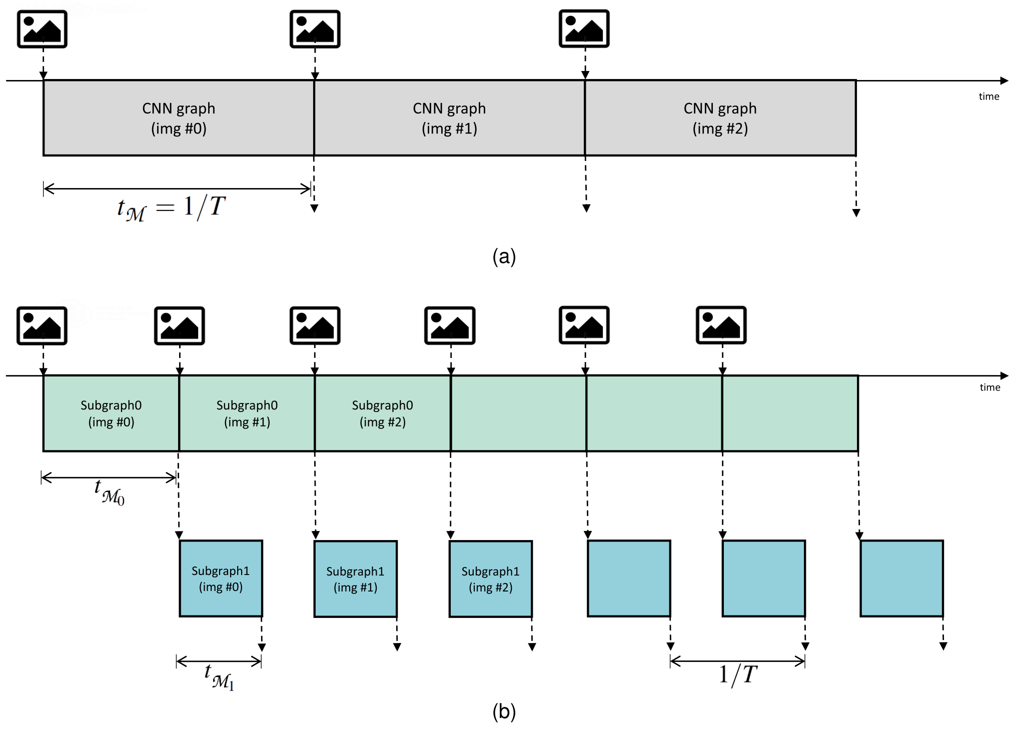


FIGURE 2. **CNN inference on input stream based on (a) single-device execution, or (b) pipeline scheme.**

As an example, the *two-stage network partition* shown in Fig. 2b schedules a model $\mathcal{M}$ with N sequential nodes over two devices $\{h_0, h_1\}$. Layers with indexes $\{0, 1, 2, \ldots, n-1\}$ are assigned to *graph0* and mapped into h_0, whereas *graph1* includes operators $\{n, n+1, \ldots, N-1\}$ running on h_1. Inference *throughput* optimization in this case involves finding the point $n \in [0, N-1]$ that minimizes the maximum runtime among all stages:

$$\hat{n} = \min_{n}[\max\{t^{h_0}_{\mathcal{M}_0}(n), t^{h_1}_{\mathcal{M}_1}(n)\}] \tag{1}$$

In this equation, runtimes are collected in advance through network profiling – see data collection and related issues in Sect. III-C.

Fig. 3 illustrates CNN inference on three consecutive frames in two scenarios: (i) single-stage graph execution, and (ii) two-stage network partition, exemplified with $n = 5$, $N = 8$. Intermediate data at layer $n-1$ must be transferred from the first processing device to the second one. These two scenarios of network workload distribution will be thoroughly studied in Sects. III–IV.

1) SCHEDULING SPACE

In a general case with multiple stages (s), the network partition is defined by a vector $\mathbf{n}$ of $s-1$ splitting points producing the sub-graphs $\mathcal{M}_i$. In total, an s-stage workload distribution with $N-1$ possible partition points renders up to $\binom{N-1}{s-1}$ possible distributions.[2]

[2]This binomial coefficient is obtained assuming N sequential nodes $\{l_0, l_1, \ldots, l_{N-1}\}$ in the DAG, and $s-1$ partition points which can be located at any of the $N-1$ intermediate positions.

Once defined the s-stage distribution, and given n_h available hardware devices, we can assign the s processing workloads to these processors in $(n_h)^s$ possible sequential processing orders. For instance, $h_2 \rightarrow h_0 \rightarrow h_1$, $h_0 \rightarrow h_2 \rightarrow h_1$, or $h_1 \rightarrow h_0 \rightarrow h_1$ are examples of device processing orders for a three-stage pipeline on three available devices.

All in all, the total number of possible configurations to distribute the workload over s stages is:

$$\binom{N-1}{s-1}(n_h)^s \tag{2}$$

Starting from $s = 1$, the greater the number of stages s, the greater the total number of possible configurations. To avoid combinatorial explosion in the number of configurations to be analyzed, in the next section we will show how to guide the search taking the memory transfer cost into account.

2) SCHEDULING SEARCH PROBLEM

In general, the problem is to find the set of partition points $\hat{\mathbf{n}}$ that optimize at least one of the following target metrics:

$$\text{Throughput: } \hat{\mathbf{n}} = \min_{\mathbf{n}}[\max_{i}\{t^{h_i}_{\mathcal{M}_i}(\mathbf{n})\}] \tag{3}$$

$$\text{Energy: } \hat{\mathbf{n}} = \min_{\mathbf{n}} \sum_{i} E^{h_i}_{\mathcal{M}_i}(\mathbf{n}) \tag{4}$$

$$\text{Latency: } \hat{\mathbf{n}} = \min_{\mathbf{n}} \sum_{i} t^{h_i}_{\mathcal{M}_i}(\mathbf{n}) \tag{5}$$

Note that in practice there will be a transfer cost due to copying data between the devices – symbolically represented in Fig. 3b with diagonal dashed arrows. So that, in the equations above, we are assuming that this transfer cost is already

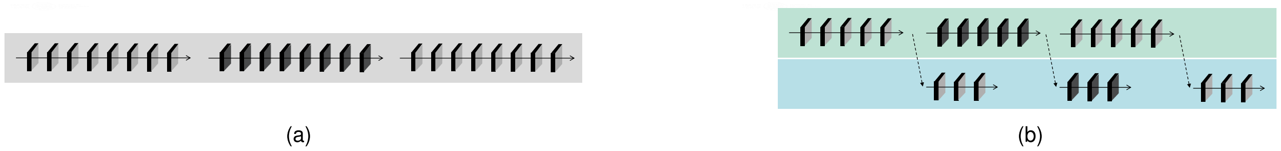


FIGURE 3. Network workload distribution for single- and two-stage approaches. CNN classification over three frames is shown for (a) single-stage execution on a particular hardware, or (b) two-stage concurrent execution on two hardware devices.

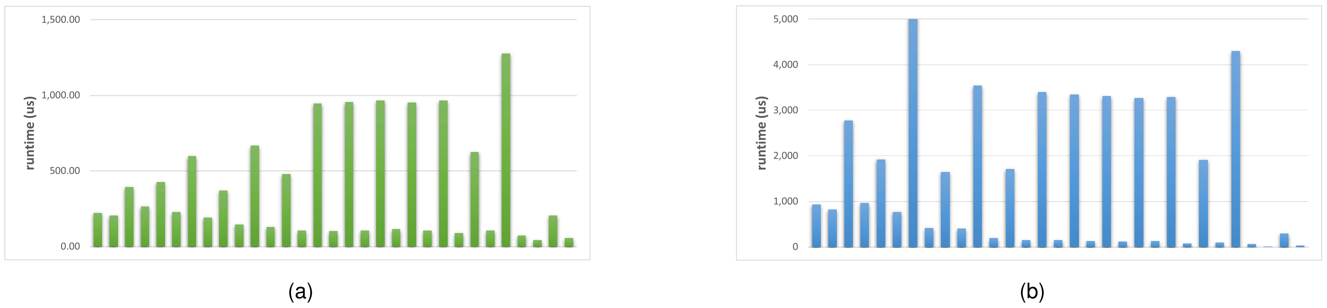


FIGURE 4. Layer-wise runtimes for MobileNet running entirely either on (a) the GPU, or (b) CPU of Jetson TX2.

included in the stage cost: $t^{h_i}_{\mathcal{M}_i}$ is the runtime of stage $\mathcal{M}_i$ running on device h_i which also includes the transfer cost to copy data to h_i. This extra cost $t^{h_i}_{transfer}$ depends on the transferred data, which will be different for each partition point and network architecture. To incorporate data-transfer cost in the problem, we can experimentally measure or estimate it – see Sect. III-C.

For such an optimization problem, an intuitive general approach consists of starting with $s = 1$ and finding the optimal solution. Then, we set $s = 2$, consider all possibilities, and we find the optimal solution again. Next, we successively minimize the function with increasing number of stages. We can introduce a heuristic stopping rule for the search. Given that the transfer cost is usually significant and given that the number of devices n_h is usually limited, eventually no better solution can be found for $s = \hat{s} + 1$ than for $s = \hat{s}$. At this point, it makes no sense to look for other solutions with more stages. In case of our heterogeneous GPU-CPU device, this will likely be the case for 2 stages (as exemplified in Sect. IV-C).

3) SCHEDULING ON TX2 USING TVM

Image-classification neural networks are composed of multiple layers that sequentially process data to produce an output vector from an input image. In most architectures, the first layers operate on data with higher dimensionality, whereas smaller feature maps are processed at ending layers. As a result, the computational demand varies among layers. This is why the performance profile strongly depends on the network architecture. As an example, Fig. 4 depicts the per-layer processing time for MobileNet [41] entirely running on one of the two processing components available in Jetson TX2, i.e., $\mathcal{H} = \{GPU, CPU\}$ ($n_h = 2$). Note that layer performance also depends strongly on the selected hardware back-end. Indeed, we will demonstrate that layer-wise operation distribution approach can benefit from high-performance computation on the GPU while offloading less computationally demanding layers to the CPU.

For instance, the simplest pipeline case is a two-stage network execution in which case the neural network is pipelined on CPU and GPU with only one memory transfer. In this case, once we know the layer-wise execution runtime profile $\{t^{h_i}_0, t^{h_i}_1, \ldots, t^{h_i}_{N-1}\}$ of the CNN on both processors $h_0 = GPU$, $h_1 = CPU$ (Fig. 4), we can estimate sub-graph latencies:

$$t^{h_0}_{\mathcal{M}_0}(n) = \sum_{l=0}^{n-1} t^{h_0}_l + t^{h_0}_{transfer} \tag{6}$$

$$t^{h_1}_{\mathcal{M}_1}(n) = \sum_{l=n}^{N-1} t^{h_1}_l + t^{h_1}_{transfer} \tag{7}$$

We maximize throughput by solving (1). Then, the expected throughput (T in img/s) and latency (L in seconds), respectively, of the optimized 2-stage configuration running on $h_0 \to h_1$ are:

$$T = \frac{1}{\max\{t^{h_0}_{\mathcal{M}_0}(\hat{n}), t^{h_1}_{\mathcal{M}_1}(\hat{n})\}} \tag{8}$$

$$L = t^{h_0}_{\mathcal{M}_0}(\hat{n}) + t^{h_1}_{\mathcal{M}_1}(\hat{n}) \tag{9}$$

This throughput optimization problem for $s = 2$ can be solved with Algorithm 1. This algorithm assumes that we have collected runtime profiling and we have estimated transfer costs. It also returns the optimal device order $\hat{h}_0 \to \hat{h}_1$, e.g., GPU → CPU. Because (6) and (7) can be pre-computed in $O(N)$ and because the evaluation of (8) is achieved in $O(N)$,

TABLE 2. Assessed neural networks.

Network	# TVM layers	# modules
MobileNet [41]	31	27 conv + 1 pooling + 1 flatten + 1 FC + 1 softmax
SqueezeNet-v1.1 [47]	40	2 conv + 4 pooling + 8 Fire + 1 flatten + 1 softmax
Inception-v3 [48]	122	5 conv + 11 Inception + 3 pooling + 1 flatten + 1 FC + 1 softmax
VGG-19 [43]	26	16 conv + 5 pooling + 1 flatten + 3 FC + 1 softmax

the algorithmic time complexity is $O(N)$ as well. The algorithm starts by assuming a processing order $h'_0 \rightarrow h'_1$. Then, for each possible partition point n, it calculates both sub-graph runtimes $t^{h'_0}_{\mathcal{M}_0}(n)$, $t^{h'_1}_{\mathcal{M}_1}(n)$ and saves the pipeline runtime $\max\{t^{h'_0}_{\mathcal{M}_0}(n), t^{h'_1}_{\mathcal{M}_1}(n)\}$. After analyzing all possible device orders, it finally obtains the minimum runtime over all cases.

Algorithm 1 Two-Stage Network Partitioning

```
Input: $\{t_l\}^{h_0}, \{t_l\}^{h_1}, \{t_{transfer,l}\}^{h_0}, \{t_{transfer,l}\}^{h_1}$
Output: $\hat{n}, \hat{t}, \hat{h_0}, \hat{h_1}$
  $\hat{t} \leftarrow \infty$
  for $(h'_0, h'_1)$ in all combinations($\{h_0, h_1\}$) do
    $t_{\mathcal{M}_0} \leftarrow 0$
    $t_{\mathcal{M}_1} \leftarrow \sum_i t_i^{h'_1}$
    for $i \leftarrow 0$ to $N-1$ do
      $t_{\mathcal{M}_0} \leftarrow t_{\mathcal{M}_0} + t_i^{h'_0}$          /* Partition 0: [0,i] */
      $t_{\mathcal{M}_1} \leftarrow t_{\mathcal{M}_1} - t_i^{h'_1}$          /* Partition 1: [i+1,N-1] */
      $m[i] \leftarrow \max\{t_{\mathcal{M}_0} + t^{h'_0}_{transfer,i}, t_{\mathcal{M}_1} + t^{h'_1}_{transfer,i}\}$
    end for
    if $\min_i m[i] < \hat{t}$ then
      $\hat{t} \leftarrow \min_i m[i]$   /* new optimal found for $h'_0 \rightarrow h'_1$ */
      $\hat{n} \leftarrow \arg\min_i m[i]$
      $\hat{h_0} \leftarrow h'_0$
      $\hat{h_1} \leftarrow h'_1$
    end if
  end for
  return $\hat{n}, \hat{t}, \hat{h_0}, \hat{h_1}$
```

III. EXPERIMENTAL PROCEDURE

A. HARDWARE AND SOFTWARE

The selected system, Nvidia Jetson TX2, was designed to jointly achieve power efficiency and high performance for different embedded applications. It is a popular commercial board that integrates a 256-core Nvidia Pascal GPU and a 64-bit ARMv8 general-purpose multi-processor. The memory and storage included in the platform are 32-bit 8GB LPDDR4 and 32GB eMMC 5.1, respectively. A camera expansion connector is included to endow the system with on-board video analysis capabilities. The Jetson TX2 platform was selected due to the (compared to a desktop CPU and GPU) relatively low power consumption (7.5W – 15W), while still providing sufficient throughput for many deep learning applications.

Using TVM on this platform, we can (1) define, (2) tune, (3) compile, and (4) run ML models:

1) ML models are internally represented in TVM as computational graphs in a so-called *Tensor Expression* language. Users can define models in an intermediate representation on TVM, or alternatively networks can be loaded from files compatible with other ML frameworks, such as TensorFlow, Keras, MXNet, PyTorch, or DarkNet.
2) TVM integrates auto-tuner tools to generate efficient code for the targeted platform [37], [38]. Specifically, the results presented in this paper regarding auto-tuned models were compiled with Ansor [38], [49]. This tool finds the best compiling options for each back-end to reduce the corresponding runtime – see Sect. IV-A.
3) TVM compiles graphs to generate the C++ code that will run on the *target* hardware. We used the TVM Python interface, which greatly facilitates network definition and execution (other available interfaces are TVMC [50] and C++).

B. NEURAL NETWORKS

Table 2 summarizes the architecture of the CNNs considered in this study. We used the TVM definitions of these networks [51]. TVM can process them using two types of data orders called NHWC and NCHW. Experiments in this work refer to channel-first (NCHW) layout. Layers in the network are represented by nodes in the graph, sequentially connected to compose the overall architecture. Although the networks may contain modules with parallel branches – as in the *Fire* module of SqueezeNet and *Inception* of Inception-v3 –, they are treated as individual nodes in the proposed procedure. Therefore, '# modules' in Table 2 refers to the number of sequential nodes N in the graph. Note that even if these models contain layers other than those listed in Table 2, such as ReLU, batch normalization, or dropout, TVM automatically optimizes the computational graph by fusing these operations with the previous layer.

C. LAYER-WISE PERFORMANCE PROFILING

In short, the proposed method comprises a first stage of layer-level network performance profiling and a second stage of operator mapping to the available hardware components. Concerning the first stage, for each hardware component h_i, we must obtain a performance profile of the N nodes in the model. Then, in the second stage, we solve the network partition problem for any of the equations (3)–(5). We assume that the neural-network layer execution performance is independent of the input data, that is, we do not consider dynamic neural networks and influences of L1/L2 caching are ignored.

- **Runtime**. Throughput optimization in Jetson TX2 means solving (1) (or (3), more generally) with the profiling of the network expressed as $\{t_l\}^{GPU}$, $\{t_l\}^{CPU}$ $\forall l \in [0, N-1]$. An example was depicted in Fig. 4 for MobileNet; this figure clearly exemplifies how a pipelined approach distributing operations over the GPU and CPU in TX2 benefits from their dissimilar performance. The operation of Algorithm 1 solving (1) for a cooperative GPU $\rightarrow$ CPU execution is illustrated in Fig. 5, which shows the runtime $t_{\mathcal{M}_i}$ of the stage as a function of the selected partition point n. Up to $N+1$ solutions are possible, being the optimal one $\hat{n} = 23$, which means that 23 nodes run on the GPU and the rest on the CPU.

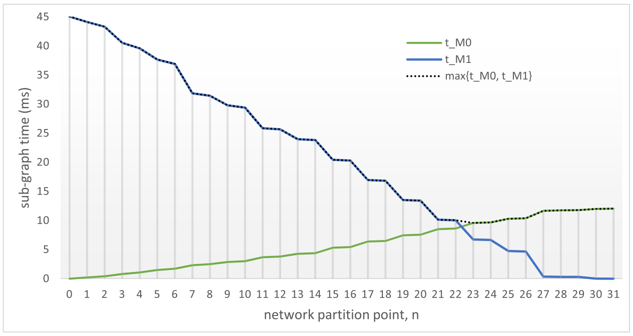


FIGURE 5. Throughput optimization for a two-stage pipeline on MobileNet running in Jetson TX2 with order GPU → CPU. Sub-graph execution runtimes $t_{\mathcal{M}_i}(n)$ depend on the selected partition point n.

  – As previously stated, data-transfer latency can be incorporated into the scheduling model. Even though the GPU memory is integrated in the CPU memory in TX2 through CUDA, the TVM implementation (and many other implementations) uses distinct memory regions for CPU and GPU, still requiring a transfer between both. Several techniques exist to mitigate or reduce the data transfer latency, but none of them totally eliminates the cost. For example, the transfer latency can be improved by using CUDA pinned host memory, but due to the lack of hardware support for cache coherence between the CPU and GPU caches of the Jetson TX2, a costly synchronization step (in principle, a call to the `cudaDeviceSynchronize` or `cudaStreamSynchronize` function) is needed, leading again to a performance cost. Another solution is to use CUDA unified memory; in this case, the transfer is handled transparently by the GPU driver, but there is still a cost (which manifests as a slower execution of the kernel function of the particular neural network layer(s), as well as the need for calling the `cudaDeviceSynchronize` function). In addition, by using multiple streams, CUDA allows overlapping of memory transfers with the actual computations, reducing the transfer costs. At the time of writing, the TVM support of this feature is in an initial stage (see e.g., [52]). However, it is generally known that this technique does not reduce the transfer cost significantly in most practical cases with a high GPU kernel occupancy.

  To experimentally characterize this transfer overhead, we could assume linear dependence w.r.t. the size of the transferred data $t_{transfer} = k \cdot B$ (where k is an arbitrary constant that depends on the considered hardware, and B is the number of bytes). We thoroughly measured $t_{transfer}$ in TX2, including several batch sizes and CNN models in experimental data. In particular, we measured the time cost of the `set_input` function of the `GraphModule` TVM python class, which transfers data to the CNN and incurs high costs. The results confirmed this linear dependence – see Fig. 6. Indeed, the methodology assessment presented in Sects. IV–V further validated this assumption.

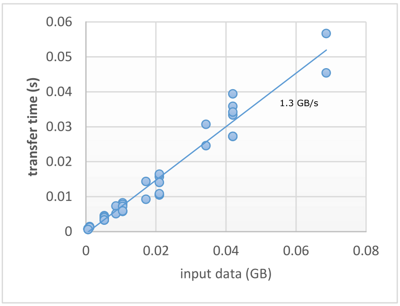
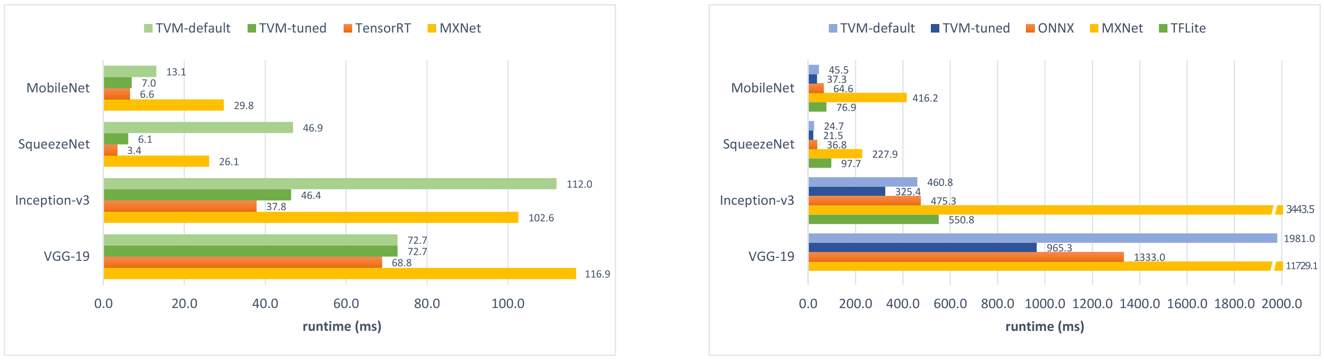


FIGURE 6. Data transfer time overhead on TVM-Python.

- **Energy**. The main power supply to the TX2 module is provided by a single DC input. According to its documentation [53], the voltage of this DC input ranges from 5.5 V to 19.6 V, with a maximum current of 4000 mA. Two INA3221 power monitors are included in the TX2 carrier board. We measured the per-layer power demand of CNNs, $\{P_l\}^{h_i}$, using the TVM debugger and software included in Tegra Linux Driver Package [54].

  – When it comes to taking accurate power consumption measurements, the limited time resolution of the INA3221 sensors plays a major role. Therefore, mismatches between per-layer and overall CNN power consumption are common [55], [56], [57]. For instance, Cai et . [56] showed that per-layer energy profiling is lower than actual CNN energy consumption in all of their study cases. An interesting aspect highlighted in [57] is that monitoring system-level power consumption (instead of taking separate GPU and CPU measurements) renders



(a) (b)

FIGURE 7. Inference performance of various software frameworks for running CNNs on TX2, and comparison with TVM. (a) CNN execution on GPU, and (b) CNN execution on CPU.

better estimations because, among other aspects, the energy consumed by memory accesses is also taken into account. This is why we measured power at system level. Interestingly, the aforementioned approaches reported in the literature do not try to correct measurement errors because estimating the energy-consumption trend is enough, instead of finding exact energy values. As we will discuss in Sect. V, the proposed scheduling model is indeed able to find the optimal cases.

IV. PERFORMANCE EVALUATION

A. SINGLE-DEVICE EXECUTION

The baseline performance of network execution on each hardware component of the platform (i.e., on each TVM *target*) is shown in Fig. 1. Notwithstanding, as previously pointed out, one of the advantages of TVM is its open-source auto-tuning tool, which generates highly efficient code and boosts inference performance. In this regard, we measured a performance improvement (up to 87%) in terms of runtime between *(i) default* TVM network execution (Fig. 1), and *(ii)*[3] *tuned* network execution (using Ansor [38]), as reported in Fig. 7.

Furthermore, for a baseline performance reference on our platform, Fig. 7 also presents a inference runtime comparison that includes other software frameworks. The performance of each library depends on the architecture of the DL model [58], [59]. TVM definitively constitutes as a promising general framework.

B. TWO-STAGE EXECUTION

Once the layer-wise performance measurements were taken (as explained in Sect. III-C), performance optimization was conducted according to (3)–(5). To define network sub-graphs, we edited the pre-defined networks [51] in TVM.

[3]We will use "*(i)*" to denote networks compiled with TVM by default and "*(ii)*" for models compiled with compiling optimizations produced by Ansor auto-tuning tool.

The proposed co-execution pipeline scheme was enabled by developing TVM-Python code to compile network sub-graphs and continuously offload the corresponding processing stage to each back-end. This code of ours is available in [60].

1) THROUGHPUT-OPTIMIZED CONFIGURATIONS

Tables 3–4 report the resulting device mapping with highest inference rate. The last two columns confirm that the 'expected' performance (from the optimization problem defined by (3) and network profiling data) is remarkably close to the experimental measurements. 'Measured' experimental performance is also reported in Fig. 8, with additional comparison with single-device execution. Interestingly, heterogeneous hardware exploitation achieves throughput improvements w.r.t. the best single-device case ranging from 22% to 150% for models compiled by default; and up to 22% for auto-tuned models. Only for tuned MobileNet and SqueezeNet, the performance resulting from the proposed two-stage approach remains similar to GPU-only execution, as otherwise analytically expected.

2) ENERGY-OPTIMIZED CONFIGURATIONS

GPU-only execution is the most energy-efficient configuration for all the networks except for *default* SqueezeNet. The underlying reason is the combination of reduced runtime on GPU with reduced data-transfer energy cost. In the particular case of *default* SqueezeNet, it is optimized in terms of both frame rate and energy when using a heterogeneous GPU $\rightarrow$ CPU pipeline with $\hat{n} = 32$. Experimental energy-consumption measurements agree with these analytical expectations, as will be demonstrated in Sect. V.

3) LATENCY-OPTIMIZED CONFIGURATIONS

Similarly, GPU-only execution is the best configuration for fast-response applications. Optimal latency metrics correspond to those shown in Fig. 7a.

TABLE 3. **Throughput-optimized configurations – (i) *default* TVM models.**

Network	Configuration (# layers)	Expected T (img/s)	Measured T (img/s)
MobileNet	23 GPU $\rightarrow$ 8 CPU	100.0 img/s	98.0 $\pm$ 2.3 img/s
SqueezeNet-v1.1	32 GPU $\rightarrow$ 8 CPU	100.2 img/s	101.3 $\pm$ 6.6 img/s
Inception-v3	88 GPU $\rightarrow$ 34 CPU	9.5 img/s	10.9 $\pm$ 0.2 img/s
VGG-19	20 GPU $\rightarrow$ 6 CPU	17.9 img/s	16.0 $\pm$ 0.2 img/s

TABLE 4. **Throughput-optimized configurations – (ii) *tuned* TVM models.**

Network	Configuration (# layers)	Expected T (img/s)	Measured T (img/s)
MobileNet	4 CPU $\rightarrow$ 27 GPU	167.9 img/s	139.9 $\pm$ 2.8 img/s
SqueezeNet-v1.1	6 CPU $\rightarrow$ 34 GPU	175.0 img/s	161.2 $\pm$ 11.9 img/s
Inception-v3	5 CPU $\rightarrow$ 117 GPU	20.9 img/s	24.5 $\pm$ 0.9 img/s
VGG-19	20 GPU $\rightarrow$ 6 CPU	17.9 img/s	16.8 $\pm$ 0.1 img/s

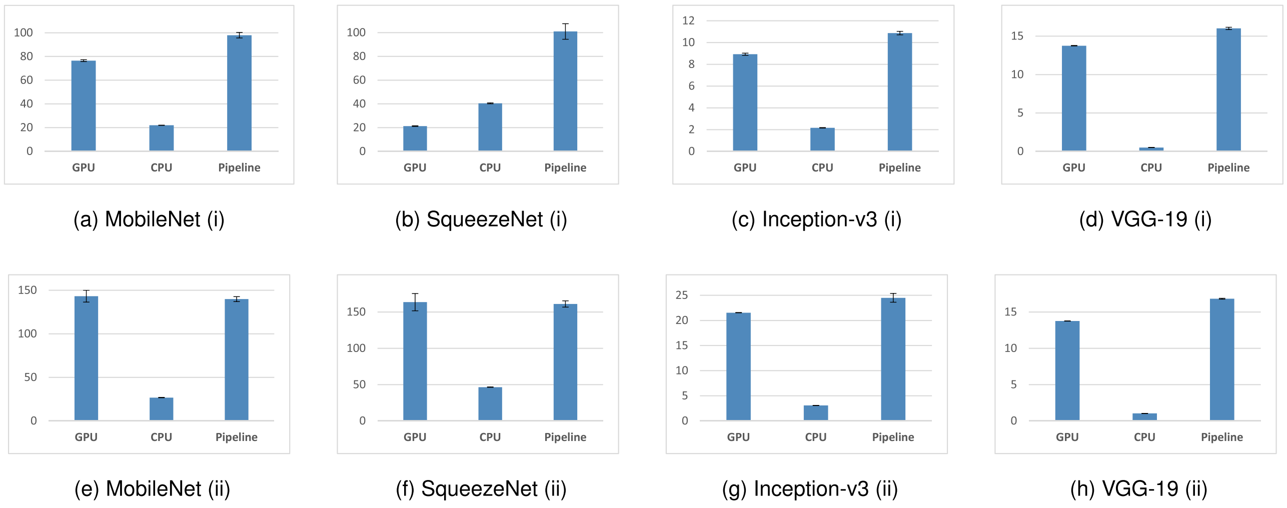


FIGURE 8. **Throughput (img/s) comparison of one-device execution vs. throughput-optimized cases. Four CNN models were assessed (i) without and (ii) with TVM auto-tuning.**

4) GENERAL TRADE-OFFS

Finally, Fig. 9 shows performance metrics for each CNN under various configurations. Single-device execution is presented with green markers for GPU, and blue ones for CPU. Energy-optimized and latency-optimized configurations match with GPU execution – in green – (except for *default* SqueezeNet). Two-stage scheduling for optimal throughput is shown in black. A particularly remarkable case is precisely *default* SqueezeNet, whose throughput increases 150% when running upon a heterogeneous scheme, with an energy reduction of 66%.

The proposed scheme found the energy-optimized cases, although no hybrid execution might be needed. If we choose appropriate lowest-energy configurations, we obtain energy savings above 10% w.r.t. best-throughput cases. Concerning latency, its behaviour is similar to energy in all cases (x-axis in Fig. 9): pipelining slightly increases latency, being GPU the lowest-latency configuration in most cases.

We conclude that the proposed optimization approach leads to throughput enhancement, especially when runtime on each device match with each other.

C. K-STAGE EXECUTION

In the previous results, we considered 2 stages. However, mathematically, better solutions for more than 2 stages might be possible. Fig. 10 exemplifies an optimal 3-stage pipeline (e.g., CPU $\rightarrow$ GPU $\rightarrow$ CPU). However, network scheduling with more than one partition point requires 1) execution times on each device matching with each other, thus avoiding timeline gaps (in this example, $t^{CPU}_{\mathcal{M}_0} + t^{CPU}_{\mathcal{M}_2} \approx t^{GPU}_{\mathcal{M}_1}$), and 2) limited transfer costs (the greater the number of stages, the larger the increase in latency). These assumptions are difficult to meet in practice; and indeed we confirmed it experimentally. In general, achieving optimal work balancing on pipelines with more than two stages requires joint kernel code and scheduling optimization. This will be part of our future work.

V. DISCUSSION

The tests conducted on networks schedules resulting from the optimization problem posed in Sect. II-A validated the expected performance metrics. However, the proposed method relies on experimental measurements, which vary

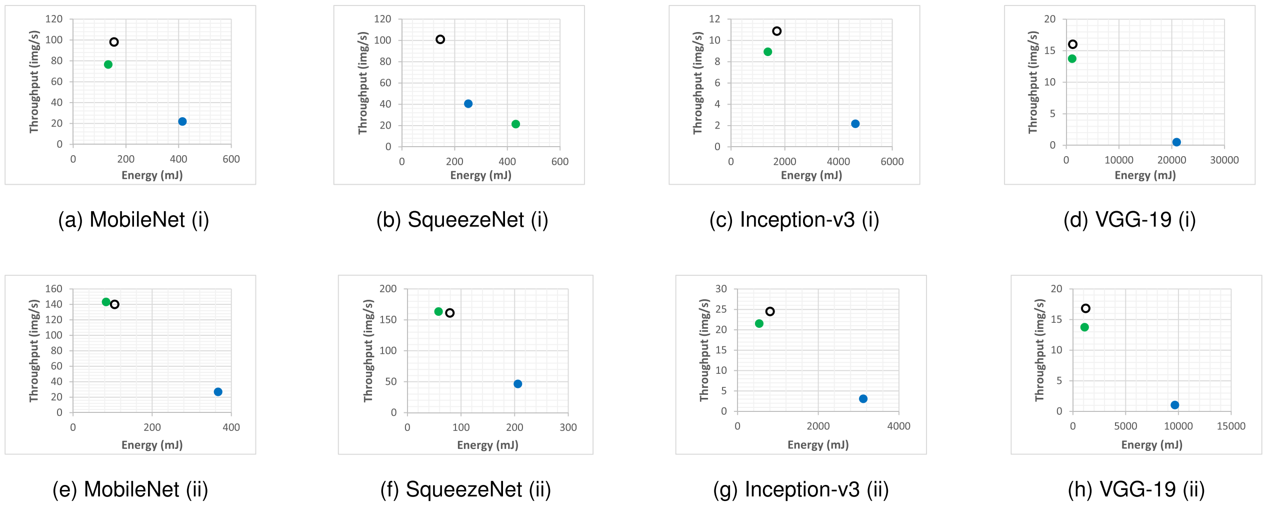


FIGURE 9. Performance trade-offs for GPU-only (green), CPU-only (blue), and throughput-optimized pipeline (black) for the CNNs considered in this study;"*(i)*" denotes networks compiled with TVM by default whereas "*(ii)*" denotes models compiled with compiling optimizations produced by Ansor auto-tuning tool.

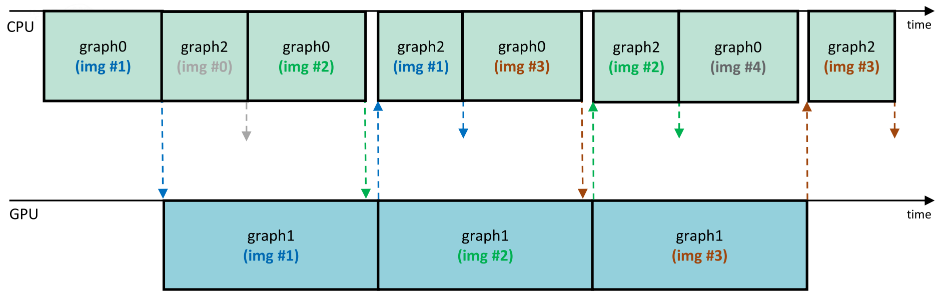


FIGURE 10. Exemplary optimal CNN workload distribution over a 3-stage pipeline.

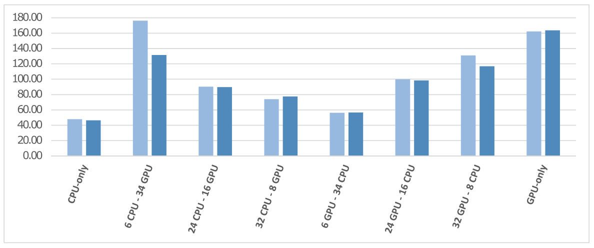
among different executions, and depends on per-layer profiling, which may differ from actual network performance [9], [55], [56], [57] – see related comments in Sect. III-C. This hinders the capability of the proposed procedure to produce accurate estimates of performance metrics. This is why all performance metrics reported in this paper were averaged over several executions. We conducted additional experiments to verify that these issues do not greatly affect the scheduling optimization results. While our pipeline scheduler does not provide exact performance values, our experimental findings did not reveal any better solutions than those deemed optimal by our method. Fig. 11 shows the performance of eight different network schedules for tuned SqueezeNet (3 CPU $\rightarrow$ GPU, 3 GPU $\rightarrow$ CPU, CPU-only, and GPU-only). The 'estimated' performance in these plots is obtained from (3)–(5) according to per-layer profiling measurements. Note that per-layer profiling measurement errors in single-device execution (CPU-, GPU-only) can certainly impact the optimization problem and the corresponding estimations. Notwithstanding, for the eight studied cases, although predicted values do not exactly match experimental ones, the method correctly estimates high- and low-throughput cases, as well as high- and low-energy configurations. Indeed, we found no case in which the proposed optimization procedure were unpredictably outperformed. Therefore, we convincingly state that predicted 'optimal' cases actually match with the best performance configurations when experimentally assessed.

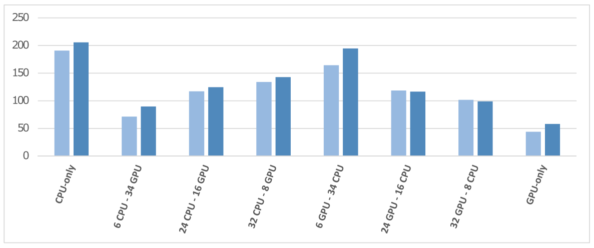
Concerning energy consumption, minimizing the number of memory transfers is expected to minimize the energy as well. Indeed, we found in our experiments that the energy-optimal configuration coincides in most cases with GPU-only networks execution – Sect. IV-B. However, adding energy transfer costs in (4) would further improve our scheduling method in general cases.

A. DYNAMIC RUNTIME SCHEDULING

As mentioned in Sect. I, the proposed TVM co-execution scheme makes it possible to select the appropriate scheduling according to the current system state, in particular, for repeated neural network inferences (e.g., when processing video frames in real-time).

For instance, Algorithm 2 exemplifies a method to dynamically select the optimal partition point when there is limited


(a) Throughput (img/s)


(b) Energy (mJ)

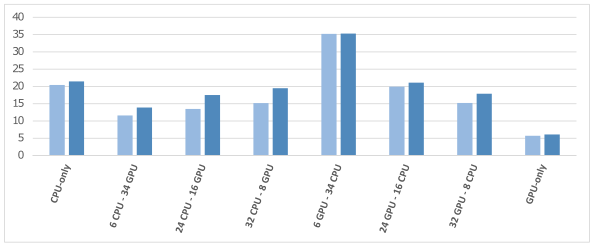

(c) Latency (ms)

FIGURE 11. Experimental performance (darker bars) vs. expected performance from per-layer profiling data (lighter bars) for several network partition schedules on SqueezeNet.

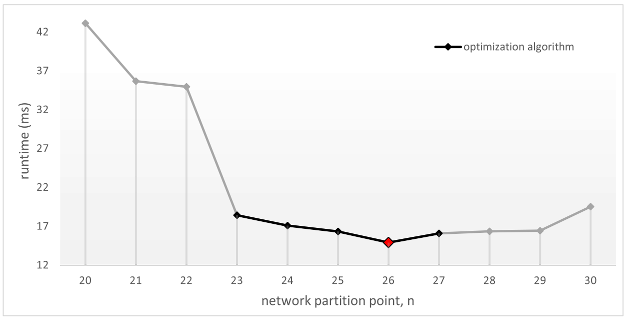


FIGURE 12. Pipeline execution time as a function of *n*. Black points represent dynamic scheduling for a two-stage pipeline on MobileNet according to Algorithm 2. Best runtime found is marked in red. Performance of additional non-optimal pipeline schedules is shown in gray dots for the sake of completeness, but they were not explored using Algorithm 2.

availability of computational resources. This method depends on measurements of the pipeline $\mathcal{M}_0^{h_0}(n) \rightarrow \mathcal{M}_1^{h_1}(n)$ for each n. To better deal with transients (sudden temporary changes in the processing), we can use a recursive moving averaging filter on these measurements (depending on parameter α in Algorithm 2). The method starts by selecting an arbitrary partition point $n \in [0, N]$; for instance, the initial value of n can be the optimal configuration when the system is not executing extra workloads – i.e., configurations $\mathcal{M}_0^{h_0}(n) \rightarrow \mathcal{M}_1^{h_1}(n)$ reported in Tables 3–4, with runtimes t. Then, it dynamically increases (or decreases) n until no better execution time is found (local minimum). Note that other optimizations algorithms could be applied that leverage our heterogeneous pipeline.

We built TVM pipelines – as explained in Sect. IV-B – to test this dynamic method. In particular, we performed experiments on MobileNet under different scenarios of system resources availability. More specifically, our simulations included (a) reading and displaying a MP4 video file on a monitor with hardware acceleration enabled; (b) executing the `glmark2` benchmark; (c) executing the `stress` command; and (d) executing the `glxgears` demo application. Table 5 reports the best solutions found for these cases. Note that single-device execution is compared with the best heterogeneous pipeline. For further clarification, Fig. 10 shows

TABLE 5. Single-processor cases vs. throughput-optimized configurations under dynamic scheduling for MobileNet.

Experiment	GPU-only	CPU-only	Pipeline	
(0) Baseline case	76.6 imgs/s	22.0 imgs/s	23 GPU $\rightarrow$ 8 CPU	98.0 img/s
(a) Video player	70.0 imgs/s	17.1 imgs/s	28 GPU $\rightarrow$ 3 CPU	77.6 img/s
(b) glmark2	51.4 imgs/s	15.6 imgs/s	26 GPU $\rightarrow$ 5 CPU	67.1 img/s
(c) stress	80.0 imgs/s	8.7 imgs/s	29 GPU $\rightarrow$ 2 CPU	88.4 img/s
(d) glxgears	67.1 imgs/s	15.5 imgs/s	26 GPU $\rightarrow$ 5 CPU	76.8 img/s

Algorithm 2 Dynamic Two-Stage Network Partitioning

```
Input: $n$, $h_0$, $h_1$, $t$, $\mathbf{m}$        /* $n$ layers $h_0 \rightarrow$ (N-n) layers $h_1$ */
Output: $\hat{n}$, $\hat{t}$
  $\hat{t} \leftarrow t$
  /* Increase n until finding minimum: */
  for $i \leftarrow n$ to $N$ do
    $m[i] \leftarrow m[i] + \alpha * measure\{t_{\mathcal{M}_0(i)\rightarrow\mathcal{M}_1(i)}\}$
    if $m[i] \leq \hat{t}$ then
      $\hat{n} \leftarrow i$                  /* new optimal found */
    else
      break for
    end if
  end for
  /* If no better solution was found, then decrease n: */
  if $\hat{n} = n$ then
    for $i \leftarrow n - 1$ to 0 do
      $m[i] \leftarrow measure\{t_{\mathcal{M}_0(i)\rightarrow\mathcal{M}_1(i)}\}$
      if $m[i] \leq \hat{t}$ then
        $\hat{n} \leftarrow i$                /* new optimal found */
      else
        break for
      end if
    end for
  end if
  $\hat{t} \leftarrow m[\hat{n}]$
  return $\hat{n}$, $\hat{t}$, $m$
```

partial results $m[i]$ from executing Algorithm 2 in Case (b). It starts with $n = 23$ and stops in $n = 27$, being $\hat{n} = 26$ the optimal partition point found (see Table 5).

Similar to the results previously presented under full resource availability, the proposed heterogeneous pipeline keeps outperforming single-device execution for Cases (a)–(d).

VI. CONCLUSION

In this paper, we explored TVM as a tool for joint exploitation of different processing resources available in edge devices. Hardware compatibility, easy implementation, and efficient compiled models are remarkable assets of the proposed approach. A large variety of tests demonstrate that an optimal co-schedule boosts the performance of edge vision systems in terms of throughput. We also confirmed that the proposed approach can automatically identify whether a particular pipeline schedule outperforms single-device execution in terms of energy or latency.

This work sets the basis for extending this method to other platforms, such as multi-GPU boards, GPU-FPGA systems, or host-accelerator systems. In future work, we will investigate platforms providing support for integer operation to analyze how integer quantization can improve inference performance while affording a small loss in accuracy. We will also explore the flexibility offered by the Nvidia *nvpmodel* tool to achieve performance-power trade-offs by adjusting the GPU and CPU operation frequencies. We will also address the implementation of the method hereby reported in a fully automated manner. A model to predict layer performance from the number of floating-point operations will be also explored. In addition, heterogeneous scheduling with more than 2 stages is also of interest, especially for multi-GPU configurations. This requires joint kernel code and schedule optimization in order to achieve an optimal workload distribution, as exemplified in Fig. 10. Finally, an interesting aspect to be pointed out is that our energy-throughput optimized solution enables us to accommodate application requirements in real time (energy-saving or fast-response) by selecting the appropriate co-execution scheduling.

REFERENCES

[1] NVIDIA Developer. *Jetson Modules*. Accessed: Dec. 16, 2022. [Online]. Available: https://developer.nvidia.com/embedded/jetson-modules

[2] NVIDIA Developer. *Jetson Nano Developer Kit*. Accessed: Dec. 16, 2022. [Online]. Available: https://developer.nvidia.com/embedded/jetson-nano-developer-kit

[3] NVIDIA Developer. *Jetson Xavier NX Developer Kit*. Accessed: Dec. 16, 2022. [Online]. Available: https://developer.nvidia.com/embedded/jetson-xavier-nx-devkit

[4] A. Shahid and M. Mushtaq, "A survey comparing specialized hardware and evolution in TPUs for neural networks," in *Proc. IEEE 23rd Int. Multitopic Conf. (INMIC)*, Nov. 2020, pp. 1–6.

[5] M. Cloutier, C. Paradis, and V. Weaver, "A Raspberry Pi cluster instrumented for fine-grained power measurement," *Electronics*, vol. 5, no. 4, p. 61, Sep. 2016. [Online]. Available: https://www.mdpi.com/2079-9292/5/4/61

[6] K. Seshadri, B. Akin, J. Laudon, R. Narayanaswami, and A. Yazdanbakhsh, "An evaluation of edge TPU accelerators for convolutional neural networks," 2021, *arXiv:2102.10423*.

[7] S. S. Ogden and T. Guo, "Characterizing the deep neural networks inference performance of mobile applications," 2019, *arXiv:1909.04783*.

[8] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *Proc. 22nd Int. Conf. Architectural Support Program. Lang. Operating Syst.* New York, NY, USA: Association for Computing Machinery, Apr. 2017, pp. 615–629, doi: 10.1145/3037697.3037698.

[9] S. Zhang, Y. Li, X. Liu, S. Guo, W. Wang, J. Wang, B. Ding, and D. Wu, "Towards real-time cooperative deep inference over the cloud and edge end devices," *Proc. ACM Interact., Mobile, Wearable Ubiquitous Technol.*, vol. 4, no. 2, pp. 1–24, Jun. 2020.

[10] A. E. Eshratifar and M. Pedram, "Energy and performance efficient computation offloading for deep neural networks in a mobile cloud computing environment," in *Proc. Great Lakes Symp. VLSI (GLSVLSI)*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 111–116.

[11] L. Zeng, E. Li, Z. Zhou, and X. Chen, "Boomerang: On-demand cooperative deep neural network inference for edge intelligence on the industrial Internet of Things," *IEEE Netw.*, vol. 33, no. 5, pp. 96–103, Sep./Oct. 2019.

[12] E. Li, Z. Zhou, and X. Chen, "Edge intelligence: On-demand deep learning model co-inference with device-edge synergy," 2018, *arXiv:1806.07840*.

[13] I. Prakash, A. Bansal, R. Verma, and R. Shorey, "SmartSplit: Latency-energy-memory optimisation for CNN splitting on smartphone environment," 2021, *arXiv:2111.01077*.

[14] F. M. C. de Oliveira and E. Borin, "Partitioning convolutional neural networks to maximize the inference rate on constrained IoT devices," *Future Internet*, vol. 11, no. 10, p. 209, 2019. [Online]. Available: https://www.mdpi.com/1999-5903/11/10/209

[15] C. Hu, W. Bao, D. Wang, and F. Liu, "Dynamic adaptive DNN surgery for inference acceleration on the edge," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr./May 2019, pp. 1423–1431.

[16] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Toward collaborative inferencing of deep neural networks on Internet-of-Things devices," *IEEE Internet Things J.*, vol. 7, no. 6, pp. 4950–4960, Jun. 2020.

[17] X. Wang, Y. Han, C. Wang, Q. Zhao, X. Chen, and M. Chen, "In-edge AI: Intelligentizing mobile edge computing, caching and communication by federated learning," 2018, *arXiv:1809.07857*.

[18] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "DeepThings: Distributed adaptive deep learning inference on resource-constrained IoT edge clusters," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2348–2359, Nov. 2018.

[19] L. Zhou, M. H. Samavatian, A. Bacha, S. Majumdar, and R. Teodorescu, "Adaptive parallel execution of deep neural networks on heterogeneous edge devices," in *Proc. 4th ACM/IEEE Symp. Edge Comput. (SEC)*. New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 195–208.

[20] Z. Xu, D. Yang, C. Yin, J. Tang, Y. Wang, and G. Xue, "A co-scheduling framework for DNN models on mobile and edge devices with heterogeneous hardware," *IEEE Trans. Mobile Comput.*, vol. 22, no. 3, pp. 1275–1288, Mar. 2023.

[21] E. Aghapour, A. Pathania, and G. Ananthanarayanan, "Integrated ARM big.Little-Mali pipeline for high-throughput CNN inference," Univ. Amsterdam, Amsterdam, The Netherlands, Tech. Rep. TCAD-2021-0385, Jul. 2021, doi: 10.36227/techrxiv.14994885.v2.

[22] S. Wang, G. Ananthanarayanan, Y. Zeng, N. Goel, A. Pathania, and T. Mitra, "High-throughput CNN inference on embedded ARM big.LITTLE multicore processors," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 10, pp. 2254–2267, Oct. 2020.

[23] H.-I. Wu, D.-Y. Guo, H.-H. Chin, and R.-S. Tsay, "A pipeline-based scheduler for optimizing latency of convolution neural network inference over heterogeneous multicore systems," in *Proc. 2nd IEEE Int. Conf. Artif. Intell. Circuits Syst. (AICAS)*, Aug./Sep. 2020, pp. 46–49.

[24] B. Kim, S. Lee, A. R. Trivedi, and W. J. Song, "Energy-efficient acceleration of deep neural networks on realtime-constrained embedded edge devices," *IEEE Access*, vol. 8, pp. 216259–216270, 2020.

[25] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "DeepX: A software accelerator for low-power deep learning inference on mobile devices," in *Proc. 15th ACM/IEEE Int. Conf. Inf. Process. Sensor Netw. (IPSN)*, Apr. 2016, pp. 1–12.

[26] X. Zhang, C. Hao, P. Zhou, A. Jones, and J. Hu, "H2H: Heterogeneous model to heterogeneous system mapping with computation and communication awareness," in *Proc. 59th ACM/IEEE Design Autom. Conf.*, Jul. 2022, pp. 601–606.

[27] M. Hosseinabady, M. A. B. Zainol, and J. Núñez-Yáñez, "Heterogeneous FPGA+GPU embedded systems: Challenges and opportunities," 2019, *arXiv:1901.06331*.

[28] M. Hall and V. Betz, "HPIPE: Heterogeneous layer-pipelined and sparse-aware CNN inference for FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*. New York, NY, USA: Association for Computing Machinery, Feb. 2020, p. 320.

[29] H.-J. Jeong, H.-J. Lee, C. H. Shin, and S.-M. Moon, "IONN: Incremental offloading of neural network computations from mobile devices to edge servers," in *Proc. ACM Symp. Cloud Comput. (SoCC)*. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 401–411.

[30] Y. Hu, C. Imes, X. Zhao, S. Kundu, P. A. Beerel, S. P. Crago, and J. P. N. Walters, "Pipeline parallelism for inference on heterogeneous edge computing," 2021, *arXiv:2110.14895*.

[31] R. Hadidi, J. Cao, M. Woodward, M. S. Ryoo, and H. Kim, "Distributed perception by collaborative robots," *IEEE Robot. Autom. Lett.*, vol. 3, no. 4, pp. 3709–3716, Oct. 2018.

[32] M. Xu, F. Qian, M. Zhu, F. Huang, S. Pushp, and X. Liu, "DeepWear: Adaptive local offloading for on-wearable deep learning," *IEEE Trans. Mobile Comput.*, vol. 19, no. 2, pp. 314–330, Feb. 2020.

[33] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "MoDNN: Local distributed mobile computing system for deep neural network," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, 2017, pp. 1396–1401.

[34] L. Zhou, H. Wen, R. Teodorescu, and D. H. Du, "Distributing deep neural networks with containerized partitions at the edge," in *Proc. 2nd USENIX Workshop Hot Topics Edge Comput. (HotEdge)*. Renton, WA, USA: USENIX Association, Jul. 2019, pp. 1–7. [Online]. Available: https://www.usenix.org/conference/hotedge19/presentation/zhou

[35] J. Tang, S. Liu, L. Liu, B. Yu, and W. Shi, "LoPECS: A low-power edge computing system for real-time autonomous driving services," *IEEE Access*, vol. 8, pp. 30467–30479, 2020.

[36] The Apache Software Foundation. (2020). *Apache TVM*. [Online]. Available: https://tvm.apache.org/

[37] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. 13th USENIX Conf. Operating Syst. Design Implement. (OSDI)*. Berkeley, CA, USA: USENIX Association, 2018, pp. 579–594.

[38] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez, and I. Stoica, "Ansor: Generating high-performance tensor programs for deep learning," in *Proc. 14th USENIX Conf. Operating Syst. Design Implement.* Berkeley, CA, USA: USENIX Association, 2020, pp. 863–879.

[39] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2012, pp. 1097–1105.

[40] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[41] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*.

[42] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 4510–4520.

[43] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.

[44] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Syst. Tech. J.*, vol. 49, no. 2, pp. 291–307, 1970.

[45] A. Grama, V. Kumar, and A. Sameh, "Parallel hierarchical solvers and preconditioners for boundary element methods," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 337–358, Jan. 1998.

[46] T. Verbelen, T. Stevens, F. De Turck, and B. Dhoedt, "Graph partitioning algorithms for optimizing software deployment in mobile cloud computing," *Future Gener. Comput. Syst.*, vol. 29, no. 2, pp. 451–459, Feb. 2013.

[47] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50× fewer parameters and <0.5 MB model size," 2016, *arXiv:1602.07360*.

[48] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 2818–2826.

[49] L. Zheng, C. Jia, M. Sun, Z. Wu, and C. H. Yu. (Mar. 3, 2021). *Introducing TVM Auto-Scheduler (A.K.A. Ansor)*. [Online]. Available: https://tvm.apache.org/2021/03/03/intro-auto-scheduler

[50] The Apache Software Foundation. (2020). *TVM. User Tutorial. Compiling and Optimizing a Model With TVMC*. [Online]. Available: https://tvm.apache.org/docs/tutorial/tvmc_command_line_driver.html

[51] (2020). *TVM Python API*. [Online]. Available: https://tvm.apache.org/docs/reference/api/python/relay/testing.html

[52] Y. Ding. (2022). *Multi-Stream Execution in Meta VM*. [Online]. Available: https://www.tvmcon.org/wp-content/uploads/2022/01/16-15_40LT-Yaoyao_Ding-Multi-stream_Support_for_Virtual_Machine_Executor_in_Meta.pdf

[53] *NVIDIA Jetson TX2 NX System-on-Module*, NVIDIA Corp., Santa Clara, CA, USA, 2022.

[54] NVIDIA Corporation. *Tegra Linux Driver Package Release Notes*. Accessed: Nov. 30, 2021. [Online]. Available: http://developer2.download.nvidia.com/embedded/L4T/r27_Release_v1.0/Docs/Tegra_Linux_Driver_Package_Release_Notes_R27.1.pdf

[55] S. Holly, A. Wendt, and M. Lechner, "Profiling energy consumption of deep neural networks on NVIDIA Jetson nano," in *Proc. 11th Int. Green Sustain. Comput. Workshops (IGSC)*, Oct. 2020, pp. 1–6.

[56] E. Cai, D.-C. Juan, D. Stamoulis, and D. Marculescu, "NeuralPower: Predict and deploy energy-efficient convolutional neural networks," 2017, *arXiv:1710.05420*.

[57] C. F. Rodrigues, G. Riley, and M. Luján, "Fine-grained energy profiling for deep convolutional neural networks on the Jetson TX1," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2017, pp. 114–115.

[58] B. Jeon, S. Park, P. Liao, S. Xu, T. Chen, and Z. Jia, "Collage: Seamless integration of deep learning backends with automatic placement," 2021, *arXiv:2111.00655*.

[59] M. Li, Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, "The deep learning compiler: A comprehensive survey," 2020, *arXiv:2002.03794*.

[60] D. Velasco-Montero. (2022). *TVM Heterogeneous Co-Execution*. [Online]. Available: https://github.com/DVM000/TVM_deploy.git

DELIA VELASCO-MONTERO received the B.Eng., M.Sc., and Ph.D. degrees (Hons.) in telecommunication from the University of Seville, Spain, in 2014, 2016, and 2023, respectively. She has spent six months with the National Institute of Aerospace Technology (INTA), Madrid, Spain, granted by the Ministry of Defense, Spain. In May 2017, she joined the Institute of Microelectronics of Seville (University of Seville–CSIC) as a Ph.D. student, granted by the Spanish Government (FPU17/02804). She was a Visiting Researcher with Ghent University, Ghent, Belgium, where she worked on neural-network acceleration. Her main research interests include vision systems and the embedded realization of deep neural networks.

BART GOOSSENS (Member, IEEE) received the M.S. degree in computer science and the Ph.D. degree in engineering from Ghent University, in 2006 and 2010, respectively. He is currently a Professor of digital image processing with the Image Processing and Interpretation Research Group, Department of Telecommunications and Information Processing. His research interests include medical image reconstruction (CT and magnetic resonance image), noise modeling and estimation, and medical image quality assessment. He currently serves as an Associate Editor for IEEE TRANSACTIONS ON IMAGE PROCESSING.

JORGE FERNÁNDEZ-BERNI received the B.Eng. degree (Hons.) in electronics and telecommunication, the M.Sc. degree (Hons.) in microelectronics, and the Ph.D. degree (Hons.) from the University of Seville, Seville, Spain, in 2004, 2008, and 2011, respectively. From 2005 to 2006, he was with Telecommunication Industry. He was a Visiting Researcher with the Computer and Automation Research Institute, Budapest, Hungary; Ghent University, Ghent, Belgium; the University of Notre Dame, Notre Dame, IN, USA; and Institute Pascal, Université Clermont Auvergne, Clermont-Ferrand, France. He is currently an Associate Professor with the Department of Electronics and Electromagnetism, University of Seville. He has authored or coauthored about 80 papers in refereed journals, conferences, and workshops, as well as two books and two book chapters. He is also the first inventor of two licensed patents. His current research interests include smart image sensors, vision chips, and embedded systems.

ÁNGEL RODRÍGUEZ-VÁZQUEZ (Life Fellow, IEEE) received the bachelor's and Ph.D. degrees in physics-electronics from Universidad de Seville, in 1976 and 1982, respectively. After research stays with the University of California at Berkeley and Texas A&M University, he became a Full Professor of electronics with the University of Seville, in 1995. He co-founded Instituto de Microelectrónica de Sevilla, jointly undertaken by Consejo Superior de Investigaciones Científicas (CSIC) and the University of Seville. He started a research laboratory on analog and mixed-signal circuits for sensors and communications. In 2001, he was the main promoter and co-founder of the start-up company AnaFocus Ltd., where he served as the CEO, on leave from the university, until June 2009, when the company reached maturity as a worldwide provider of smart CMOS imagers and vision systems-on-chip. AnaFocus was founded on the basis of his early patents. He holds eight patents. While in academia, he has conducted research and development activities on mixed-signal microelectronics for massive sensory data, including vision chips and neuro-fuzzy controllers. He also pioneered the application of chaos to instrumentation and communications. His team designed the first, worldwide, integrated circuits with controllable chaotic behavior and the design and prototyping of the first worldwide chaos-based communication MoDem chips. His team made also significant contributions in the areas of structured analog and mixed-signal design and the data converter design, including the elaboration of advanced teaching materials on this topic for different industrial courses and the production of two widely quoted books on the design of high-performance CMOS sigma–delta converters. Many high-performance mixed-signal chips were successfully designed by him and his co-workers in the framework of different research and development programs and contracts. These included three generation of vision chips for high-speed applications, analog front-ends for XDSL MoDems, ADCs for wireless communications, ADCs for automotive sensors, chaotic signals generators, and complete MoDems for power-line communications. Many of these chips were state-of-the-art in their respective fields. Some of them entered in massive production. He was also active regarding industrial training. He produced teaching materials on data converters that were employed by several companies. His courses got the Quality Label of EuroPractice. His research work has received some 9100 citations. He has an H-index is 48 and an I10-index is 177 according to Google Scholar. He received several national and international awards, including the IEEE Rogelio Segovia Torres Award, in 1981, during his studies. He has received a number of awards for his research: the IEEE Guillemin-Cauer Best Paper Award, two Wiley's IJCTA Best Paper Awards, two IEEE ECCTD Best Paper Awards, one IEEE-ISCAS Best Paper Award, one SPIE-IST Electronic Imaging Best Paper Award, the IEEE ISCAS Best Demo-Paper Award, and the IEEE ICECS Best Demo-Paper Award. He is on the committee of several international journals and conferences. He has chaired several international IEEE (NDES 1996, CNNA 1996, ECCTD 2007, ESSCIRC 2010, and ICECS 2013) and SPIE conferences. He served as a VP Region 8 for the IEEE Circuits and Systems Society, from 2009 to 2012 and the Chair for the IEEE CASS Fellow Evaluation Committee, in 2010, 2012, 2013, 2014, and 2015. He has been the General Chairperson of IEEE ISCAS 2020. He has served as an editor, an associate editor, and the guest editor for IEEE and non-IEEE journals.

WILFRIED PHILIPS (Senior Member, IEEE) was born in Aalst, Belgium, in 1966. He received the Diploma and Ph.D. degrees in electrical engineering from the University of Ghent, Belgium, in 1989 and 1993, respectively. From October 1989 to September 1993, he was a Research Assistant with the Department of Electronics and Information Systems, Ghent University, for the Flemish Fund for Scientific Research (FWO). He is currently a Senior Professor and the Leader of the Image Processing and Interpretation Research Group, TELIN Department, Ghent University. His main research interests include image and video restoration, image analysis, lossless and lossy data compression of images and video, and processing of multimedia data.