A Design Space Exploration Framework for Convolutional Neural Networks Implemented on Edge Devices

Foivos Tsimpourlas, Lazaros Papadopoulos, Anastasios Bartsokas, and Dimitrios Soudris

***Abstract*—Deploying convolutional neural networks (CNNs) in embedded devices that operate at the edges of Internet of Things (IoT) networks provides various advantages in terms of performance, energy efficiency, and security in comparison with the alternative approach of transmitting large volumes of data for processing to the cloud. However, the implementation of CNNs on low power embedded devices is challenging due to the limited computational resources they provide and to the large resource requirements of state-of-the-art CNNs. In this paper, we propose a framework for the efficient deployment of CNNs in low power processor-based architectures used as edge devices in IoT networks. The framework leverages design space exploration (DSE) techniques to identify efficient implementations in terms of execution time and energy consumption. The exploration parameter is the utilization of hardware resources of the edge devices. The proposed framework is evaluated using a set of 6 state-of-the-art CNNs deployed in the Intel/Movidius Myriad2 low power embedded platform. The results show that using the maximum available amount of resources is not always the optimal solution in terms of performance and energy efficiency. Fine-tuned resource management based on DSE, reduces the execution time up to 3.6% and the energy consumption up to 7.7% in comparison with straightforward implementations.**

***Index Terms*—Convolutional neural networks (CNNs), design space exploration (DSE), embedded systems.**

I. Introduction

THE MARKET of Internet of Things (IoT) is expected to continue to grow, as the number of connected devices will reach 20.8 billion by 2020 [1]. Low-power embedded platforms that often operate within power envelopes of less than 1 W act as the edge devices of IoT networks. Among others types, low-power visual sensors enable a wide range of smart IoT applications, from domains such as transportation and surveillance. These applications perform complex tasks, such as image recognition and face detection enabled by convolutional neural networks (CNNs). Due to the increased requirements in terms of hardware resources of state-of-the-art CNNs, they are often deployed in IoT network layers with higher computational capabilities (e.g., Fog/Cloud) [2]. However, the fact that the edge devices operate within a tiny power envelope, imposes limitations in the amount of data that can be wirelessly transferred to the Fog/Cloud layers for processing. Additionally, transferring safety critical data introduces significant security challenges, especially for healthcare wearables and surveillance applications.

Near-sensor processing is a natural direction of the evolution of the IoT networks, since it addresses the above challenges by avoiding expensive and possibly insecure data transmissions [3]–[5]. Eliminating the latency imposed by data transfers within IoT networks is critical for a wide range of IoT applications, such as in augmented reality and robotics, where lags impose speed restrictions on drones and robots that try to use CNN inference to interpret real-time video. Also, shifting the load from centralized workstations to distributed edge devices reduces the energy consumption by minimizing the energy spent on communication, thus making IoT networks scalable. However, near-sensor processing requires the efficient implementation of CNNs on edge devices, which is a significant challenge due to the complexity of state-of-the-art CNNs and the limited computational resources that edge devices provide. To achieve CNN inference execution with low latency along with low energy consumption, efficient fine-tuned implementations are required that effectively exploit the hardware resources of edge devices.

The need for mobile vision solutions with near-sensor processing capabilities of IoT data prompted the development of specialized low power architectures for CNN inference execution. IoT devices that locally process data from visual sensors are enabled by low power embedded platforms such as multicore DSPs, embedded GPUs, and field programmable gate arrays (FPGAs). From the market perspective, the annual growth rate of the machine vision market is exceeding 13% [6] and many commercial solutions are available today that enable applications from domains such as drones, robotics, and autonomous driving [7]–[9].

Current research mainly focuses on the acceleration of CNNs using FPGAs and ASICs. CNNs have been deployed in CPU/DSP and GPU architectures using highly optimized frameworks such as Caffe [10], Tensorflow [11], and

Manuscript received April 3, 2018; revised June 8, 2018; accepted July 2, 2018. Date of publication July 18, 2018; date of current version October 18, 2018. This work was supported by the European Union's Horizon 2020 Research and Innovation Programme through SDK4ED Project under Grant 780572 and through VINEYARD Project under Grant 687628. This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis 2018 and appears as part of the ESWEEK-TCAD special issue. *(Corresponding author: Lazaros Papadopoulos.)*

The authors are with the School of Electrical and Computer Engineering, National Technical University of Athens (Zographou Campus), 15780 Athens, Greece (e-mail: lpapadop@microlab.ntua.gr).

cuDNN [12] by Nvidia. These frameworks support GPU-based platforms, such as Nvidia Tegra X1 [13] and specialized architectures for computer vision applications, such as Intel/Movidius Myriad [7] and Mobileye EyeQ3. From the software perspective, optimizations such as the Winograd convolution algorithm [14] and Strassen [15] have been proposed to further optimize CNN implementations. FPGA approaches for CNN acceleration include, among others, NeuFlow [16] and nn-X [17]. The former has been implemented as an ASIC and integrated in the IBM SOI processor. Other ASIC implementations are the Origami [18] and the ShiDianNao [19].

Although many research groups focus on the design of low power CNN accelerators, as described above, only few works examine techniques and methodologies for the efficient implementation of CNNs on edge devices [20]. This paper proposes a framework for CNN deployment on low power processor-based architectures that have been designed for computer vision applications. The implementation methodology is based on design space exploration (DSE) to optimize the implementation of CNNs on the underlying platform.

DSE is another area related with the present work. DSE is a technique widely used in the embedded systems design. In the context of neural networks, it is mainly used for two purposes: 1) for the optimization of the CNN architectural characteristics (e.g., [21]) and 2) for the optimization of the underlying architecture in which CNNs will be deployed. Indeed, there is a lot of work in the literature that focuses on DSE for optimizing FPGA-based and ASIC accelerators for CNNs [22]–[24]. However, the proposed framework targets processor-based architectures that are used at the edges of IoT networks and by leveraging DSE techniques, it provides efficient CNN deployments. To the best of our knowledge there exists no similar work about DSE for CNN implementation on processor-based systems. The main features of the proposed framework are the following.

1) Automatic deployment of CNNs on low power edge devices. The framework is fully compatible with the Caffe framework.
2) It supports DSE of hardware features, such as the number of processing units in which each layer can be implemented.
3) It provides efficient CNN execution techniques, such as the parallel execution of CNN branches, to further reduce execution time of CNN inference.
4) It can be easily extended with new features and be adapted to various low power embedded architectures.

The main contributions of this paper can be summarized as follows.

1) The development of a framework for the implementation of trained CNNs on edge devices using the Caffe interface.
2) Evaluation of DSE techniques for the optimization of CNNs implemented on edge devices. DSE allows the efficient utilization of hardware resources and the identification of tradeoffs between performance and energy consumption.

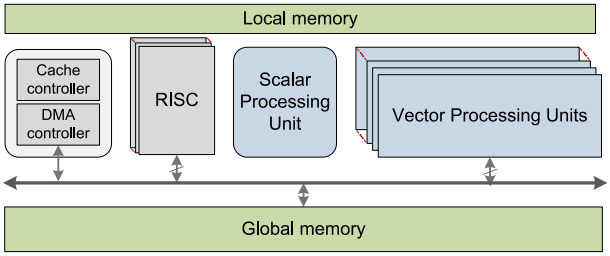


Fig. 1. High-level schematic of target architectures.

3) A methodology for implementing CNNs on edge devices through systematic exploration, supported by the proposed framework.

The rest of this paper is organized as follows. In Section II, we describe the target architectures of the proposed methodology. The framework and the CNN implementation methodology are analyzed in Section III. The evaluation of the proposed methodology and the demonstration of the framework follows. Finally, in Section V we draw the conclusions.

II. Edge Devices and CNN Implementation Challenges

The proposed framework targets low power heterogeneous processor-based architectures designed for computer vision and deep learning applications. These devices are extremely low power (often less than 1W) and are often used at the edges of IoT networks to perform tasks enabled by CNNs, such as object detection and recognition. The family of low power edge devices includes the Intel/Movidius Myriad [7], CEVA XM [8], and Cadence Tensilica vision DSP [9]. Fig. 1 shows a high-level schematic of the architectures which are supported by the deployment framework. The common architectural features are the following.

1) *Multiple Memory Hierarchies:* A local memory, usually an SRAM, provides low latency and high throughput access. Its size is limited to few MB. A global larger memory, such as a DRAM, provides significantly more storage (e.g., hundreds of MB), but it is slower and it is accessed through DMA transactions. A cache memory may be used to reduce the cost of accessing data stored in the global memory. A typical data management pattern is to fetch data from the global memory to the local through DMA transactions and after the data are processed by the processing elements, the result is transferred back to the global memory through DMA.
2) *Multiple Vector Processing Units (VPUs):* They usually support VLIW, SIMD, and multiply accumulate operations (MACs) with high efficiency. CEVA mx-6 and Tensilica vision P6 integrate 128 and 256 MACs, respectively.
3) *RISC processors* often run an operating system. Examples include two LEON CPUs in Myriad2 and ARM Cortex i.MX 6 in YouSiP vision DSP based on CEVA platform. They also handle tasks such as interrupts, IO, etc.

The efficient implementation of CNNs on edge devices with the aforementioned characteristics is challenging, due to the limited resources they provide and the high computational and storage requirements of CNNs. A major implementation issue that significantly affects the CNN inference performance is the utilization of the limited local memory. Also, stalls may frequently occur in ported memories under heavy data sharing. Efficient local memory utilization will reduce the number of DMA transfers which will benefit the energy consumption. Additionally, extensive experimentation with Myriad2 has shown that heavy DMA usage reduces the performance of the DMA engine, with negative impact on application's execution time [25]. Furthermore, the management of the available processing units, such as the efficient partitioning of the algorithm between them and the memory alignment issues should be carefully examined to effectively exploit the provided SIMD features. They can be considered significant development challenges that require meticulous, fine grained implementation tuning. Experimentation and time consuming *adhoc* solutions are often used to address the above challenges. However, a systematic exploration is required that will assist developers to automatically deploy CNNs on such devices.

III. CNN Implementation Framework and Methodology

In this section, we present the implementation methodology of CNNs on edge devices and the proposed framework that supports it.

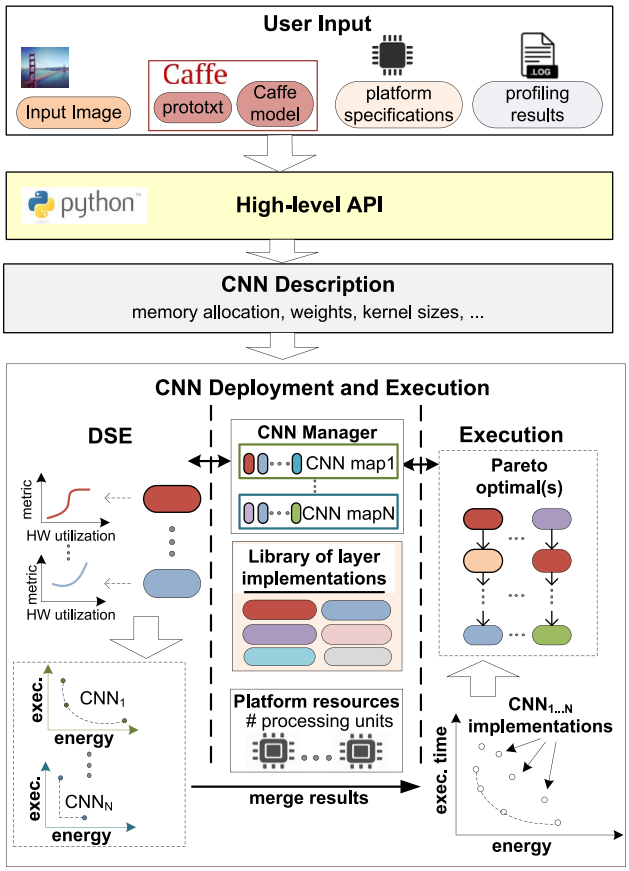


Fig. 2. Overview of the proposed CNN implementation framework.

A. Overview

Fig. 2 shows a high-level view of the CNN implementation framework. The key components of the CNN implementation framework are as follows.

1) The high-level API, written in Python, which is exposed to developers for configuring the operation of the CNN framework.
2) The CNN description, which analyzes the CNN specifications, allocates memory for weights, etc.
3) The deployment and execution layer of the framework contains the CNN manager, which generates the CNN maps that store information about each CNN layer (such as type of layer, where the data associated with this layer are allocated, which layer implementation from the library will be used, e.g., convolution/fully connected, which is the subsequent layer, etc.). Depending on the functionality that has been selected (DSE, execution or analysis), the appropriate deployment(s) and execution(s) are instructed.
4) The output is either the execution time and the energy consumption for a single execution or DSE results with tradeoffs between execution time, energy consumption, and accuracy, as shown in Fig. 2. This depends on the operation that has been selected by developers.

Details about each one of the components are provided in the following sections.

B. High-Level API and Framework Functionality

The high-level API of the proposed framework is fully compatible with the Caffe API and it is developed in Python. It serves three purposes: 1) to allow users to select between different framework operation modes; 2) to provide configuration parameters for the CNN deployments on edge devices (e.g., configure resource utilization); and 3) to generate a set of source files that provide a platform-specific CNN description. It parses *prototxt* and *caffemodel* files, processes information about edge platform specifications, such as the memory hierarchy and the number of available VPUs and manages profiling results obtained by DSE.

The framework can operate in three different modes, which are depicted in Fig. 3 and described below.

1) *Single CNN Inference Execution:* By providing CNN architecture details (i.e., a *prototxt* and a *caffemodel* file), along with an image or a batch of images, a single CNN inference is executed on the edge device. Platform specifications, such as the number of VPUs, which each layer of the CNN will utilize, may also be provided for implementation tuning. The provided output is execution time and energy consumption results of the CNN inference execution.
2) *DSE Mode:* The input required to perform DSE is the CNN architecture details, an image or a batch of images and the hardware specifications which will be

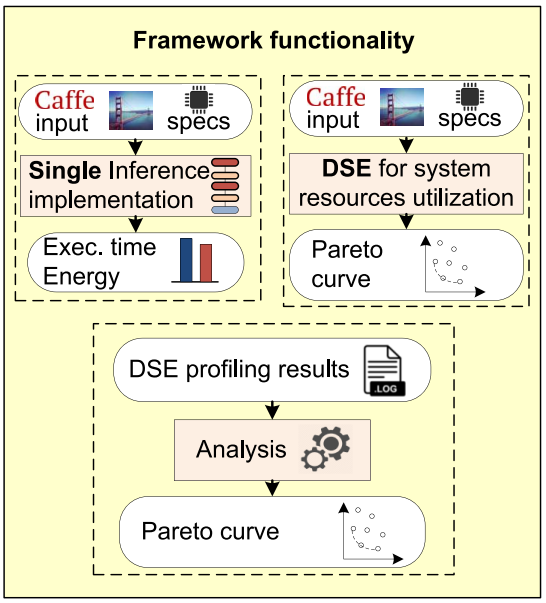


Fig. 3. Functionality provided by the CNN deployment framework.

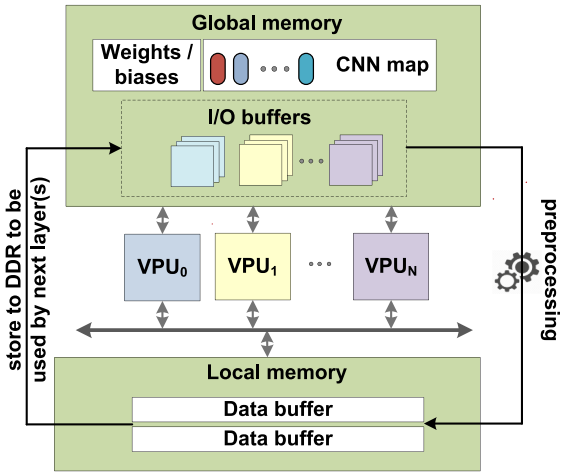


Fig. 4. CNN deployment and execution model for edge devices.

explored, such as the number of VPUs that each specific CNN layer will utilize. The output is a set of raw profiling results for execution time and energy consumption for each single CNN layer deployed by utilizing a different amount of hardware resources. The API can further process this output and provide a set of Pareto optimal implementations of execution time versus energy consumption.

3) *DSE Profiling Results Analysis:* Finally, in case the raw profiling results have been produced by DSE earlier, they can be fed to the high-level API to generate an execution time versus energy consumption Pareto curve. By using this option, no CNN inference is executed on the underlying hardware platform. It performs processing of raw profiling results only.

C. CNN Description and CNN Deployment and Execution Layers

The high-level API generates a set of C/C++ source files that provide a low-level CNN description with information manageable by the underlying platform. More specifically, the source files contain the following.

1) Line-by-line suitable instructions for constructing and storing in a vector data structure each layer object that needs to be executed, including the layer parameters and hyper-parameters of the CNN.
2) Raw floating point weights and bias vectors of the network, as well as the batch of images and all the necessary output buffers that will be used for implementing and executing the CNN inference.
3) Details about the CNN architecture and information about the types of layers used in order to minimize code size in compile time.
4) The memory storage requirements of the CNN.

The above information is forwarded to the CNN deployment and execution layer to be processed by the CNN manager. The latter controls the whole deployment and execution, depending on the operation mode selected.

Fig. 4 shows a high-level execution model of the proposed framework. With respect to data allocation, the CNN map and all CNN data are allocated in the global memory of the edge device, which is usually hundreds of MBs. The local memory (few MBs) is used only for processing of data fetched from DDR. Two buffers are allocated, to overlap communication with computation, implementing the widely used double buffering technique.

A major implementation goal for all CNN layers was to efficiently exploit the limited available memory of edge devices and to reduce as much as possible the required DMA transactions, especially when handling relatively large output maps. The goal of preprocessing is to transform data in way to be efficiently handled by VPUs, reduce the required memory size and the number of DMA data transfers. The following two data management techniques are applied in the context of preprocessing.

1) Transferring padding elements from global to local memory introduces a significant overhead. However, in our implementation model, the elements that are expected to be used for padding are computed in advance by the corresponding VPU. Therefore, only the useful data are transferred in the local memory, while padding elements are added in the local memory. Thus, the amount of data required to be transferred by DMA is significantly reduced. The local calculation of padding by each VPU is trivial and it does not impose any significant overhead.
2) To effectively exploit the SIMD features of the VPUs, data are aligned in memory and dummy elements are added in vectors where required. Thus, although a small memory size overhead is added, performance is significantly improved.

As an example to highlight the significance of data preprocessing, the input of the third layer of the GoogleNet is $64*56*56$

with padding equal to 1 and 192 output maps. The amount of useful data is $64*56*56*2 = 401\,408$ Bytes, while the amount of padding elements is $(56+56+55+55)*2*64 = 28\,416$ Bytes. Thus, by skipping the transferring of padding elements from global to local memory and reproducing them locally, the amount of data transferred is reduced by about 5.4 MB for the 192 output maps.

After the preprocessing, the data are allocated in data buffers in the local memory, to be processed by the VPUs. Typical embedded systems data management techniques, such as double buffering are applied to enable overlapping of computation and communication and improve performance. After the data are processed, they are fetched back to the global memory through DMA transactions.

The CNN layers are executed one after the other. All information about each layer (pointers to their I/O buffers, weights, biases, as well as parameters like kernel size, striding) is precomputed offline and it is placed in the CNN description source files in a map data structure, in which each object is a single CNN layer. This process (which is actually an initialization phase) lasts a few seconds and needs to be done just once for each CNN implementation. The map is allocated in the global memory (CNN map in Fig. 4). During the inference execution the CNN manager retrieves each layer object and instructs its execution. As soon as the execution of the current layer is completed and the data are transfered back to the global memory, the CNN manager initiates the execution of the subsequent layer.

The library of CNN layer implementations includes the *Convolution*, *Pooling*, *Fully Connected*, *LRN*, *Concat*, *Split*, and *Dropout* layers. All layers are implemented in C for improving the compatibility of the framework, however, platform-specific implementations (e.g., in highly optimized assembly) usually provide major performance improvements in the execution of CNN inference, especially with respect to the convolution operation.

The Concat layer is a utility layer that concatenates multiple input blobs to a single output blob and it is used by CNNs such as the GoogleNet and Squeezenet. This layer is efficiently implemented in the library of the proposed framework, in order to avoid unnecessary memory accesses and data transfers. The input layers of each Concat layer are offline identified and they allocate their output directly to the Concat buffer, instead of their own buffers, as shown in Fig. 5. Thus, there is no need to transfer data from input layers to the Concat buffer and the Concat layer execution time overhead is eliminated. Finally, the Split layer is a hidden layer produced by the Caffe computational engine, which indicates that a certain layer feeds multiple layers, by initiating a block of layers in parallel branches. The Split layer is used to identify parallel blocks and to perform the required computations to calculate the number of buffers and the memory size of each one.

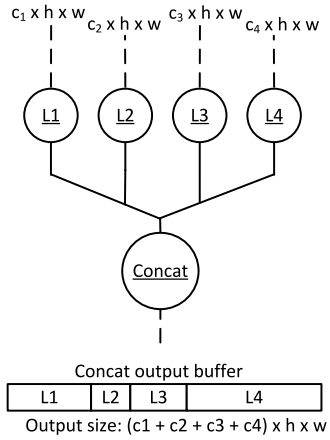


Fig. 5. Concat layer. c, h, and w are the channels, the height, and the width, respectively.

D. Design Space Exploration

The DSE operation mode is initiated by the high-level API and managed by the CNN manager. The CNN manager executes each single CNN layer by utilizing different amount of hardware resources in each execution. Execution time and energy consumption results versus hardware resources utilization are collected for each CNN layer. Thus, the output is a set of different CNN implementations, where each implementation utilizes different amount of resources and therefore has different characteristics in terms of execution time and energy efficiency. The results are collected in a raw csv file. The framework processes the profiling results and exports Pareto plots of performance versus energy consumption, where each Pareto point is a different CNN implementation. Developers may select the one that better fits the CNN deployment requirements.

E. Portability and Extensibility Issues

The framework has been designed to be applicable to edge devices with architectural features similar to ones described in Section II. Therefore, the high-level API and the CNN description are platform independent. With respect to the CNN deployment and execution layer, the library of CNN layer implementations can be considered platform-independent, since it provides C implementations for all layers. However, in practice highly optimize platform-specific implementations of each layer are preferable, since they provide improved results in terms of both execution time and energy consumption.

Although the library contains a wide variety of layers, so that several state-of-the-art CNNs can be implemented, new layers can be seamlessly added by developers, in three steps.

1) Add the new layer in the library of CNN layer implementations in C or assembly.
2) Add required information about the new layer in the CNN manager.
3) Provide the appropriate high-level API functions for the newly added layer.

The high-level API provides a set of functions that enable the exploration of an arbitrary number of hardware specifications. The framework does not provide any limitation in the type of hardware specification that can be explored, since the platform-specific details of the implementation of each

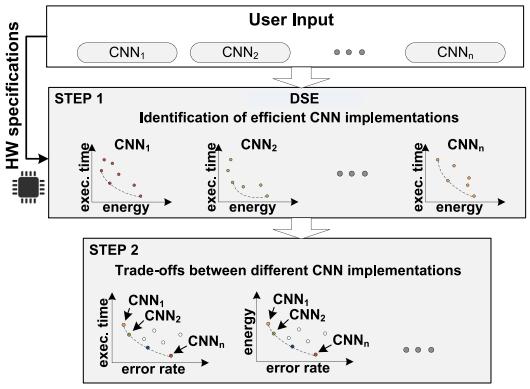


Fig. 6. Proposed CNN implementation methodology.

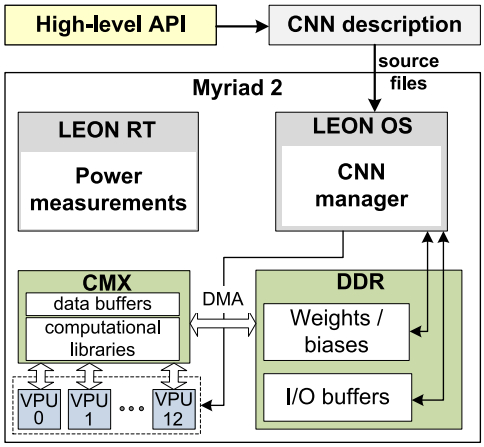


Fig. 7. Framework instantiation in Intel/Movidius Myriad2.

layer are described in the library of CNN layers implementation. Although in this paper we focus on the exploration of the number of VPUs in which each layer is implemented, other options such as the type of convolution (direct, im2col, Winograd) can also be explored by the proposed framework. It is important to state that the algorithms used by the high-level API exploit the native Caffe C++ libraries exposed to Python packages. The fact that the auto-generated libraries are in C/C++ improves the portability of the framework, as they are usually fully supported by any modern device and therefore minimizes the requirements for modifications. Also, the C++ framework is structured upon an object-oriented principle that can be applied in any general-purpose or specialized system. Although the CNN implementation framework is proposed for low power edge devices, in which the deployment of CNNs is challenging due to their limited resources, it is applicable in any processor-based system, as well.

F. CNN Implementation Methodology

Fig. 6. shows how the framework can be actually used by application developers for the deployment of CNNs in edge devices. The user input is the specifications of various CNN architectures. The implementation of each CNN is optimized on the edge device through DSE. Pareto curves for each CNN implementation are generated. Finally, the results are merged to identify tradeoffs between execution time, energy consumption, and error rate for different implementations. It is important to clarify that in step 1, each Pareto point corresponds to a different implementation of the same CNN architecture, while in step 2, each Pareto point corresponds to a different CNN architecture. This implementation methodology is supported by the framework presented earlier.

The input of the methodology is a set of CNN architectures in a Caffe-compatible description, which are trained for the same purpose, so that they can be interchangeable. However, the execution time, the energy consumption, the memory size requirements, and the accuracy provided by each CNN architecture differs. The proposed framework provides a wide variety of layers and supports various CNNs.

The implementation of each one of the provided CNN architectures is fine-tuned on the edge device, through DSE. By evaluating various implementations of each layer, a set of optimized CNN implementations are identified for each CNN architecture. Thus, the output of the first step of the methodology is a Pareto curve of each CNN architecture, of execution time versus energy consumption. Each Pareto point is a different implementation of the specific CNN architecture on the edge device.

Then, the output of the first step is processed and in the second step 3 Pareto plots are provided that show tradeoffs between execution time, energy efficiency, and accuracy. In contrast with the previous step, each Pareto point is a specific implementation of a different CNN architecture. The Pareto optimal implementations of each CNN architecture that are obtained in the first step are used to generate the corresponding Pareto plots in the second step. In other words, the first step the methodology identifies tradeoffs for each CNN architecture. In the second step, the results are merged, to identify tradeoffs between fine-tuned implementations of different CNN architectures.

IV. Evaluation

In this section, we provide the evaluation of the proposed framework and the CNN deployment methodology applied to the Intel/Movidius Myriad2 embedded device, using a set of state-of-the-art CNNs.

A. Framework Instantiation in Myriad and Evaluation Setup

Myriad 2 is a low power (about 1 W) SoC designed by Intel/Movidius for computer vision and deep learning applications. It integrates 12 VPUs that operate at 600 MHz, 2 LEON4 processors and various hardware accelerators. With respect to the memory hierarchy, Myriad provides a 2 MB local multiported SRAM memory, named connection matrix (CMX) that acts as a scratchpad and a 512 MB DDR2 global memory.

Fig. 7 is a high-level view of the instantiation of the framework on Myriad. LEON OS and LEON RT are two RISC CPUs integrated in Myriad chip. Therefore, CNN manager runs on LEON OS, while power measurements are handled by LEON RT.

More specifically, LEON OS executes the CNN description source files, allocates weights and biases in the DDR and configures the CNN deployment details. Then, it initiates the execution of the CNN manager, which manages the execution of each CNN layer: the required segments of weights, biases, and output maps are retrieved from the DDR and allocated in the data buffers of the CMX. Then, the VPUs operate on the data, by using the computational libraries that reside in the CMX as well and are implemented in assembly. Finally, the data are transfered back to the DDR I/O buffers and the execution of the subsequent CNN layer follows. The above process is repeated, until the whole CNN inference is executed.

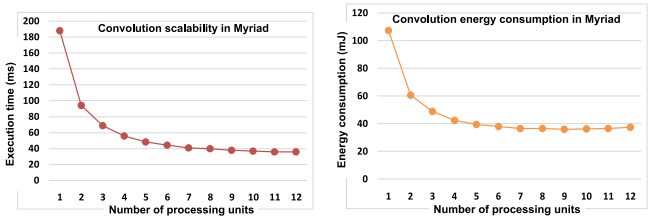
The framework is evaluated using six CNNs of various complexity: 1) AlexNet; 2) GoogleNet; 3) NiN-imagenet; 4) SqueezeNet; 5) VGG; and 6) ZFnet. The CNNs have been selected based on the following two criteria.

1) To require various amount of computational resources and to provide significantly different execution time, energy consumption and accuracy results.
2) To be considered state-of-the-art and be widely used for image recognition tasks.

Table I shows the specifications of the above CNNs. All CNNs use the same input and are trained using ImageNet dataset for the same number of output classes, therefore, they can be used interchangeably. The number of layers significantly ranges, from 13 (AlexNet and ZFnet) to 83 (GoogleNet). The same applies to the memory requirements, which refers the size that the weights and biases occupy in the global memory. All CNNs can be implemented in Myriad, since their data fit in the global memory, which is 512MB. Another important metric is the top-5 error rate of each CNN, that ranges from 19.7 (SqueezeNet) to 8 (VGG). The proposed methodology will be used: 1) to provide fine-tuned implementations of the above CNNs in Intel/Movidius device and 2) to demonstrate tradeoffs between the execution time, energy consumption, and between the fine-tuned CNN implementations.

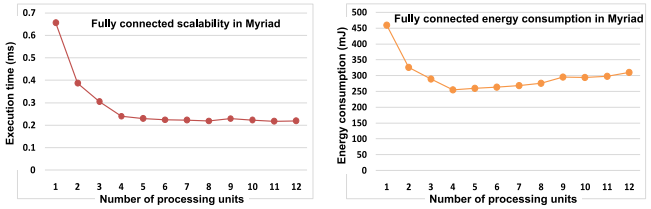
B. Experimental Results

Execution time was measured using recommended functions provided by the Myriad development kit. Energy consumption has been accurately calculated by using on-chip sensors that measure the current that flows through various power rails, provided by the Myriad 2 evaluation board. Energy consumption measurement process is managed by the LEON RT, as shown in Fig. 7. All experimental results refer to CNN inference execution time and operations are performed under the IEEE 754 fp16 standard. The hardware specification which is been explored in the context of this paper is the number of Myriad VPUs in which each CNN layer is implemented. Four hundred and eighty two configurations have been examined for AlexNet, about 11M for GoogleNet, 351 for NiN-imagenet, 921K for SqueezeNet, 56 105 for VGG, and 315 for ZFnet. The exploration time lasts up to 6 min for the relatively small CNNs (e.g., AlexNet and SqueezeNet), up to 15 min for the large ones (VGG, GoogleNet). In case of more degrees of freedom (i.e., exploration for other parameters apart from the number of VPUs) it may increase significantly. Therefore, if the exploration time becomes intolerable, heuristics may be used to prune inefficient configuration and decrease the exploration time.



(a) (b)

Fig. 8. Scalability of convolution in Myriad. (a) Convolution exec. time versus number of processing units. (b) Convolution energy consumption versus number of processing units.



(a) (b)

Fig. 9. Scalability of fully connected in Myriad. (a) Fully connected exec. time versus number of processing units. (b) Fully connected energy consumption versus number of processing units.

Before applying the CNN deployment methodology in Myriad, we examine the scalability of the convolution and the fully connected layers. Fig. 8 shows the execution time and the energy consumption of a 3×3 convolution, as the number of processing units increases gradually. Although the execution time continues to drop up to 12 cores, the energy consumption decreases up to 11 cores, while it slightly increases when using 12 cores. With respect to the scalability of the fully connected layer (Fig. 9), the execution time drops up to 4 VPUs. By utilizing more than 4 VPUs, the energy increases, without improvement in the execution time.

There are two tasks which are performed during the execution of each layer: 1) the communication, which is the DMA transaction between the global and the local memory and 2) the computation, which is the actual data processing. It is important to state that when multiple VPUs request DMA transactions concurrently, the DMA engine becomes a bottleneck and the communication overhead increases. Therefore, although the computational overhead tends to be reduced when using utilizing more processing units, due to the exploitation of parallelism, the communication overhead increases.

The execution time of convolutions is usually optimal on 11 or 12 VPUs. The reason, is the fact that since the convolution is computationally intensive, the execution time is dominated by the exploitation of parallelism and the communication overhead is only a small amount of the total execution time, even though a large number of VPUs is employed. On the other hand, in the fully connected, the computation part is trivial and the execution time is dominated by the communication

TABLE I
DETAILS OF CNNS USED FOR EVALUATION

CNN	input image	output vector	#layers	memory(MB)	Error rate
AlexNet	$[277 \times 277 \times 3]$	$[1 \times 1000]$	13	117	17
GoogleNet	$[277 \times 277 \times 3]$	$[1 \times 1000]$	83	16.6	7
NiN-imagenet	$[277 \times 277 \times 3]$	$[1 \times 1000]$	16	15.5	17.5
SqueezeNet	$[277 \times 277 \times 3]$	$[1 \times 1000]$	38	4.68	19.7
VGG	$[277 \times 277 \times 3]$	$[1 \times 1000]$	16	276	8
ZFnet	$[277 \times 277 \times 3]$	$[1 \times 1000]$	13	121	16.5

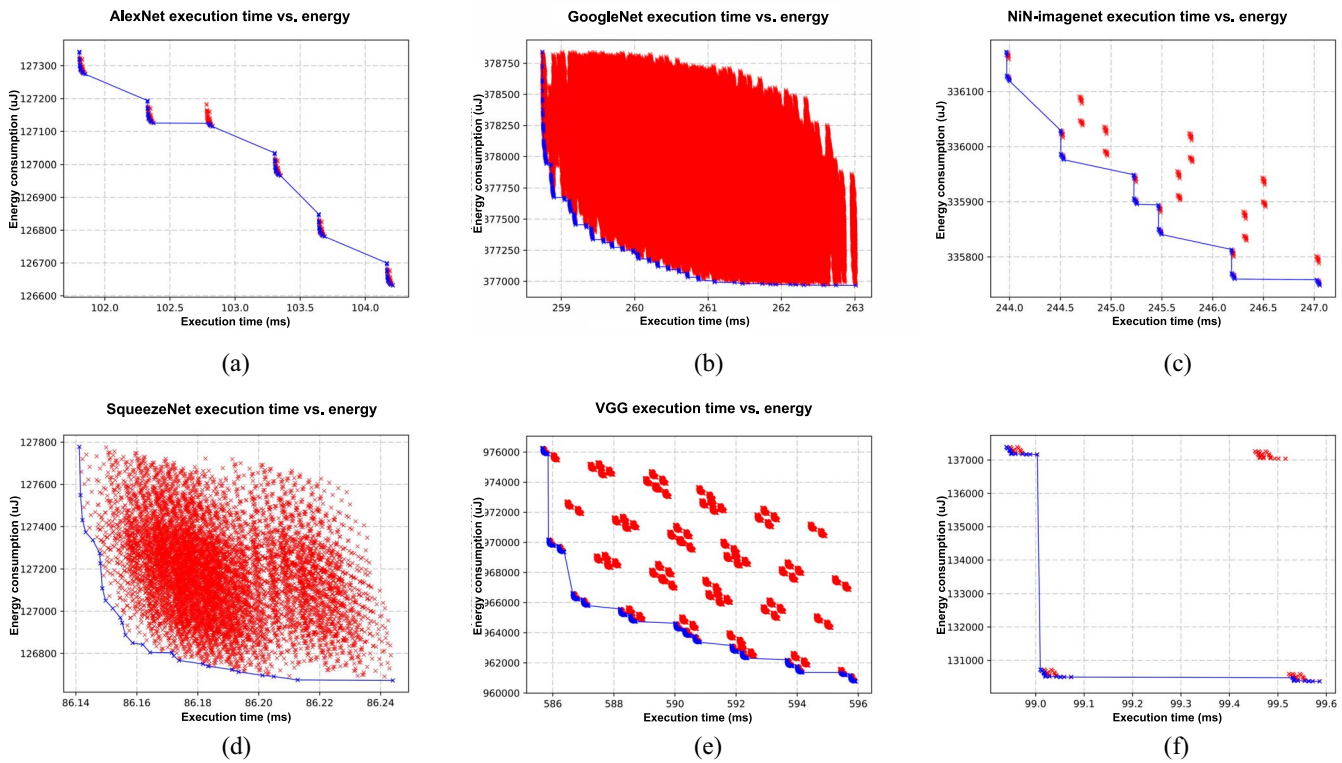


Fig. 10. Output of the first step of the methodology: fine tuning of CNN implementations on an edge device. (a) AlexNet: execution time versus energy. (b) GoogleNet: execution time versus energy. (c) NiN-imagenet: execution time versus energy. (d) SqueezeNet: execution time versus energy. (e) VGG: execution time versus energy. (f) ZFnet: execution time versus energy.

overhead. Therefore, utilizing more than 3–4 VPUs it only adds communication overhead, with trivial benefits from the exploitation of parallelism.

The above experiments highlight the need for exploration in order to identify the most efficient utilization of hardware resources for each layer. It is not possible for developers to be aware of the exact amount of VPUs in which each layer should be implemented in order to minimize execution time and energy consumption, without applying DSE techniques.

The proposed CNN deployment methodology has been applied to the implementation of the six CNNs of Table I on Intel/Movidius Myriad2. The output of the first step of the methodology is shown in Fig. 10. The Pareto plots for each CNN architecture for execution time versus energy consumption are automatically provided by the framework that supports the methodology. Each point in each plot is a CNN implementation that utilizes a different amount of hardware resources. In other words, a CNN implementation differs from another one in the fact that at least one CNN layer utilizes a different number of VPUs.

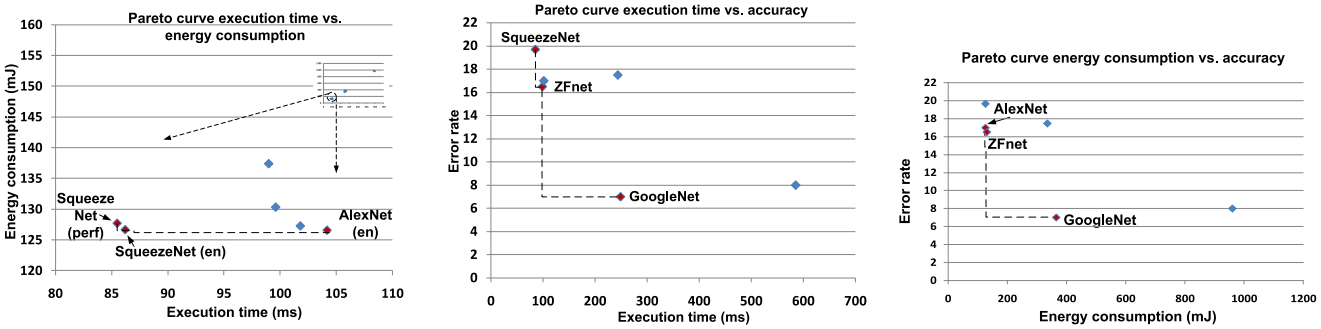
Tradeoffs between execution time and energy consumption are identified for all CNN architectures. Detailed examination of results shows that the most computationally intensive layers are the convolution layers, as expected. Indeed, experiments shows that 68% up to 99% of the execution time of the CNNs of Table I is spent in convolution. Since convolution is a compute-bound operation, the corresponding layers tend to utilize 11 or 12 VPUs. However, as shown earlier in Fig. 8(b) that is not always the most energy efficient solution.

An interesting observation is the fact that in AlexNet, Nin-imagenet, VGG, and ZFnet, the implementations are clustered in groups. The results show that in the implementations that belong to the same cluster, the convolution layers have the exact same configuration (i.e., the corresponding convolution layers use the same number of VPUs). However, they differ in the number of VPUs used by the rest of the layers, such as the pooling and the fully connected, which have a relatively small impact in the execution time.

On the other hand, for the implementations that belong to different clusters, the convolution layers have

TABLE II
CUMULATIVE OUTPUT RESULTS OF THE FIRST STEP OF THE METHODOLOGY

	AlexNet	GoogleNet	NiN-imagenet	SqueezeNet	VGG	ZFnet
Exec. time (ms)	101.8	249.1	244	85.5	586	99
% Exec. time gain	2.3	5.3	1.2	0.81	1.68	0.6
Energy consumption (mJ)	126.6	365.2	335.7	126.7	961	130.3
% Energy gain	0.5	3.6	0.12	0.78	1.53	5.16



(a) (b) (c)

Fig. 11. Output of the methodology: tradeoffs between execution time, energy consumption, and accuracy between various CNNs. (a) Execution time versus energy consumption. (b) Execution time versus accuracy. (c) Energy consumption versus accuracy.

TABLE III
COMPARISON BETWEEN STRAIGHTFORWARD AND FINE-TUNED CNN IMPLEMENTATIONS

	AlexNet	GoogleNet	NiN-imagenet	SqueezeNet	VGG	ZFnet
Exec. time 12 VPUs (ms)	102	258.6	244	88.4	587.2	99.1
Exec. time fine-tuned (ms)	101.8	249.1	244	85.5	586	99
% Exec. time gain	0.29	3.67	0.0015	3.21	0.26	0.17
Energy 12 VPUs (mJ)	139	380.5	337	131.9	1004	141.2
Energy fine-tuned (mJ)	126.6	365.2	335.7	126.7	961	130.3
% Energy gain	8.95	4	0.39	3.98	4.3	7.72

different configurations. For example, AlexNet has five convolution layers. All the implementations of the bottom-right cluster (the most energy efficient) in the AlexNet Pareto curve in Fig. 10(a) utilize 11 VPUs for the first layer and 12 for the rest ones. The implementations within this cluster differ in the number of VPUs utilized by a pooling layer. However, the implementations in the cluster above this one utilize 11 VPUs for the third convolution, as well. Since the implementation of convolution has major impact both in the execution time and in the energy consumption, even a change in the implementation of a single convolutional layer only, affects the execution time and the energy consumption much more than the pooling and fully connected layers.

Table II shows the maximum gains in execution time and energy consumption between the most high performance and the most energy efficient implementations in each CNN, which reach 5.3%.

The results of the second step of the methodology are presented in Fig. 11. The DSE results of Fig. 10 are merged and the accuracy parameter (i.e., error rate from Table I) is also considered. Therefore, in Fig. 11 we present tradeoffs, not between the implementations of a single CNN architecture as we did in Fig. 10, but between different CNN architectures. The implementations have been previously optimized though DSE (and the results of DSE were presented in Fig. 10). The Pareto plots that demonstrate tradeoffs between execution time, energy consumption, and accuracy are presented in Fig. 11. Fig. 11(a) shows the Pareto curve for execution time versus energy. The most efficient implementation in terms of execution time is the SqueezeNet, implemented for high performance (as obtained by the first step of the methodology). The rest of the Pareto points are the SqueezeNet and AlexNet, both tuned for energy efficiency. For instance, GoogleNet provides 258.6 ms execution time and 7% error rate. However, by using SqueezeNet, error rate increases significantly, but execution time decreases by 65%. Similarly, switching from GoogleNet to AlexNet developers can trade accuracy (error rate increases from 7% to 17%) for energy consumption, which decreases by 63%.

To further examine the benefits of DSE methodologies in comparison with straightforward solutions, Table III shows the execution time and energy consumption results of implementing all CNN layers in 12 VPUs. We consider the implementation on 12 VPUs as the baseline for comparison, since it is the one in which all available resources are fully utilized. The results show that by fine-tuning resource utilization through DSE, the execution time slightly improves in comparison with the straightforward approach. However, the energy

consumption decreases significantly, up to 9%. This highlights the importance of fine-tuning CNN implementations, as performed by the first step of the methodology, before identifying tradeoffs between execution time, energy, and accuracy.

V. Conclusion

This paper provides a systematic way of implementing computationally intensive CNNs on low power edge devices. It describes a CNN deployment methodology, which is supported by a framework. The methodology is based on DSE. It enables the fine tuning of the implementation of each CNN layer, along with the identification of tradeoffs between execution time, energy efficiency, and accuracy. The framework is demonstrated by using six CNN architectures implemented on a low power edge device.

Acknowledgment

The authors would like to thank Intel/Movidius for the fruitful discussions and for providing the Myriad Evaluation Board.

References

[1] (2015). *6.4 Billion Connected Things Will be in Use in 2016*. [Online]. Available: https://www.gartner.com/newsroom/id/3165317

[2] S. Yi, C. Li, and Q. Li, "A survey of fog computing: Concepts, applications and issues," in *Proc. Workshop Mobile Big Data*, 2015, pp. 37–42.

[3] F. Porikli *et al.*, "Video surveillance: Past, present, and now the future," *IEEE Signal Process. Mag.*, vol. 30, no. 3, pp. 190–198, May 2013.

[4] C. Shi *et al.*, "A 1000 fps vision chip based on a dynamically reconfigurable hybrid architecture comprising a PE array processor and self-organizing map neural network," *IEEE J. Solid-State Circuits*, vol. 49, no. 9, pp. 2067–2082, Sep. 2014.

[5] C. Labovitz, S. Iekel-Johnson, D. McPherson, J. Oberheide, and F. Jahanian, "Internet inter-domain traffic," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, pp. 75–86, 2010.

[6] (2014). *Machine Vision Market Worth $9.50 Billion by 2020*. [Online]. Available: http://www.marketsandmarkets.com/PressReleases/machine-vision-systems.asp

[7] D. Moloney, "Embedded deep neural networks: 'The cost of everything and the value of nothing,"' in *Proc. Hot Chips 28 Symp. (HCS)*, 2016, pp. 1–20.

[8] Y. Siegel, "The path to embedded vision & AI using a low power vision DSP," in *Proc. Hot Chips 28 Symp. (HCS)*, 2016, pp. 1–28.

[9] G. Efland, S. Parkh, H. Sanghavi, and A. Farooqui, "High performance DSP for vision, imaging and neural networks," in *Proc. Hot Chips 28 Symp. (HCS)*, 2016, pp. 1–30.

[10] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. 22nd ACM Int. Conf. Multimedia*, 2014, pp. 675–678.

[11] M. Abad *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proc. 12th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2016, pp. 265–283.

[12] S. Chetlur *et al.*, "cuDNN: Efficient primitives for deep learning," *Comput. Res. Repository*, vol. abs/1410.0759, 2014.

[13] "Nvidia tegra x1," Santa Clara, CA, USA, Nvidia Inc., White Paper, 2015.

[14] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proc. CVPR*, 2016, pp. 4013–4021.

[15] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *Proc. ICANN*, 2014, pp. 281–290.

[16] P.-H. Pham *et al.*, "NeuFlow: Dataflow vision processing system-on-a-chip," in *Proc. MWSCAS*, 2012, pp. 1044–1047.

[17] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 G-ops/s mobile coprocessor for deep neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. Workshops*, 2014, pp. 696–701.

[18] L. Cavigelli and L. Benini, "Origami: A 803-GOp/s/W convolutional network accelerator," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 27, no. 11, pp. 2461–2475, Nov. 2017.

[19] Z. Du *et al.*, "ShiDianNao: Shifting vision processing closer to the sensor," in *Proc. ACM/IEEE 42nd Annu. Int. Symp. Comput. Archit. (ISCA)*, 2015, pp. 92–104.

[20] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2016, pp. 1–8.

[21] F. N. Iandola *et al.*, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," *Comput. Res. Repository*, vol. abs/1602.07360, 2016.

[22] C. Zhang *et al.*, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2015, pp. 161–170.

[23] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi, "Design space exploration of FPGA-based deep convolutional neural networks," in *Proc. 21st Asia South Pac. Design Autom. Conf. (ASP-DAC)*, 2016, pp. 575–580.

[24] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *Proc. IEEE 31st Int. Conf. Comput. Design (ICCD)*, 2013, pp. 13–19.

[25] L. Papadopoulos, D. Soudris, I. Walulya, and P. Tsigas, "Customization methodology for implementation of streaming aggregation in embedded systems," *J. Syst. Archit.*, vol. 66, pp. 48–60, May 2016.

Foivos Tsimpourlas received the graduation degree in computer engineering from the National Technical University of Athens, Athens, Greece. He is currently pursuing the Ph.D. degree with the School of Informatics, University of Edinburgh, Edinburgh, U.K.

His current interests include combining machine learning and neural networks with contemporary software design techniques.

Lazaros Papadopoulos received the Diploma degree in electrical and computer engineering from the Democritus University of Xanthi, Komotini, Greece, and the Ph.D. degree from the School of Electrical and Computer Engineering, National Technical University of Athens, Athens, Greece.

He is a Research Associate with the Microprocessors and Digital Systems Laboratory, National Technical University of Athens. His current research interests include memory management optimization techniques for embedded systems and power aware computing.

Anastasios Bartsokas received the Diploma degree in electrical and computer engineering from the National Technical University of Athens, Athens, Greece, in 2018.

He is an Embedded Software Engineer with Intel, Dublin, Ireland. His current research interests include convolutional neural networks and low power embedded computing.

Dimitrios Soudris received the Diploma and Ph.D. degrees in electrical and computer engineering from the University of Patras, Patras, Greece, in 1987 and 1992, respectively.

From 1995, he has served as a Professor with the Department of Electrical and Computer Engineering, Democritus University of Thrace, Komotini, Greece, for 13 years. He is currently an Associate Professor with the School of Electrical and Computer Engineering, National Technical University of Athens, Athens, Greece. He has published over 350 papers. He has co-authored/co-edited six Kluwer/Springer books. He is the Leader and the Principal Investigator in numerous research projects funded from the Greek Government and Industry, European Commission, and European Space Agency. His current research interests include embedded systems design, low power very large scale integration design (VLSI), and reconfigurable architectures.

Dr. Soudris is a member of the VLSI Systems and Applications Technical Committee of IEEE CAS and ACM.