Vertical Optimizations of Convolutional Neural Networks for Embedded Systems

Politecnico di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Computer Engineering (34^{th}cycle)

Vertical Optimizations of Convolutional Neural Networks for Embedded Systems

By

Antonio Cipolletta

******

Supervisor(s):
Prof. Enrico Macii
Prof. Andrea Calimera

Doctoral Examination Committee:
Prof. Ibrahim Elfadel, Referee, Khalifa University
Prof. Fatih Ugurdag, Referee, Ozyegin University
Prof. Andrea Acquaviva, Università di Bologna
Prof. Alberto Bosio, École Centrale de Lyon
Prof. Andrea Bottino, Politecnico di Torino

Politecnico di Torino
2022

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Antonio Cipolletta
2022

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

Acknowledgements

First, I would like to thank my supervisor Enrico Macii for letting me pursue a doctorate in his research group. Special thanks go to my supervisor Andrea Calimera for the endless sessions of discussions of research ideas and for helping me drive my efforts and enthusiasm in the right direction.

I would also like to thank all the people at LAB4 with whom I have interacted during this period. Above all, I thank my colleagues Luca and Matteo for sharing their passion for the research topic and for helping me go through all the ups and downs of these last few years. Next, I thank all the students of the Synthesis and Optimization of Digital Systems course, especially Giorgio and Nicole, who ended up doing a master thesis under my supervision: you taught me how to be a better communicator and mentor.

Finally, I would like to thank my family and my friends for their support during these years. To my friends, I still believe that friendship is not important, but it is the only thing that matters. To my parents, Maria Letizia and Martino, I cannot be grateful enough for the environment you created to let me grow up. To my grandparents and my uncle Luca, you have been my role models. To my brother Enzo, you can always count on me. To Elisa, *quando sei qui con me, questa stanza non ha più pareti...*

Abstract

The need for high levels of user privacy, low latency, and low cost has recently required moving the Deep Convolutional Neural Network (CNN) inference process from the cloud to "the edge", that is, on lightweight resource-constrained embedded systems. Such a paradigm shift creates a huge technical challenge: filling the gap between the computational and memory requirements of modern CNNs and the limited hardware and energy resources of embedded systems.

This dissertation tackles this challenge through *vertical and automated optimizations* across the entire software stack, focusing on how to build *small, fast, and energy-efficient* CNNs. After reviewing the most adopted memory allocation algorithms in CNN compilers, the first part of this thesis introduces the dataflow restructuring, a novel functionality-preserving method for minimizing the activation memory footprint of CNNs. Then, a new compression pipeline is presented, which combines weight pruning with dataflow restructuring to deploy more accurate CNNs on tiny MCU devices. The second part of this dissertation presents an optimization framework based on neural architectural design, quantization, and optimized integer kernels to accelerate CNNs on mobile-friendly CPUs and MCUs. The framework is evaluated in the context of a key use case for embedded computer vision, namely, monocular depth estimation, demonstrating the importance of a vertical approach to meet stringent application and hardware constraints. Finally, the last part of the dissertation deals with the deployment of energy-quality scalable CNNs. To this end, first, an energy-quality scalable system for monocular depth estimation named EQPyD-Net is described and characterized. Then, Nested Sparse CNNs, a class of low footprint, dynamic CNNs, are proposed to tackle inference tasks at the edge of the Internet-of-Things.

Overall, the contribution of this dissertation is threefold. Novel automated optimization techniques are presented to improve the efficiency of state-of-the-art CNNs with minimal to no accuracy loss. New dynamic knobs are introduced to extend the achievable accuracy-complexity trade-off at run time. Finally, the proposed optimizations show that working across multiple levels of the optimization stack pushes further the boundary of accurate CNNs that can be deployed on tiny embedded devices.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context and Motivation

The field of Deep Learning (DL), and more specifically a class of DL algorithms, the Deep Neural Networks (DNNs), has achieved astonishing results in the last decade. In 2015, a Deep Convolutional Neural Network (CNN) architecture, ResNet, achieved super human-level accuracy in the ImageNet classification task [1]. Only one year later, in 2016, a DL-based agent named AlphaGo won against the world champion Lee Sedol in the game of Go [2], and in 2018, the challenging task of translating a Chinese text to English was performed automatically by a DNN with human-level performance [3]. Thus, DNNs have become the standard backbones of data reasoning tasks in many different domains, like image and video processing [4], speech recognition and translation [5], and sensor data analysis [6].

Traditionally, DL-based applications have been deployed in cloud infrastructures running on large-scale datacenters, where both DNN training and inference are performed on powerful devices [7]. However, the proliferation of mobile and Internet-of-Things (IoT) applications has recently demanded moving the inference process from the cloud to "the edge", that is, on lightweight resource-constrained embedded systems. The need for high levels of user privacy, low latency, and low-cost requirements represent the main driving factors of such a paradigm shift. For example, the users of a healthcare application dealing with biometric information may have privacy concerns in sending their

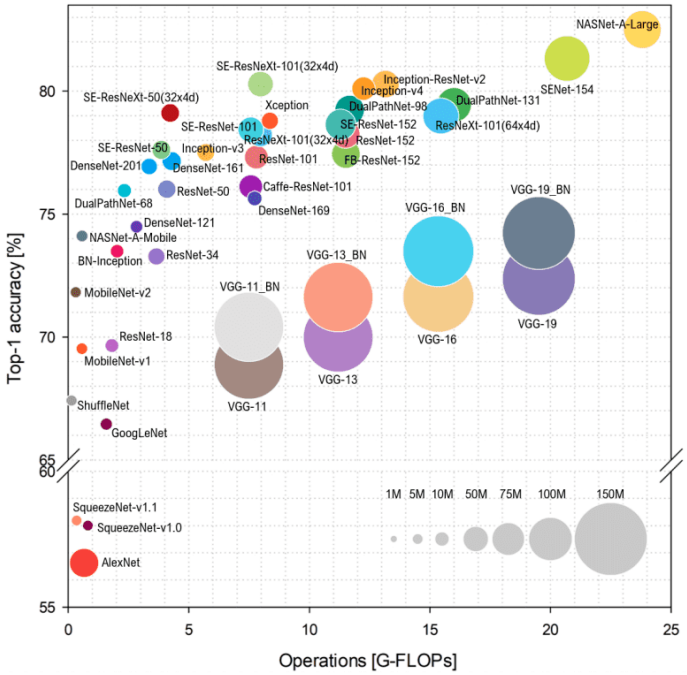


Fig. 1.1 Top-1 accuracy on ImageNet [8] vs. number of floating point operations (FLOPs) vs. model size. Image taken from [9].

data to the cloud. Applications relying on real-time sensing of the environment to perform fast decision-making, such as autonomous driving and augmented reality, cannot sustain the round-trip latency of the data transfer to the cloud. Finally, in the case of IoT applications based on a large-scale sensor network, the cost of transmitting data to a centralized server is simply unaffordable. Processing the DNN inference on-device guarantees higher levels of user privacy, as data stay local, and higher quality of service, as latency is much more deterministic. Moreover, the network congestion and the volume of data exchanged are highly reduced, lowering energy consumption and cutting down the infrastructure cost for distributed applications.

However, modern DNNs require massive computing power and considerable memory resources, making their deployment on resource-constrained embedded systems very challenging. Fig. 1.1 shows the relationship between the Top-1 classification accuracy on the ImageNet dataset and the computational and memory intensity for different DNN architectures. Most models require tens of GFLOPs and hundreds of MBs to achieve high accuracy. Unfortunately, mobile and IoT systems are usually battery-powered, with severe thermal, area,

and power constraints that limit the number of computational units and the on-chip memory resources. Therefore, bringing intelligence to such systems relies on the availability of small, fast, and energy-efficient DNNs.

This dissertation stems from the idea that the key to small, fast, and energy-efficient DNNs is *vertical and automated optimizations across the entire software stack*. Optimizations must act vertically across the different layers to holistically combine the benefits of specialization, from the algorithmic and neural architecture design process to compiler passes and computational strategies. Such a cross-layer approach not only can bring more remarkable gains but also allows the non-functional figures of merit to be co-optimized with the model accuracy. Optimizations must be automated to free embedded designers from the burden of manually dealing with the wide variety of DNN architectures used in different tasks and with the diversity of embedded platforms employed at the edge. Toward this goal, this dissertation presents a set of *novel automated optimization techniques* working across different levels of the software stack to improve the efficiency of DNNs on embedded systems. Specifically, this dissertation focuses mainly on CNNs as they were the dominant DNN architecture when the works presented here were developed. As the main outcome, the works presented in this thesis contribute to the state-of-the-art by pushing further the boundary of accurate CNNs that can be deployed on tiny embedded devices.

This chapter first discusses the optimization challenges involved in deploying DNNs on embedded systems, also introducing the minimal background needed to understand the rest of the dissertation (Section 1.2). Then, it introduces the main contributions of our research, together with the overall organization of the dissertation (Section 1.3).

1.2 Neural Network Optimization Stack

As shown in Fig. 1.2, the DL design, optimization, and deployment stack is usually structured in four main layers. At the highest level stands the *neural architecture design* process. This step aims at designing efficient DNN architectures, and it is usually carried out in a DL framework, like PyTorch [10], Tensorflow [11], or Paddle Paddle [12], together with other learning-related

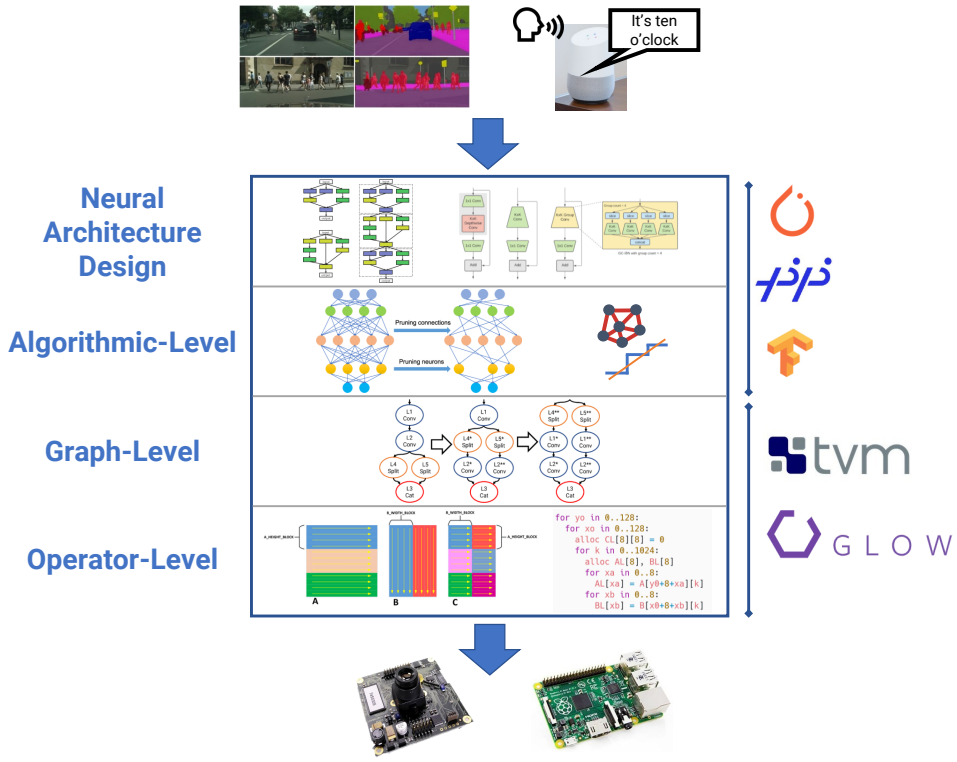


Fig. 1.2 DNN optimization stack.

steps, such as data preparation, data preprocessing, and training. Then, further optimizations can be performed at *algorithmic-level* by exploiting the intrinsic redundancy of DNNs and the statistical nature of DL. At this point, the trained and optimized model is fed as input to a DL compiler, like TVM [13] or GLOW [14]. Such compilers first translate the DL model into a high-level computational graph representation, then apply *graph-level* rewritings to generate an optimized graph, and finally perform *operator-level* optimizations to generate efficient code for each operator of the graph. This section briefly describes each layer of the stack, introducing the main optimization passes.

1.2.1 Neural Architecture Design

In its general embodiment, a DNN is a graph of tensor operators called *layers*. Fig. 1.3a reports an example of a generic DNN. A layer operates on one or multiple activation tensors, which are either fed as inputs to the network, e.g., the input image to classify, or are produced by previous layers — the orange

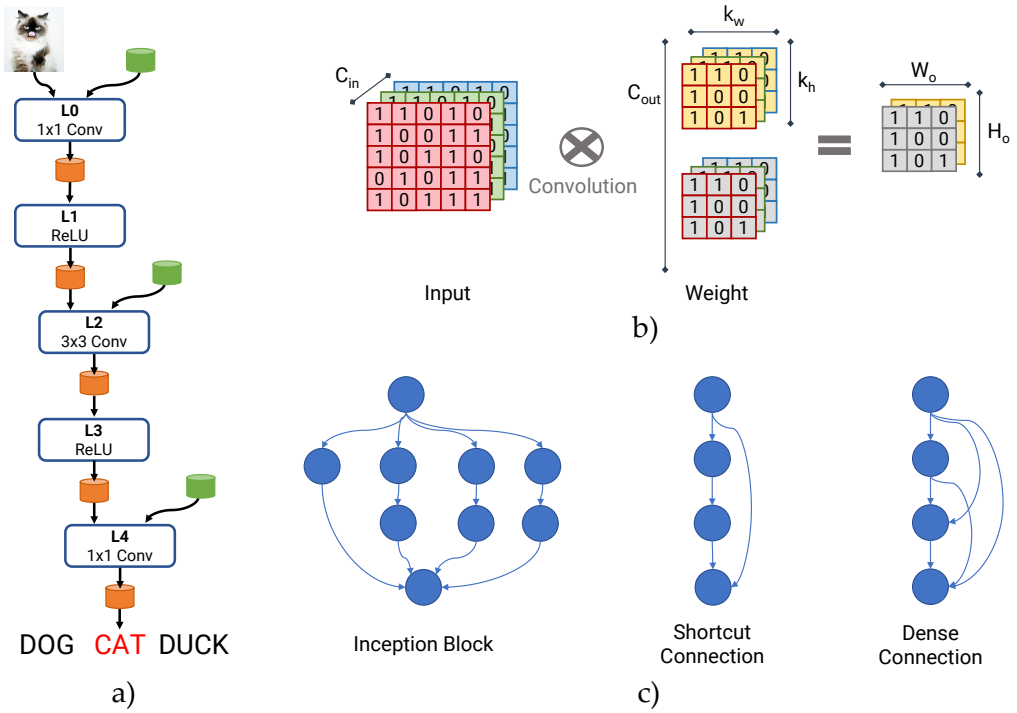


Fig. 1.3 a) A pictorial representation of a DNN; b) main parameters of a convolutional layer; c) three common neural modules.

boxes in Fig. 1.3a. The most commonly-used DNN layers are the convolutional, the fully connected, and the neural activation layers, like ReLU, Leaky ReLU, and Sigmoid. Apart from the activation tensors, the convolutional and fully-connected layers also accept one other tensor as input, usually called the weight tensor — the green boxes in Fig. 1.3a. The weight tensors are learned during the training phase but are constant during the inference stage.

In general, a DNN architecture is defined by the types and number of layers, the graph topology, and the dimensions of each layer. From Fig. 1.1, it is evident that different DNN architectures set a different trade-off in terms of accuracy, computational complexity, and model size. For this reason, several manual [15, 16] or automatic [17, 18] design methodologies have been recently proposed to find the DNN architecture that maximizes the accuracy while meeting latency, storage, or energy efficiency constraints. The following paragraphs introduce the main architectural parameters, highlighting their effect on performance.

Kernel size

A convolutional layer is defined by the number of input channels (C_{in}), the number of output channels (C_{out}), also called filters, and the kernel size ($k_h \times k_w$), also called the receptive field. All these parameters contribute to the computational complexity (FLOPs $\propto 2 \cdot H_{out} \cdot W_{out} \cdot k_w \cdot k_h \cdot C_{in} \cdot C_{out}$, where H_{out} and W_{out} are height and width of the output activation tensor) and to the parameter count ($|W| \propto k_w \cdot k_h \cdot C_{in} \cdot C_{out}$). A pictorial representation of a convolutional layer and its parameters is shown in Fig. 1.3b. Early CNN architectures, like AlexNet [19] or InceptionNet [20], used various kernel sizes, from 3×3 and 5×5 up to 11×11. However, the authors of VGG [21] demonstrated that convolutional layers with smaller receptive fields but more activation layers outperform architectures with bigger receptive fields. Thus, VGG [21] replaces the large convolutional kernels used in AlexNet with multiple 3×3 convolutional layers stacked together. Moreover, VGG also features 1×1 convolutions to increase the number of layers, hence the nonlinearity of the model, with a small computational cost. For this reason, virtually all modern CNN architectures adopt small filter sizes, namely, 1×1 and 3×3, to reduce both the computational complexity and the parameter count.

Depth

The depth of a network refers to the maximum number of layers traversed from the primary inputs to the primary outputs. Increasing the depth of the network enlarges the learning capacity of the model and hence allows the model to achieve higher accuracy [21]. However, it is intuitive that having more layers increases the computational cost and the number of parameters. Moreover, the training of very deep networks is rather challenging, as it suffers from stability issues due to the gradient vanishing during backpropagation [1]. The authors of InceptionNet [20] addressed such issues by having multiple auxiliary classifiers present in different parts of the model at training time and by adopting a more complex topology with parallel layers (see Fig. 1.3c). The authors of ResNet [1], instead, tackled the gradient vanishing problem by introducing the shortcut connection, which is a by-pass connection summing two tensors at different depths of the network (see Fig. 1.3c). The shortcut connection

allows very deep networks with hundreds of layers to be trained. Other recent works have proposed to modulate at run time the number of layers traversed using attention modules or gating blocks [22], possibly with the addition of intermediate classifiers called early-exit branches [23]. In this case, the latency of the DNN can be tuned at run time based on the difficulty of the inputs or on some external user-defined constraints, but at the cost of storing additional weights for the auxiliary modules.

Width

The width of the model refers to the number of filters of the convolutional layers (C_{out} in Fig. 1.3b). Reducing the number of convolutional filters lowers the size of the activation tensors, the number of parameters, and the computational complexity. Playing with the width of the convolutional layers is a simple but powerful architectural design trick, and it can be combined with other knobs to be even more effective. For example, the bottleneck module, proposed in InceptionNet [20] and ResNet [1], has become a standard module in modern CNNs. The bottleneck module comprises three stacked 1×1, 3×3, and 1×1 convolutional layers (see Fig. 1.3a). The first 1×1 convolutional operation lowers the number of channels to alleviate the computational load of the intermediate 3×3 layer. Then, the last 1×1 layer recovers the original number of channels. Similar architectural tricks have been used in other efficient CNN architectures targeting embedded systems such as MobileNet [15] and SqueezeNet [16]. The pioneering work by Howard et al. [15] proposed to scale the number of filters across all the layers according to a predefined ratio α, called the width multiplier [15]. In this way, different CNNs can be built starting from the same architectural template, which can be resized depending on memory and performance requirements. Follow-up works, e.g., Slimmable Networks [24], have elaborated on the concept of width scaling, enabling its use at run time to dynamically tune the latency of the model.

Resolution

The resolution of the intermediate feature maps is another important architectural knob (H_{out} and W_{out} in Fig. 1.3b). High-resolution feature maps

may contain fine-grain details, which help achieve high accuracy. However, processing high spatial resolution tensors significantly affects the computational complexity and the run-time working memory needed to process the network. For this reason, CNN architectures use pooling layers or the stride parameter of the convolutional layers to reduce the spatial resolution of the network. For instance, in CNN architectures for classification tasks, to balance the overall computational workload, the first layers operate on high-resolution feature maps with a small number of channels, whereas the last layers operate on low-resolution feature maps with a large number of channels. Scaling the resolution of the input tensors fed to the network [15] represents another way of controlling the resolution of the network. For this reason, similarly to the width multiplier, Howard et al. [15] proposed to scale the resolution of the input using a predefined ratio ρ, called the resolution multiplier, to build a family of CNNs from the same architectural template.

Graph Connectivity

While early CNN architectures, like AlexNet [19] and VGG [21], relied on a linear sequential topology, where all layers are stacked on top of each other, more recent CNN architectures have adopted more complex topologies to achieve higher accuracy with fewer parameters and fewer computations. For example, Fig. 1.3c shows the inception block with 4 parallel branches used in InceptionNet [20], the skip connection used in ResNet [1], and the skip connection added between all convolutional layers used in DenseNet [25]. Playing with the graph connectivity to improve the overall performance of the model has been largely used in automatic neural architectural design algorithms, e.g., [26]. Such algorithms are, in fact, usually formulated as search optimization algorithms aimed at finding the best connectivity among a pool of basic modules, like the one previously described, that maximizes the accuracy of the model under latency or memory constraints.

1.2.2 Algorithmic-level Optimizations

Once the neural architecture has been fixed, additional optimizations can be performed at the algorithmic level. There are two main techniques in this

space: *pruning*, aimed at removing the redundant parameters of the network, and *quantization*, aimed at reducing the complexity of the network through the use of low-precision arithmetic.

Pruning

Based on the assumption that DNNs are over-parametrized, pruning strategies aim to seek and remove the weights with a negligible contribution to the accuracy of the model. The existing methods differ in the policy used to identify the less important weights and the level of granularity at which they are applied [27]. In terms of policy, even if complex methods have been recently proposed, e.g., gradient or Hebbian-based methods [28], the magnitude-based [29] are the preferred option in many modern training pipelines due to the ease of their implementation. For what concerns the granularity, there exist three main classes. *Unstructured* pruning plays at the lowest level, namely, on the individual weights of the model [30], providing a high degree of flexibility to the learning process of the sparse model. Such flexibility comes at the cost of more complex processing for the remaining non-zero elements. Specifically, since non-zero elements are scattered across the tensor, they induce irregular data reading, processing, and writing patterns, which could be challenging to efficiently manage in systems with a cache-based memory hierarchy or wide SIMD compute units. Therefore, fully exploiting such unstructured sparsity necessitates additional hardware mechanisms to extract, communicate, and compute only with the non-zero elements [31]. At a coarser granularity, *Block* pruning techniques [32] group neighboring weights in specific patterns to decrease the indexing overhead and to ease the adoption of sparse compute kernels on general-purpose cores [33, 34]. At the coarsest level, *Filter* pruning drops entire convolutional filters [35], achieving aggressive storage savings and speed-up at the cost of substantial accuracy loss due to fast information removal.

Quantization

Pioneering works have shown that a 32-bit floating-point representation of weights and activations is redundant at inference time [30]. In fact, for many

tasks, existing DNNs can be quantized to a fixed-point representation with 16-bit or 8-bit integers [36] with no or minimal loss on the accuracy. More recent works have also proposed more extreme quantization, i.e., using 2 bits in ternary DNNs [37], or even 1 bit in binary DNNs [38]. However, ternary and binary DNN still suffer from a substantial accuracy drop, depending on the application and the network topology. In general, floating-point numbers can be discretized using either linear or non-linear schemes, as investigated over the years in the field of classic Digital Signal Processing (DSP). According to the linear scheme, the distance between two adjacent fixed-point values, called the quantization step, is constant across the entire input value range. Then, the mapping between real and integer values can be symmetric, if the integer distribution is centered around zero, or asymmetric, if it is shifted by a given offset. The first choice is simpler to implement, whereas the second can fit better the original floating-point value distribution but at the cost of additional processing stages to manage the offset at run time. Finally, the scaling factor used in the mapping can be a power-of-two, an arbitrary floating-point factor, or a fixed-point one. The former requires simple shift operations to be implemented [39], whereas the latter might be more accurate but at the cost of additional and more complex operations [40]. In the case of non-linear quantization, a non-linear function is used to map the real value range in the discrete set of integer values. Common approaches use logarithmic functions [41] or more complex clustering procedures [30]. Non-linear quantization schemes shine when the original data distribution is highly non-uniform. Obviously, their implementation introduces additional overhead, requiring more complex procedures to perform arithmetic operations between quantized numbers. Another key aspect for quantizing DNNs is the granularity of the quantization. Coarse-grain approaches [42] use the same fixed-point format for all the layers of the DNN, whereas more fine-grain approaches tune the format per-layer or per-channel [43]. Hybrid strategies may use a static bit-width for the whole model and a per-layer or per-channel fixed-point scaling factor [44]. Unfortunately, a one-size-fits-all solution does not exist, and so the quantization scheme to adopt depends on the task, the specific DNN architecture, the underlying hardware characteristics, and the latency and memory constraints.

1.2.3 Graph-level Optimizations

Early machine learning frameworks, like Caffe [45], simply iterated over the layers of the DNN to perform the inference stage. Unfortunately, this node-visitor processing method is inefficient, as it misses several optimization opportunities. As a result, modern machine learning frameworks, like PyTorch [10] and Tensorflow [11], hand over the DNN graph to domain-specific DL compilers [46, 14, 13], which perform a series of optimization passes to make the inference process on the target hardware more efficient.

When a DNN model is loaded into a DL compiler, it is first translated into a high-level Intermediate Representation (IR), which usually corresponds to a Dataflow Graph Representation (DFG). Several optimization passes are applied at this level to orchestrate the flow of macro operations, rather than the implementation of the single layer, i.e., the internal code organization. Such passes remove, modify, or add nodes to the graph while preserving the overall functionality. The following paragraphs briefly review the main graph-level optimization passes proposed in the literature, while the reader can refer to [47] for a more detailed overview.

Node simplification

These optimizations simplify one or more nodes to reduce the computational workload [48] or the memory requirements [49]. Such simplifications exploit the algebraic rules, namely, commutativity, associativity, and distributivity, in the case of linear algebra operators, or hand-written rewriting rules in the case of DL-specific operators, e.g., reshape, transpose, and pooling operations. Additional simplifications can be performed extending classic compiler optimizations, such as common sub-expression elimination (CSE) and dead code elimination (DCE), to work on the graph IR.

Layout assignment and transformation

The data layout assignment pass aims at finding the best data layouts for storing the intermediate activation tensors, adding layout transformation nodes if necessary. As data layouts of the input and output tensors have a considerable

influence on the final performance of an operator and the transformation operations may add significant overhead, the layout assignment is a challenging problem, which has been tackled either through heuristic [46, 14] or search-based algorithms [50].

Static memory allocation

As virtually all DNNs have static shapes, the memory allocation step can be entirely carried out at compile time. Specifically, one contiguous memory region, called the memory pool, is allocated and partitioned through a simple offset assignment to obtain the memory buffers for all the concurrently active tensors. The static memory allocation is usually formulated as an optimization problem aimed at finding the placement of the memory buffers in the pool that minimizes the overall memory requirement. Additional memory optimizations could be performed by reusing the same memory for both the input and the output tensors of an operation, usually known as the in-place optimization, and by scheduling the nodes such that the peak activation memory footprint is minimized [49].

1.2.4 Operator-level Optimizations

Once the graph-level optimizations have been performed, DL compilers usually lower the graph-level IR into a low-level IR to perform additional hardware-specific optimizations. Specifically, a series of passes are used to optimize the implementation of each operator, i.e., each node of the graph. The commonly applied passes include hardware-intrinsic mappings, memory latency hiding, parallelization, and loop-oriented optimizations, like loop tiling, loop reordering, and vectorization. Operator-level passes are usually highly specialized for the target platform, as different hardware architectures may require different implementation choices. Code optimizations can be performed automatically by the compiler using code generation techniques based on auto-scheduling, e.g., using the polyhedral model [51], or on auto-tuning [52]. Alternatively, the compiler can map the operators of the graph on the most suitable implementation extracted from highly-optimized libraries provided by the hardware vendors, e.g., cuDNN [53], ARM Compute Library [54], CMSIS-NN [39].

Note that there also exist some optimizations spanning multiple levels of the stack. For example, operator fusion bridges the graph-level and the operator-level of the stack by combining multiple operators into one. Operator fusion eliminates intermediate allocations and reduces memory bandwidth requirements, launch, and synchronization overhead. TVM [13] and BrainSlug [55] perform operator fusion based on pre-defined fusion rules, whereas Tensor comprehensions [51] exploits the polyhedral representation. However, how to identify and fuse more complicated sub-graphs is still an open challenge, especially considering its interaction with other optimization passes, such as layout assignment and tensor rematerialization [50].

1.3 Objectives and Contributions

This dissertation proposes to tackle the design and deployment of small, fast, and energy-efficient CNNs on embedded systems from a holistic system perspective. The objectives of this dissertation can be summarized as follows:

- Develop novel automated optimization techniques to reduce the memory footprint, lower the inference processing time, and increase the energy efficiency of state-of-the-art CNNs with minimal to no accuracy loss.
- Devise dynamic knobs to extend the achievable accuracy-complexity trade-off of a CNN at run time while still fulfilling the strict compute and memory constraints of embedded platforms.
- Demonstrate that the combination of different optimizations working across multiple levels of the optimization stack opens up new regions of the solution space, pushing further the boundary of accurate CNNs that can be deployed on embedded devices.

Fig. 1.4 shows the organization of the thesis, which is divided into three main chapters, each one focusing on a specific optimization objective.

Chapter 2 focuses on how to build *small* CNNs, i.e., on memory-driven optimizations. First, it reviews the problem of memory allocation in DL compilers, surveying the most adopted problem formulations and presenting a

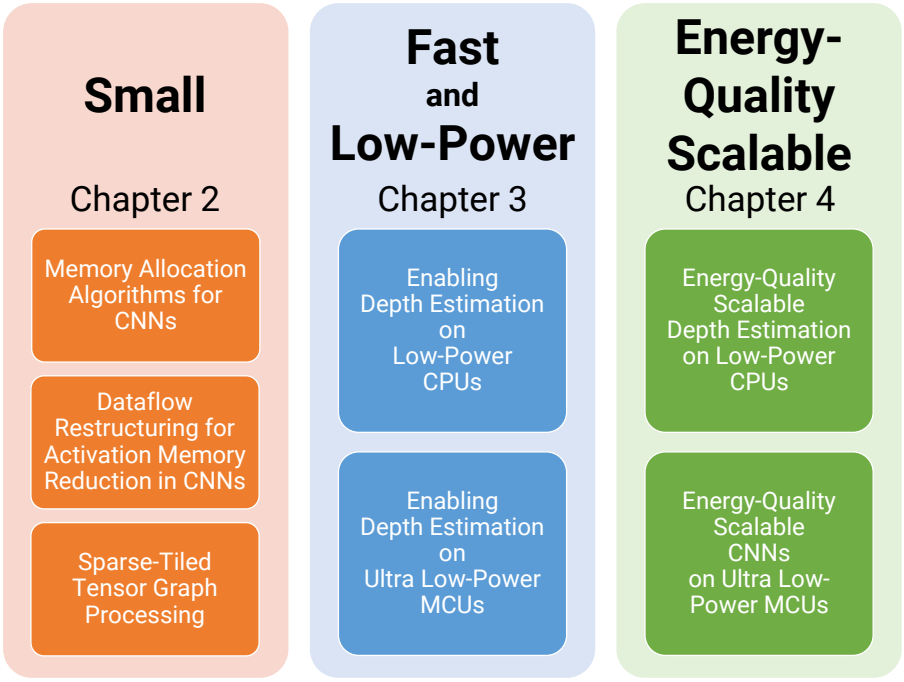


Fig. 1.4 Pictorial representation of the the dissertation outline.

quantitative assessment of the underlying optimization methods. The results collected over 851 benchmarks reveal that the most recent heuristics integrated into state-of-the-art compilers often fail to achieve the global optimum, with a substantial memory penalty (up to 33.6%). On the other hand, a mixed-integer linear programming method finds the optimal solution with a negligible run time (1.69 s on average). Such findings suggest the need to revisit the dominant trends in memory allocation for CNNs, shifting the focus of memory optimization on higher levels of the optimization stack (see Fig. 1.2). To this end, the second part of the chapter introduces a novel, automated method for minimizing the memory footprint of the intermediate activations of a CNN. Specifically, the proposed technique relies on graph-rewriting rules, which exploit algorithmic-level characteristics of CNNs, namely, the spatial locality of tensor operators and the change of the spatial resolution across the layers, to lower the activation memory footprint without affecting the functionality. Results collected on a representative class of different CNN architectures show that the proposed method is widely applicable and highly effective, achieving remarkable memory savings (62.9% on average) with low computational overhead (8.6% on average). The last part of the chapter presents a new compression pipeline,

which plays at the algorithmic level by pruning the unimportant weights of the network and at the graph level with the dataflow restructuring technique to reduce both weights and activations footprint with minimum accuracy loss. The collected results reveal that the proposed pipeline opens up a new feasible region of the memory-accuracy solution space, showing fast configurations of MobileNets deployed at full scale on tiny MCU devices, like an ARM M7 core equipped with 512KB of RAM and 2MB of FLASH memory.

Chapter 3 focuses on how to build *fast* CNNs suitable for tackling challenging inference tasks on *low-power* devices like ARM Cortex-A CPUs and ARM Cortex-M MCUs. In this chapter, we present an end-to-end optimization framework based on neural architectural design, quantization, and optimized integer kernels. The framework is evaluated in the context of a key use case for embedded computer vision, namely, monocular depth estimation, to demonstrate the importance of a vertical approach when targeting stringent application and hardware constraints. Specifically, we first introduce a comprehensive design and optimization framework aimed at accelerating the inference of PyD-Net, a lightweight CNN capable of achieving close to state-of-the-art accuracy with ultra-low resource usage. The accuracy driven optimization framework combines a hardware-friendly fixed-point quantization method with integer neural kernels custom-tailored to the specific target platform. The experimental evaluation performed on a multi-core ARM Cortex A53 CPU shows marginal accuracy loss on the KITTI dataset [56] with 16-bit (8-bit) integers, latency reduction up to 1.16× (1.64×), and memory footprint reduction up to 2× (4×) compared to single-precision floating-point. The second part of the chapter deals with the scaling of monocular depth estimation "at the very edge", i.e., on tiny MCU-powered platforms. In particular, it introduces μPyD-Net, a lightweight CNN vertically designed to be deployed on microcontrollers. The neural architecture design process of μPyD-Net is driven by the need to meet hard memory and latency constraints. At the algorithmic level, the network is trained in a peculiar self-supervised manner, leveraging proxy labels obtained through a traditional stereo algorithm, and is quantized to 8-bit data. At the operator level, the network is mapped to low-level layers custom-tailored to the target microcontroller architecture. Experimental results on standard datasets and an in-depth evaluation with an ARM Cortex-M7 MCU prove the feasibility

of obtaining sufficiently accurate monocular depth cues even on ultra low-power microcontrollers.

Chapter 4 focuses on the deployment of *energy-quality scalable* CNNs on embedded systems. Specifically, this chapter builds upon the idea that by leveraging the error resilience of many real-life applications, the quality-of-result can be gracefully degraded at run time to achieve higher energy efficiency, depending on the specific task, the external context, or the battery level. The first part of the chapter describes and characterizes an energy-quality scalable system for monocular depth estimation, named EQPyD-Net. It describes the architecture of the CNN and the optimization flow, covering the knobs that enable the dynamic scaling, namely, the scalable neural topology and the variable arithmetic precision of the custom operators. Tested on an off-the-shelf ARM Cortex A53, EQPyD-Net can be shifted across five operating points, ranging from a maximum accuracy of 82.2% on the KITTI dataset with 0.4 Frame/J up to 92.6% of energy savings with 6.1% of accuracy loss. Nevertheless, EQPyD-Net still has a minimal memory footprint of 5.2 MB for the weights and 38.3 MB (in the worst-case) for the run-time processing. The second part of the chapter introduces jointly-designed training and compression techniques to build Nested Sparse CNNs, a class of dynamic CNNs suited for inference tasks deployed at the edge of the IoT. A Nested Sparse CNN consists of a single CNN containing N sparse sub-networks with nested weights subsets, like a *Matryoshka* doll, and can trade accuracy for latency at run time, using the model sparsity as a dynamic knob. When tested on image classification and object detection tasks on an off-the-shelf ARM-M7 Micro Controller Unit (MCU), Nested Sparse CNNs outperform dynamic solutions naively built assembling single sparse models and state-of-the-art dynamic strategies, like dynamic pruning and layer width scaling.

Finally, Chapter 5 concludes the dissertation, summarizing the main findings of our research.

Chapter 2

Memory Optimizations for Convolutional Neural Networks

2.1 Introduction and Motivation

As discussed in chapter 1, the quest for CNNs that can be deployed on a wide variety of embedded architectures has accelerated the development of multi-stage pipelines where not just accuracy but also non-functional metrics, such as energy consumption, latency, and memory footprint, play as concurrent variables to optimize. Among all performance factors, the memory footprint of a CNN represents one of the biggest concerns as memory is the most expensive hardware resource. Specifically, the memory cost of a CNN is split among two components: weights and activations. The weight footprint depends on the number of parameters of the network, whereas the activation footprint depends on the memory needed to store the intermediate results, i.e., the feature maps produced during the inference process.

While reducing the number and the bitwidth of the learned weights has been treated initially as the crucial (and in many cases the only) aspect of the memory optimization problem, lowering the activation footprint has lately become even more critical. There are three main factors behind this new trend. First, CNNs have been successfully applied to novel applications, like image demosaicing [57] or depth estimation [58], where the feature maps resolution is much higher than that required by image classification tasks. Second, hardware-aware auto-ML

tools have become more commonly used to design efficient but irregular neural architectures [59] where the weight-to-activation ratio has reduced substantially. For instance, SwiftNet [60], one of the winning submissions of the Visual Wake Words competition [61], occupies 250 KB for the weights and 200 KB for the activations[1]. Third, off-the-shelf MCUs used in many IoT systems (such as the ARM Cortex-M cores [62]) are characterized by minimal memory resources: 1−2MB of FLASH for storing the weights of the model and tens to hundreds of KBs of on-chip SRAM to store the feature maps. The availability of such few memory resources results in strict constraints that must be satisfied to make the deployment of a CNN feasible on an MCU-powered device. Moreover, the available memory to reserve for the CNN inference can be further reduced due to other routines running concurrently on the same device.

In order to reduce the memory footprint of a CNN, several techniques have been proposed in the last few years, and they can be divided into two main categories: data-driven and data-independent. The data-driven techniques attack the problem by leveraging the statistical nature of DL, and hence they usually work at the neural architectural design and algorithmic level of the stack (see Section 1.2). Lightweight neural architectural design [16], resolution and topology scaling [15], quantization to low-precision integers [40, 63], and weight pruning [33, 64], represent the most adopted and effective optimization strategies belonging to this category. Such techniques require access to the training data as they must be embedded in the training procedure, they are model- and task-dependent, and, in some cases, they may cause an unrecoverable accuracy loss. On the other hand, data-independent methods tackle the memory reduction problem from a computational perspective, namely, applying data-independent transformations during the compilation pipeline, and hence they work at the graph- and operator-level of the stack (see Section 1.2). For instance, they play with the scheduling of the tensor graph [49], or with the layout of the tensor operators [65]. Data-independent techniques are training-free, hence model- and task-agnostic, and can be superimposed to data-driven optimizations without any accuracy loss.

This chapter first surveys the optimization algorithms adopted in several state-of-the-art DL compilers to solve the memory allocation problem (Sec-

[1] https://github.com/newwhitecheng/vwwc19-submission

tion 2.2). Then, it introduces a novel data-independent technique, named the dataflow restructuring, that performs a functionality-preserving, automated graph restructuring procedure to reduce the activation memory requirements of a CNN (Section 2.3). Finally, it presents a novel memory optimization pipeline that combines data-driven DL techniques with a data-independent technique, the dataflow restructuring, to enable the deployment of accurate CNNs on tiny devices powered by MCUs (Section 2.4).

The content of this chapter is an extended, improved version of our previous publications found in [66, 67].

2.2 Memory Allocation Algorithms for CNNs

2.2.1 Motivation

Memory allocation is a crucial pass in any compiler infrastructure, but it is even more critical when memory-intensive workloads, such as CNNs, are deployed on resource-constrained embedded devices. Achieving the optimal memory allocation is, in fact, paramount as even a small overhead may prevent the deployment of a given CNN. In the case of CNNs with static shapes, one contiguous region of the memory, referred to as the *memory pool*, is reserved at initialization time and statically partitioned to make space for the input and output tensors of each layer. Specifically, partitioning the memory pool involves assigning to each tensor an offset address, which defines the portion of the pool reserved for the tensor during its lifetime. Since tensors are accessed by a limited number of layers scheduled sequentially, the memory space assigned to a tensor can be reused over time to store other non-conflicting tensors. Therefore, in ML compilers, the memory allocation step usually takes the form of an optimization problem aimed at finding the placement of tensors in the memory pool that minimizes its size.

Some recent works [68–72] have addressed the problem by proposing and implementing different greedy heuristics, some of which had also been integrated into commercial inference engines (e.g., TensorFlow Lite [73], PyTorch Mobile [74], Arm NN [75], and ONNC [76]). Unfortunately, these heuristics were validated on simple hand-crafted CNNs with a regular linear graph topology.

However, more recent studies in DL theory have promoted an emerging class of CNNs generated via Automated Machine Learning (AutoML), e.g., Neural Architecture Search [77]. AutoML CNNs achieve higher accuracy with fewer weights and fewer operations than hand-crafted models, guaranteeing better performance with fewer hardware resources, particularly memory. However, they also present an irregular graph topology characterized by complex connectivity, making the memory allocation problem much more challenging to solve efficiently in practice. Therefore, it is natural to ask whether prior heuristics can still achieve optimality as reported in the recent literature, and if not, how far they are from the global optimum. To this end, in this section:

- We review different formulations of the memory allocation problem introducing two widely adopted options. The first one relies on a constrained version of the two-dimensional strip packing problem; the second treats the memory allocation as a graph coloring problem.
- We survey existing algorithms proposed to solve the presented formulations. Our analysis includes both heuristic algorithms and an exact method relying on a MILP solver.
- Finally, we evaluate the discussed methods quantifying their efficiency in terms of memory minimization and processing time. The numerical results were collected from a large set of CNNs, including both hand-crafted and automatically generated architectures. About the latter, we picked 845 CNNs from the NAS-Bench-101 database [26] with different connectivity and depth (from 77 to 204 layers).

As the primary outcome, the experimental evaluation results demonstrate the sub-optimality of existing heuristic methods, currently the most common choice in standard tensor graph compilers, revealing substantial overhead (up to 33.6%) when tested on AutoML CNNs. Moreover, they empirically demonstrate that a MILP method can consistently achieve optimal solutions in a reasonable run time (1.69 s on average, 28.31 s in the worst case[2]).

[2]On a standard CPU-based server machine.

2.2.2 Problem Formulation

For memory allocation purposes, a CNN can be modeled as a DFG $\mathcal{G}(T, E)$, where $T = \{t_i | i = 0, \ldots, N-1\}$ is the set of nodes representing the N tensors read and written at run time, and $E = \{e_{i,j} | t_i, t_j \in T, i \neq j\}$ is the set of edges describing the dependencies among tensors. Fig. 2.1 shows the DFG for a simple CNN with six tensors. Each tensor t_i has a memory size h_i equal to the product of its cardinality with the adopted data type (e.g., 32-bit floating-point or 8-bit integer).

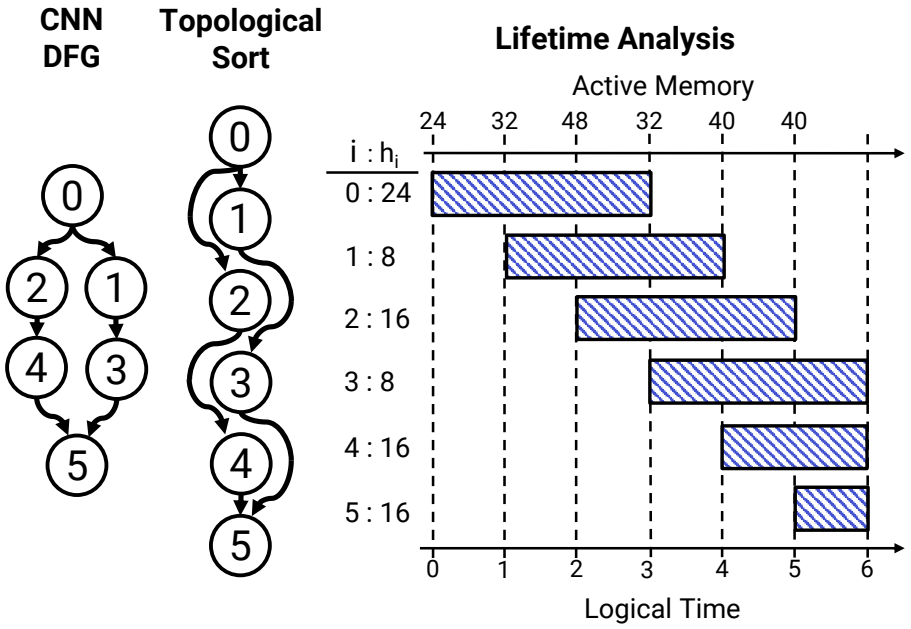


Fig. 2.1 On the left, the DFG representation of a CNN. In the middle, the topological order σ as a linearization of the DFG. On the right, the lifetime analysis and the active memory for the ordered DFG.

Inference engines for embedded systems, e.g., TensorFlow Lite [73] or PyTorch Mobile [74], process a CNN layer by layer sequentially, following a valid order obtained by a topological sort of the DFG. We denote the topological sort with the ordered set of tensors $\sigma = (t_i | t_i \in T)$, such that for any edge $e_{i,j} \in E$, t_i precedes t_j in σ; in Fig. 2.1, $\sigma = (t_0, t_1, t_2, t_3, t_4, t_5)$. For a given σ, the resulting lifetime w_i of each tensor t_i is the difference between its end- and start-time. The start-time x_i is defined as the position of t_i in the topological order σ, while the end-time is the start-time of its latest successor t_j ($e_{i,j} \in E$).

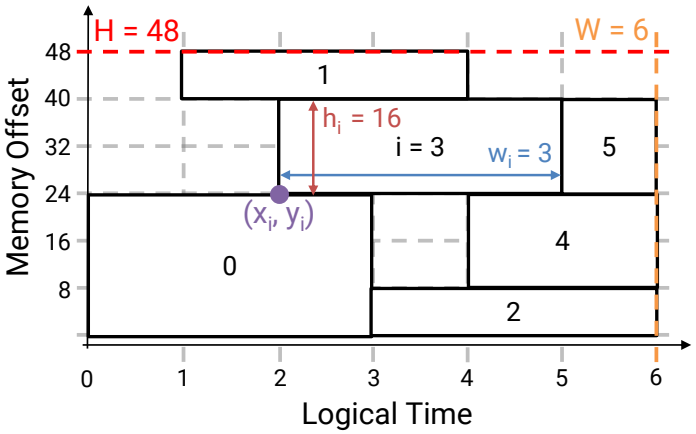


Fig. 2.2 Example of a memory allocation for the sorted DFG reported in Fig. 2.1.

The end-time of the last tensor in σ defines the overall processing time W of the DFG ($W = 6$ for the example in Fig. 2.1).

As stated previously, each tensor t_i is placed in a contiguous memory region called the memory pool. Allocating a tensor t_i within the memory pool corresponds to defining its address offset y_i. The total size of the tensors concurrently alive within a given logical time sets the active memory, whereas the total memory footprint H is equal to the maximum active memory. Fig. 2.2 shows a possible memory allocation for the sorted DFG of Fig. 2.1, highlighting the main variables and parameters of the problem (also summarized for the sake of clarity in Tab. 2.1).

The memory allocated for a tensor can be freed once the tensor is not accessed anymore, and it can be used for hosting other tensors. For instance, t_4 (starting at time 4) can reuse a fraction of the memory previously allocated for t_0 (ending at logical time 3); the same is not valid for t_1 or t_3 which are alive together. Thus, the memory allocation involves the following optimization problem: *search the optimal set* $\mu = \{y_i | t_i \in T\}$ *that minimizes the memory footprint* H *while guaranteeing that the intersection between the memory regions of concurrently active tensors is empty.*

The trivial solution is to place tensors into dedicated disjoint regions of the memory pool. The resulting memory pool size H is the sum of all the tensors size: this is the upper bound $H_{\max}$ in the solution space of the memory allocation problem ($H_{\max} = 88$ for the example of Fig. 2.1). The lower bound $H_{\min}$ is defined by the group of concurrent tensors with the highest memory

Table 2.1 Notation adopted for the main variables and parameters of the memory allocation problem.

Notation	Definition
$N \in \mathbb{Z}^+$	Number of tensors.
$T = \{t_0, \ldots, t_{N-1}\}$	Set of tensors.
$w_i \in \mathbb{Z}^+$	Lifetime of tensor t_i.
$h_i \in \mathbb{Z}^+$	Size of tensor t_i (in bytes).
$x_i \in \mathbb{N}$	Start-time of tensor t_i.
$y_i \in \mathbb{N}$	Memory offset of tensor t_i.
$W \in \mathbb{Z}^+$	Total logical time.
$H \in \mathbb{Z}^+$	Memory pool size.
$H_{\min} \in \mathbb{Z}^+$	Theoretical lower bound of the pool size.
$H_{\max} \in \mathbb{Z}^+$	Upper bound of the pool size.
σ	Topological order of the tensors.
$\mu = \{y_0, \ldots, y_{N-1}\}$	Memory allocation.

demand ($H_{\min} = 48$ in Fig. 2.1). $H_{\min}$ can be easily computed from the result of the liveness analysis. In general, $H_{\min}$ depends on the topological order σ, and many topological orders may exist for a given DFG. For instance, in Fig. 2.1 an alternative topological order is $\sigma = (t_0, t_2, t_4, t_1, t_3, t_5)$ with $H_{\min} = 56$. In our study, we consider the topological order as an input, and all the algorithms under analysis are compared using the same topological order (more details in Section 2.2.4).

Since most CNNs are static and the size of each tensor is fixed, the memory allocation problem can be solved at compile time before deploying the model on the target device. Then, the obtained memory offsets can be off-loaded into the target device in the form of a table, which is used at run time during the processing of the CNN.

2.2.3 Memory Allocation Algorithms

In this subsection, we first describe the formulation based on the two-dimensional strip packing problem and the related optimization strategies. Then, we present the graph coloring formulation, showing its intrinsic limitations.

2D Strip Packing Formulation and Methods

One way to solve the memory allocation problem is to cast it as a particular instance of the two-dimensional strip packing problem (2D-SPP). The objective of such a problem is to determine an overlapping-free packing of fixed-size rectangles into a strip of bounded width such that its height is minimized [78]. Recalling the description introduced in the previous section (see Fig. 2.2), each rectangle of size $w_i \times h_i$ represents a tensor, where the width is equal to its lifetime and the height its size in bytes. The coordinates of the bottom-left corner of each rectangle (x_i, y_i) define its placement in the strip. Specifically, x_i is the start-time of the tensor set by the topological order σ; y_i is the memory offset to be optimized. The width W of the strip is fixed and defined through the total logical time of the DFG, whereas its height H is the cost function to minimize. This formulation differs slightly from the general 2D-SPP, where both rectangle coordinates are optimization variables. The following equations formalize the problem:

$$\begin{aligned} &\text{minimize } H \\ &\text{subject to:} \\ &y_i + h_i \leq H,\ t_i \in T && (2.1) \\ &y_i + h_i \leq y_j \text{ or } y_j + h_j \leq y_i,\ (t_i, t_j) \in L && (2.2) \end{aligned}$$

where $L \subseteq T$ is the set of tensor pairs with overlapping lifetimes, i.e., $L = \{(t_i, t_j) |\ t_i, t_j \in T,\ [x_i, x_i + w_i) \cap [x_j, x_j + w_j) \neq \emptyset, i \neq j\}$. The constraint (2.1) guarantees that all tensors are placed inside the pool, while (2.2) that concurrent tensors occupy non-intersecting regions of the memory pool.

In the remainder of this section, we introduce the existing optimization methods [69–72, 76], specifically, the greedy algorithms and a MILP-based formulation that any standard mathematical optimization toolbox can solve, e.g., SCIP [79].

Greedy algorithms are the preferred option in current CNN compilers due to their low worst-case computational complexity (quadratic in the number of tensors). Tab. 2.2 summarizes the most representative ones. Specifically, we

Table 2.2 Taxonomy of the existing heuristic algorithms proposed as memory allocators for CNNs. ↓ indicates decreasing sorting order, whereas ↑ ascending sorting order.

Placement Policy	**Tensor Indexing**	**Offset Indexing**	**Notation**	**References**
Tensor-first	Size ↓	Best-fit	*gs-best*	[71, 72, 76]
		First-fit	*gs-first*	[70–72, 76]
	Start-time ↑	Best-fit	*fifo-best*	[76]
		First-fit	*fifo-first*	[70, 76]
	Start-time ↓	Best-fit	*lifo-best*	—
		First-fit	*lifo-first*	[70]
	Breadth ↓	Best-fit	*gb-best*	[72]
		First-fit	*gb-first*	[72]
Offset-first	Lifetime ↓	Lowest-first	*b2sp*	[69]

propose a new taxonomy of the heuristic algorithms based on the observation that they all share the same basic strategy, i.e., an iterative procedure where tensors are allocated one by one, but they differ for the placement policy. The placement policy encompasses the definition of two indexing criteria: (*i*) how to rank the tensors, and so how to select the tensor to allocate; (*ii*) how to select the free offset to assign to a tensor. These are the *Tensor* and *Offset* indexing columns reported in Tab. 2.2. Depending on which criteria gets served first, the placement can be classified as *tensor-first*, i.e., tensor selection before offset selection, or *offset-first*, i.e., offset selection before tensor selection.

The pseudocode for the *tensor-first* class of heuristics is reported in Algorithm 1. First, the pool size is set to zero (line 1) as no tensor is placed (line 2). Then, the tensors are sorted (line 3) and placed sequentially through an iterative procedure (lines 4–7). Different sorting criteria can be adopted, which differ for the key (*Size*, *Start-time*, or *Breadth*) or for the order (descending ↓ or ascending ↑): in the by-*size* indexing, tensors are sorted in descending order by their memory size; in the by-*start-time* indexing, the ordering can be either ascending (*fifo*) or descending (*lifo*); in the by-*breadth* indexing, tensors are sorted in descending order using the total active memory at their start-time as ordering key (in Fig. 2.1, the breadth of tensor t_1 is 24, the breadth of t_2 is 32).

Algorithm 1: Tensor-First Memory Allocation

Input: A set of tensors T and their topological order σ.
Output: The memory pool size H and tensor placement μ obtained according to the adopted *Tensor* and *Offset* indexing policies.

1 $H \leftarrow 0$
2 $\mu \leftarrow \emptyset$
3 $T^* \leftarrow \texttt{Sort}(T, \texttt{TensorIndexing})$
4 **for** $t_i \in T^*$ **do**
5 $\quad y_i \leftarrow \texttt{GetOffset}(t_i, \mu, \texttt{OffsetIndexing})$
6 $\quad \mu \leftarrow \mu \cup \{y_i\}$
7 $\quad H \leftarrow \max(H, y_i + h_i)$
8 **return** (H, μ)

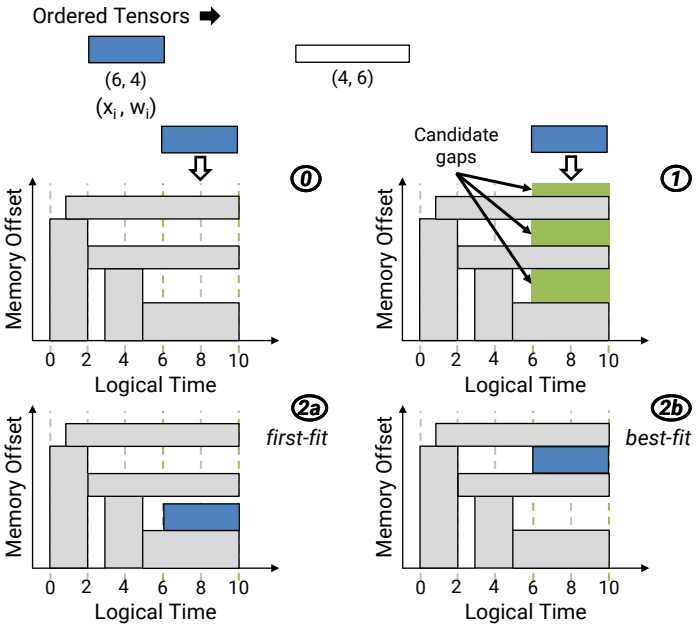


Fig. 2.3 Schematic view of the tensor-first heuristic algorithm. The picture refers to a *by-size* tensor policy. The bottom row shows the first-fit (step 2a) and best-fit (step 2b) offset indexing policies.

The for-loop (line 4) iterates over the ordered list T^* pulling one tensor at a time and selecting the best offset available y_i (line 5); the placement and the memory pool size are updated accordingly (line 6–7). The loop ends when T^* is empty, returning the pool size and the memory placement (line 8).

As graphically depicted in Fig. 2.3, once a tensor is picked from the list T^* (step 0), there are several possible offset candidates (step 1). According to the

first-fit indexing option (step 2a), the lowest available offset that does not cause overlapping with already placed tensors is assigned to the tensor. According to the *best-fit* indexing instead (step 2b), the tensor is placed into the smallest gap available, still avoiding overlaps. If there is a lack of memory space, the memory pool is extended upward by assigning the first available offset on top of the currently placed tensors.

Overall, the tensor and offset indexing criteria can be combined, originating eight different versions of the tensor-first allocation method (see Tab. 2.2).

The pseudocode of the *offset-first* class of heuristics is reported in Algorithm 2. As for the tensor-first algorithm, the initialization stage (lines 1–2) is followed by the optimization loop (lines 3–13). In the optimization loop, the first decision involves the offset selection (lines 4–9), and it works as follows. First, based on the current placement μ, the lowest and leftmost offset y^* among those still available is extracted (line 6). Second, the non-placed tensor list is indexed to search the tensors that fit into y^* (line 7). These tensors are stored in a temporary set of candidates T^*. If none of the non-placed tensors fit the current offset y^* (line 8), then the current offset is merged with the lowest adjacent one to create a larger gap at the cost of leaving a portion of the memory space unused (line 9). As soon as a non-empty subset of tensor candidates is found (line 5), the one with the longest lifetime is picked (line 10) and placed by assigning to it the offset candidate y^* (line 11). Then, the optimal placement and the memory pool size (line 12-13) are updated, and the outer loop repeats until all the tensors are placed. Finally, the pool size and the memory allocation are returned (line 14).

Fig. 2.4 gives a representation of the optimization loop and its implementation details. Among the four offsets available at the current iteration, those highlighted in green (step 0), the lowest and leftmost offset is selected: the one highlighted in red (step 1). Since none of the non-placed tensors can fit the available room without overlapping already placed tensors, the offset is merged with the lowest adjacent one (step 2). As multiple tensors can now be placed at the current offset, those highlighted in blue (step 2), the priority is given to the one with the longest lifetime. The selected tensor is placed, and then the available offsets are updated for the next loop iteration, as highlighted in green (step 3).

Algorithm 2: Offset-First Memory Allocation.

Input: A set of tensors T and their topological order σ.
Output: The memory pool size H and tensor placement μ.

$H \leftarrow 0$
$\mu \leftarrow \emptyset$
while $|\mu| < |T|$ **do**
  $T^* \leftarrow \emptyset$
  while $T^* = \emptyset$ **do**
    $y^* \leftarrow \texttt{GetOffset}(\mu)$
    $T^* \leftarrow \texttt{GetFittingTensors}(T, y^*, \mu)$
    if $T^* = \emptyset$ **then**
      $\texttt{UpdateCandidateOffsets}(\mu)$
  $t_i \leftarrow \text{argmax}_{t_j \in T^*} w_j$
  $y_i \leftarrow y^*$
  $\mu \leftarrow \mu \cup \{y_i\}$
  $H \leftarrow \max(H, y_i + h_i)$
return (H, μ)

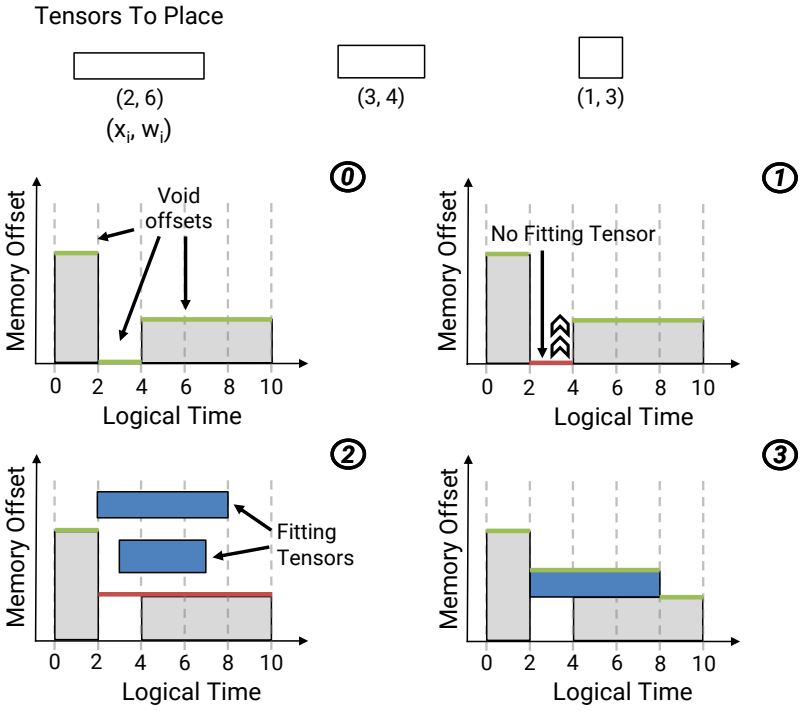


Fig. 2.4 Schematic view of the offset-first heuristic algorithm.

Given that the proposed heuristics are characterized by low computational complexity, a simple yet effective ensemble strategy can run them all to de-

termine the solution that performs best [72]. This strategy, which we refer to as *bag-of-heuristics*, works under the assumption that different heuristics may fail to achieve optimality on different networks. Therefore, their combination may find the global optimum or reduce the memory overhead on average.

Alternatively to the greedy algorithms, a MILP solver can be used for the memory allocation problem after performing a linear relaxation of the logical constraints in (2), for example, through the *big-M* method. The analytical formulation is as follows:

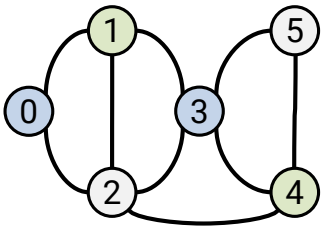
$$\begin{aligned} &\text{minimize } H \\ &\text{subject to:} \\ y_i + h_i \leq H,\ t_i \in T \quad &(2.3) \\ y_i + h_i \leq y_j + p_{ij} \cdot M,\ (t_i, t_j) \in L^3 \quad &(2.4) \\ y_j + h_j \leq y_i + (1 - p_{ij}) \cdot M,\ (t_i, t_j) \in L \quad &(2.5) \end{aligned}$$

The M refers to the "Big" value associated with the artificial binary variables p_{ij}:

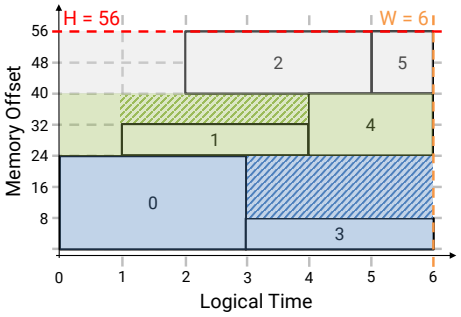
$$p_{ij} = \begin{cases} 1, & \text{if } x_i + w_i \leq x_j \\ 0, & \text{if } x_j + w_j \leq x_i \end{cases} \quad (t_i, t_j) \in L$$

As we observed during our experiments, the optimization time is substantially affected by the value of M, especially for large networks, and an overestimation of M might prevent the branch-and-cut method [80] from converging to a feasible solution [69]. It is indeed paramount to estimate a reasonable lower bound for M, which should be large enough to guarantee the existence of a feasible solution but small enough to ensure a run time of the MILP solver acceptable in a compilation flow. We empirically observed that $M = 1.5 \cdot H_{\text{min}}$ is a good value for our suite of benchmarks (851 CNNs as described later in Section 2.2.4). Should this choice fail, we suggest setting M following the methodology proposed in [81].

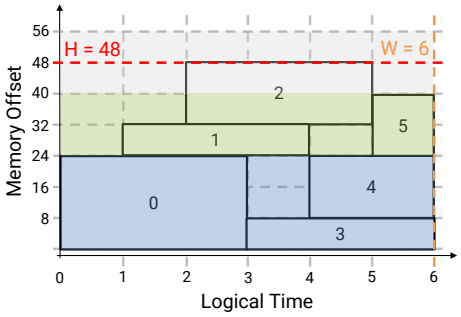
[3] L is the set of tensor pairs with overlapping lifetimes.



a) Conflict Graph.



b) Coloring-based offset assignment.



c) Post-processing [68].

Fig. 2.5 Schematic view of memory allocation with graph coloring.

Graph Coloring Formulation and Methods

The memory allocation of CNNs can also be formulated as a graph coloring problem. This formulation relies on building a conflict graph, where the nodes are the tensors, and the edges represent the conflict among tensors; here, conflict refers to tensors with overlapping lifetimes. The graph coloring problem consists of assigning a color to each node of the conflict graph such that connected nodes have different colors and the number of colors is minimum. Fig. 2.5a shows the conflict graph for the example reported in Fig. 2.1 and a minimum color assignement.

Graph coloring algorithms are commonly adopted to solve the register allocation problem [82] in compilers for general-purpose CPUs, which consists of mapping each local variable of a piece of code into the lowest number of internal registers. In the register allocation problem, however, all variables share the same size, whereas tensors generally differ in size in the case of CNNs. As illustrated in the example of Fig. 2.5b, this is the primary source of sub-optimality. In fact, the memory pool is split into disjoint regions, one for each color in the optimal color set, and each region has a size equal to the size

of the largest tensor assigned to that color. Such a placement can under-utilize the available space when tensors of different sizes share the same color. The authors of [68] tried to address this issue by introducing a post-processing stage after the standard graph coloring stage. The post-processing stage involves an iterative procedure that lowers the offset of tensors with available free space at the bottom (Fig. 2.5c). However, the proposed solution does not have any optimality guarantee.

2.2.4 Experimental Results

Benchmarks

We evaluated the allocation methods on two sets of CNNs. The first consists of six hand-crafted architectures commonly adopted in computer vision applications for image classification and feature extraction in segmentation and object detection tasks. GoogLeNet [83] is one of the pioneering works that introduced irregular connectivity in CNNs, demonstrating that an irregular topology improves the prediction accuracy. It is built upon a stack of multiple *Inception* modules composed of four parallel branches converging in a concatenation layer. Different instances of the *Inception* module were proposed by more recent network designs, such as Inception-v3 [20], where the *Inception* module varies in the number of branches or in the number of layers in each branch. ResNets [1] introduced the residual connection, which is a by-pass connection summing two tensors at different depths of the network. Several ResNets exist, each with the same underlying topology but with a different layer count; we adopted the ResNet-18 version. MobileNetV1 [15] proposed the use of depthwise separable convolutional layers to reduce the number of parameters and operations for mobile applications. MobileNetV2 [84] is an extension of MobileNetV1; it is made up of a stack of inverted residual blocks, where residual connections are used along with depthwise separable convolutions. DenseNets [25] are networks with a higher connection density: each layer takes as inputs the output tensors produced by all the previous layers and feeds all the subsequent layers. As for the other CNNs, different-sized networks are available and, with no lack of generality, we picked the DenseNet-121 instance. The adopted hand-crafted

Table 2.3 Overview of the six hand-crafted CNNs adopted as benchmarks.

CNN	n	**# Constr.**	$H_{\max}$ [MB]	$H_{\min}$ [MB]
GoogLeNet [83]	85	542	6.38	1.15
Inception-v3 [20]	122	867	13.50	1.98
ResNet-18 [1]	33	136	3.28	0.96
MobileNetV1 [15]	29	86	4.81	1.15
MobileNetV2 [84]	64	251	6.58	1.44
DenseNet-121 [25]	248	1091	26.71	1.72

models serve as test-benches to validate our implementation of the methods presented in Section 2.2.3.

Tab. 2.3 summarizes the main features of the selected benchmarks. It reports on the total number of tensors (column n) and the number of MILP constraints (column *# Constr.*), which can be used as indirect proxies to infer the complexity of each instance, together with the upper-bound of the memory footprint ($H_{\max}$), obtained through a naive allocation strategy, and the theoretical lower-bound ($H_{\min}$). The memory requirements refer to an 8-bit data-type implementation, which is a common implementation choice for CNNs on embedded systems. The gap between $H_{\max}$ and $H_{\min}$ quantifies the maximum savings achievable by an optimal memory allocation. The optimization margin is large, from 3.4× to 15.5×, motivating the need to investigate efficient memory allocation methods.

The second set includes a collection of CNNs picked from the NAS-Bench-101 [26], an open-source dataset that was designed to provide support for the research of AutoML search algorithms. These CNNs are built by stacking three instances of the same cell; different architectures make us of cells with a different topology as the backbone. Specifically, each cell has a maximum of seven layers and nine interconnections; each layer can be chosen among one of three different operators. Overall, the dataset contains 423k CNN architectures varying in size and structure. Among them, we considered 845 architectures with a unique connectivity pattern across the cell layers; the remaining ones do not bring any additional value to our analysis, as they differ for the kind of arithmetic operation applied, and so they can share the same memory

Table 2.4 Overview of the 845 NAS-Bench-101 CNNs adopted as benchmarks.

	Min.	**Average**	**Max.**
$\boldsymbol{n}$	79	154	205
# Constr.	236	856	1346
$\boldsymbol{H}_{\max}$ **[MB]**	2.93	5.57	14.74
$\boldsymbol{H}_{\min}$ **[MB]**	0.25	0.30	0.63

Table 2.5 Memory pool size (H) in MB and execution time (Time) in s for different allocation algorithms. Solutions achieving optimal memory ($H = H_{\min}$) are highlighted in bold.

Benchmark		**gs-best**		**gs-first**		**gb-best**		**gb-first**		**b2sp**		**milp**	
	$H_{\min}$	H	Time	H	Time	H	Time	H	Time	H	Time	H	Time
GoogLeNet [83]	1.15	**1.15**	0.01	**1.15**	0.01	**1.15**	0.01	**1.15**	0.01	**1.15**	0.02	**1.15**	0.16
Inception-v3 [20]	1.98	**1.98**	0.01	**1.98**	0.01	2.05	0.02	2.05	0.02	**1.98**	0.03	**1.98**	0.13
ResNet-18 [1]	0.96	**0.96**	0.01	**0.96**	0.01	**0.96**	0.01	**0.96**	0.01	**0.96**	0.02	**0.96**	0.03
MobileNetV1 [15]	1.15	**1.15**	0.01	**1.15**	0.01	**1.15**	0.01	**1.15**	0.01	**1.15**	0.01	**1.15**	0.06
MobileNetV2 [84]	1.44	**1.44**	0.01	**1.44**	0.01	1.63	0.01	1.63	0.01	**1.44**	0.02	**1.44**	0.04
DenseNet-121 [25]	1.72	1.91	0.03	1.91	0.03	2.11	0.04	2.11	0.04	**1.72**	0.08	**1.72**	0.52

allocation. Tab. 2.4 collects the statistics, emphasizing the diversity of the selected benchmarks, both in terms of problem complexity (n and # Const.) and memory requirements ($H_{\min}$ and $H_{\max}$), which is an important aspect to consider for testing the efficiency of the memory optimization.

Software Tools and Hardware Specifications

We resorted to a custom implementation of the methods described in Section 2.2.3 and summarized in Tab. 2.2. Our assessment includes six different algorithms: *gs-best*, *gs-first*, *gb-best*, *gb-first*, their combination in a *bag-of-heuristics*, and the MILP formulation. We omitted (*i*) the heuristic algorithms based on start-time tensor policies (*fifo-best*, *fifo-first*, *lifo-best*, and *lifo-first*), and (*ii*) the formulation based on graph coloring, since prior works have shown their inferior performance even on the most simple handcrafted CNNs [70, 76]. The heuristic methods were implemented in Python, whereas we adopted the SCIP framework [79] as MILP solver. Even though Python is known to be not

performance-oriented, we observed that the heuristic methods require negligible CPU time (more details later in the text), making custom code optimization unnecessary and expensive. As an additional note, the CPU time refers to single-thread execution on a workstation powered with an Intel Xeon Silver 4114 CPU running at 2.2 GHz.

Results on Hand-crafted CNNs

Tab. 2.5 shows the results for the set of hand-crafted CNNs, reporting for each method the returned memory pool size and the execution time of the optimization. The collected numbers confirm the trend already observed by previous research [72], validating our implementation of the different methods. Overall, the heuristics achieve a pool size close or equal to the theoretical lower-bound H_{min}, within negligible time (0.08 s in the worst-case). As one can observe, *b2sp* gets the best result with a marginal time overhead. Compared to the numbers reported in [72], there are slight differences due to the topological order of the tensors (Section 2.2.2), which is an input of the memory allocation process. Interestingly, the table reveals that the MILP solver reaches the global optimum in negligible time as well (less than 1 s for all test cases). This result does contrast with the existing literature claiming MILP-based approaches are unfeasible. It is, however, true that the value of the big-M may affect performance and hence must be properly tuned, as already mentioned in Section 2.2.3.

Results on NAS-Bench-101

Tab. 2.6 summarizes the statistics of the results collected on the NAS-Bench-101 CNNs. The row *Optimal* shows the percentage of cases reaching the lower-bound H_{min}, *Avg. Ovhd.* and *Max. Ovhd.* the average and maximum relative distance from the optimal solutions, *WCET* the Worst-Case Execution Time, and *ACET* the Average-Case Execution Time.

Unlike hand-crafted networks, no heuristic achieves H_{min} in all cases. In the best case (column *gs-first*), only 35.5% instances are solved with an optimal allocation; the remaining 64.5% show substantial memory overhead: 10.4% on average, 33.6% at worst. Such penalty represents a potential barrier for

Table 2.6 Percentage of optimal solutions, average and maximum overhead (Avg. Ovhd. and Max. Ovhd.), Worst-Case Execution Time (WCET) and Average-Case Execution Time (ACET) for each algorithm over the NAS-Bench-101 CNNs.

	gs-best	**gs-first**	**gb-best**	**gb-first**	**b2sp**	**bag-of**	**milp**
Optimal [%]	32.4	35.5	31.1	33.3	34.3	59.4	100
Avg. Ovhd. [%]	10.6	10.4	12.9	13.2	14.1	5.9	0
Max. Ovhd. [%]	33.6	33.6	33.6	33.6	33.6	25.0	0
WCET [s]	0.03	0.03	0.04	0.03	0.11	0.24	28.31
ACET [s]	0.02	0.01	0.02	0.02	0.03	0.10	1.69

deploying CNNs, considering that the memory footprint for intermediate activations increases linearly with the batch size and quadratically with the input resolution. Notice that the worst-case overhead is the same for all the five heuristics (33.6%).

The quality of results can be improved by leveraging the bag-of-heuristics approach (column *bag-of*). In this case, the number of optimal solutions almost doubles (from 35.5% to 59.4%), and the overhead reduces substantially (5.9% on average, 25% at worst). This result confirms that different heuristics fail on different instances of the problem.

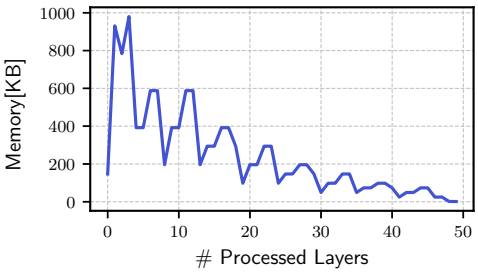
An important outcome concerns the MILP solver (column *milp*): it always converges to an optimal solution. Even though the execution time gets substantially larger than the bag-of-heuristics (up to 118×), the optimal allocation can be found in negligible time (1.69 s on average, 28.31 s in the worst case). As stated in the introduction of the paper, these findings raise the urgent need to revise the dominant approach in standard tensor graph compilers, which instead rely on heuristic schemes that are too weak to deal with the complexity of modern CNNs.

2.2.5 Discussion

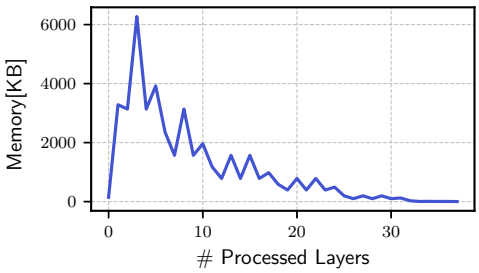
This first part of the chapter presented a thorough review of existing methods for the pool-based memory allocation of static tensor graphs, like CNNs. Our

analysis considered two different problem formulations and the related proposed optimization algorithms. Finally, we conducted a quantitative assessment on six hand-crafted models and 845 AutoML networks extracted from the NAS-Bench-101 database. The experimental results reveal that heuristic methods can achieve optimality only on hand-crafted networks, which presents a limited number of branches and regular connectivity patterns. When tested on networks generated by AutoML, the heuristics fail to identify an optimal solution in most cases ($\geq 65.5\%$), generating substantial memory overhead (up to 33.6%). Their combination in a *bag-of-heuristics* can only limit the number of sub-optimal cases (down to 40.6%). Instead, an exact method based on a MILP formulation guarantees convergence to the optimal allocation for all benchmarks in negligible run time, 28.31 s as the worst-case.

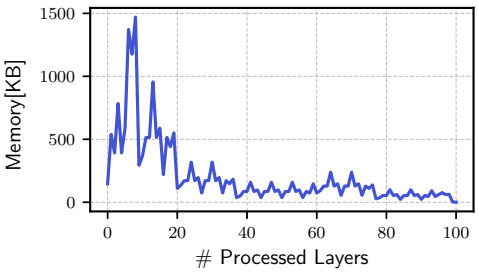
The conclusions achieved through the proposed quantitative assessment hold for a pool-based memory allocation, a type of memory allocation that can be applied under the following assumptions and constraints. First, the network can be described with a static graph, where the shapes of the tensors are static and known and compile time. This hypothesis applies to CNNs but not to other DL algorithms, e.g., Recurrent Neural Networks, where the size of intermediate tensors is input dependent. In such cases, a compiler can either rely on dynamic memory allocation [85] or on allocating memory based on the worst-case scenario. Second, a pool-based memory allocation implies the possibility to place multiple concurrent tensors within the pool through an explicit assignment of their offsets. This feature is not possible for some software libraries, e.g., when dealing with memory allocation in mobile GPUs through OpenGL textures. The OpenGL runtime, in fact, does not allow offset assignment and therefore prevents storing concurrent tensors in the same texture. This constraint forces the creation of multiple pools and therefore calls for other kinds of methods to solve the allocation problem [72]. Third, a pool-based memory allocation is suitable for hardware platforms with a single-level memory hierarchy, or, in the case of multi-level hierarchy, when the software does not have direct control over memory management. Most commercial MCUs and mobile CPUs available in the embedded market [86] fall in these categories of hardware platforms. However, there exists an emerging class of hardware accelerators [87, 88] that requires explicit management of the



ResNet18 - Peak Mem. 980KB.



VGG16. Peak Mem. 6272KB.



MobileNetV2 - Peak Mem. 1470KB.

Fig. 2.6 Memory profile of three different CNNs during the forward pass. Input tensor resolution is 3x224x224 (i.e., ImageNet [8] standard resolution).

different levels of the memory hierarchy. The memory allocation problem raises additional challenges in this context, calling for more complex formulations.

2.3 Dataflow Restructuring for Activation Memory Reduction in CNNs

This section presents our novel data-independent optimization technique aimed at reducing the activation memory footprint of CNNs. The idea behind this work originated from the observation that, for many CNNs, the total active memory reaches its peak value for a limited amount of time, i.e., only when processing a few layers[4], whereas it is remarkably lower before and after. The plots reported in Fig. 2.6 confirm the trend for three popular CNNs. While the specific memory profile of a model depends on its hyper-parameters and topology, at least two elements in the neural architecture design methodology

[4]In this chapter, the terms *layer and* operator are used interchangeably.

make this trend quite general. Regions processing higher resolution tensors require more memory than those processing lower resolution feature maps; regions with more dependencies between layers are more memory demanding than those with a chain-like structure. Thus, in the case of a pool-based memory allocation, the most common choice in DL compilers, a considerable part of the allocated space remains underutilized for most of the inference time. It is therefore natural to ask the following question: *is possible to redistribute the memory peaks over less critical layers to reduce the overall activation memory footprint, getting a more balanced memory profile?* That is precisely the purpose of the dataflow restructuring technique presented in this section. The dataflow restructuring technique first identifies the most memory demanding sub-graphs of the model, the critical subgraphs, and then uses graph-rewriting procedures to transform them into smaller independent branches, reducing the peak memory value without affecting the functionality.

As key features, the proposed technique can work on any CNN, it does not require specialized code routines or hardware modules, and it can be easily combined with other optimizations. In fact, unlike similar related works, e.g., [89], the restructurer operates at a higher level of abstraction, managing tensors as abstract data objects, regardless of how they are specifically packed, stored, and processed on the device. In other words, it does not add dependencies to specific operator-level implementations or hardware modules, ensuring flexibility and portability. An extensive evaluation of the proposed technique on five state-of-the-art CNN architectures, VGG-16, ResNet18, InceptionV3, MobileNetV2, and SqueezeNetV1.1, assesses the memory gain achieved (-62.9% on average) and the computational overhead introduced (8.6% on average).

2.3.1 Background

As discussed more in-depth in the previous section, CNNs can be represented as DFGs where the nodes model the tensor operators and the edges the data dependencies. Each node processes at least one input tensor and produces one output tensor. Fig. 2.7 shows an example DFG, namely, a residual block of ResNet [1]. Note that this work deals with static graphs where all tensors have a static shape defined at compile time, leaving the extension to dynamic graphs for future works.

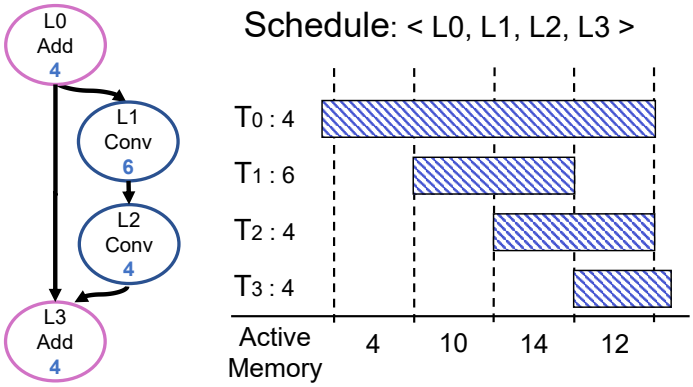


Fig. 2.7 On the left, the DFG of a residual block [1]; on the right, the conflict graph of its intermediate tensors T_i. Each node is labeled with an ID, the tensor operation performed, and the size of the output tensor.

Before performing the memory allocation step, the compiler first schedules the DFG through a topological sort operation; then, it runs a liveness analysis for the intermediate tensors to estimate the size of the memory pool to allocate. The lifetime of a tensor is defined as the difference between the end-time of its latest consumer and the start-time of its producer. Tensors with non-overlapping lifetimes can share the same portion of memory. As shown in Fig. 2.7, the overall size of tensors concurrently alive in a given cycle sets the active working memory, while the cycle with peak memory utilization defines the minimum size of the memory pool (see Sec. 2.2.2). Therefore, reducing the activation footprint requires lowering the peak memory value, which generally depends on the DFG topology and the tensor sizes.

2.3.2 Related Works

While a detailed overview of the optimization strategies for CNNs is reported in sec. 1.2, here we briefly discuss the main differences of the proposed dataflow restructuring algorithm compared to other existing data-independent optimization techniques.

Tab. 2.7 reports a taxonomy of the main related SOTA techniques, highliting the abstraction level, *graph-* or *operator-level*, the main optimization objective, *Memory* or *Latency*, and the target hardware platform, *CPU*, *GPU*, or *Custom* ASIC/FPGA. The taxonomy reflects the hierarchical organization of modern

Table 2.7 Taxonomy of related works.

Works	Graph-Level	Operator-Level	Main Objective	Platform
[49]	x		Memory	Any
[90]	x		Latency	Any
[53, 39, 52]		x	Latency	CPU/GPU
[55]	x	x	Latency	CPU/GPU
[89, 91]	x	x	Memory	Custom
Ours	x		Memory	Any

ML frameworks and compilers [13, 14], which adopt a multi-level structure to lower the high-level dataflow description of a neural model to binary code (see Sec. 1.2).

Graph-level transformations are usually implemented as functionality-preserving rewriting rules, which manipulate the graph by removing, modifying, or adding nodes. They have been used to search for the best parallelization strategy [90] or to reduce the peak activation memory [49] by leveraging the algebraic properties of the operators. For instance, a sequence of a concatenation operator and a convolution can be rewritten as a sum of convolutions [49]. Instead of exploiting the algebraic properties of some tensor operators, the proposed dataflow restructuring algorithm uses the spatial locality of the operators to lower the activation memory footprint, ensuring applicability to a broader range of CNN architectures.

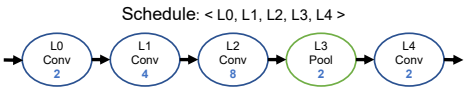
Operator-level transformations deal with the implementation of a single operator, optimizing the scheduling of the low-level code based on the characteristics of the specific target platform. For example, CPUs with multi-level caches and SIMD units require different optimizations than those applied to massively parallel GPUs with scratchpad memories or to custom accelerators with simple PE and an explicitly managed distributed memory hierarchy. These transformations can be operated automatically by the compiler [52], or manually, through a hardware-aware code rewriting process [53, 39].

Operator- and graph-level transformations complement each other and can be combined in cross-level optimizations. One of the most effective techniques in this space is operator fusion, which rearranges the processing of subsequent chained operators in a depth-first manner, mainly to reduce memory bandwidth

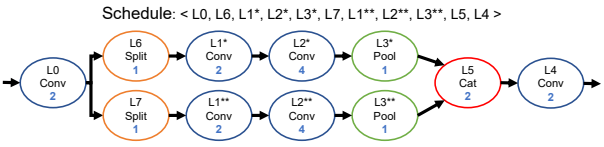
requirements. For instance, a subset of elements of a feature map produced by an intermediate operator can be consumed by a subsequent operator as soon as they are ready instead of waiting for the computation of the entire feature map. Such a schedule enhances the data-reuse of intermediate results, reducing the number of accesses to higher levels of the memory hierarchy. The result of operator fusion is an optimized graph with new nodes representing the custom intra-layer operators. In [55], the authors implemented a framework that first searches the candidate operators to fuse, then generates a fused code implementation targeting high-end CPUs and GPUs. Due to the memory architecture of these platforms, the framework does fuse only a convolutional operator with the following element-wise activation function or pooling operator. The authors of [89] and [91] introduced more aggressive strategies leveraging custom hardware modules with special local buffers to fuse even multiple convolutional operators. Such aggressive fusion enhances on-chip data reuse [89] and increases the achievable troughput [91]. In these works, the graph restructuring is tightly coupled with a specific schedule of the fused tensor operators. In contrast, our dataflow restructuring algorithm works entirely at the graph level, avoiding any dependencies from other operator-layer optimizations, which can be applied later, depending on the specific target platform or on other application constraints.

2.3.3 Restructuring Algorithm

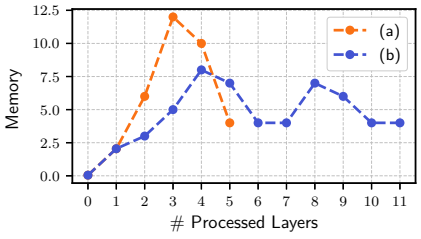
The proposed dataflow restructurer aims to identify and reduce the memory peaks of a CNN starting from a valid schedule of its DFG. This goal is accomplished by isolating those sub-graphs contributing to the peak memory value and applying a localized and memory-driven topology restructuring. Specifically, the new topology preserves the same functionality of the original model, hence the value of all weights of the operators, but it comprises smaller independent sub-graphs that are less memory and computational dense, and that can be processed in sequence, allowing for more memory reuse. Each of these sub-graph computes a tile of the output tensor by processing a tile of the original input tensor(s); the input tiles are obtained by adding proper *split operators* to the original DFG, whereas the output tiles are concatenated (thanks to newly inserted *cat operators*) to reconnect the new sub-graphs with



Toy network DFG before the restructuring process.



Toy network DFG after the restructruing process.



Memory profiles.

Fig. 2.8 Example of the dataflow restructuring process on a linear DFG.

the other nodes of the model. The split and cat operators can be seen as special nodes that create lightweight and independent processing paths between the model regions with a lower memory pressure.

Before detailing the algorithmic implementation, Fig. 2.8 reports an example of the optimization process, showing the DFG of a simple linear model before (a) and after (b) the topology restructuring and their memory profiles (c). Operator L2 dictates the original peak memory value (12 units in the example). Thus, reducing the peak memory requires lowering the volume of tensors concurrently processed by L2. Indeed, the memory-driven topology restructuring splits L2 into smaller nodes and propagates the transformation backward to L1 and forward to L3. The resulting DFG has two lightweights (in terms of activation memory) and independent branches: from L6 and L7, which are the inserted split operators in charge of tiling the tensor produced by L0 such that the functionality of the sub-graph is preserved, to L5, which concatenates the output tiles produced by L3* and L3**. The two branches are copies of the original operators that work on tensors halved in size. As the computational complexity depends on the size of the tensors, the two branches are almost two times faster than the initial monolithic path (except for a small overhead discussed later in this section), so the overall latency remains almost the same. Even more important, they are independent: they consume and produce disjoint tensors and can be therefore processed sequentially to reduce lifetime conflicts, maximizing memory reuse. The plot reveals that the obtained memory profile shows a lower peak value and, thus, a smaller memory footprint (33% savings).

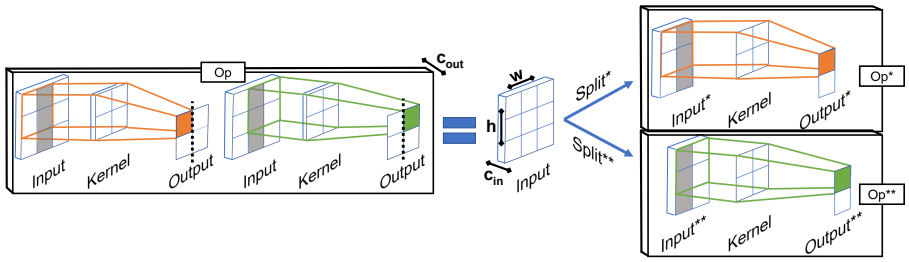


Fig. 2.9 The split of a stencil operator into two smaller independent operators.

In order to make the restructuring procedure generic and automated, the placement of the fork-points, i.e., the split operators (L6 and L7), and of the join-point(s), i.e., the concatenation operator (L5), is an essential aspect to consider. Fork- and join-points define, in fact, the critical region to be restructured, or, equivalently, the graph regions with a lower memory pressure (L0 and L4 in the example) to bridge with the new independent processing paths. The wrong placement of such anchor points would make the restructuring process ineffective. For instance, splitting L2 alone does not bring any savings, as the total size of the overlapping tensors would be the same as the original DFG. Moreover, fork- and join-points create further dependencies with neighboring tensors, affecting the overall working memory. The input tiles produced by the split operators and the output tiles processed by the concatenation operator must be kept alive when processing each independent branch. Last but not least, the tensor operators mainly used in CNNs, such as convolution and pooling, are stencils with overlapping windows that lead to redundant computations (the *halo* regions) and duplicated elements when creating the independent branches. An example is shown in Fig. 2.9 for a stencil operation with a kernel window of size 2×2. The output tensor is split vertically, generating two smaller operators; each operates on two tiles of the original input tensor, which have duplicated values (the grey-shaded elements in the picture) to preserve the functional equivalence of the new sub-graphs. When the split operator is back-propagated across a path of the graph, those duplicated elements are computed several times by different operators, resulting in additional computations. Playing with the fork- and join-points placement and the number of independent branches affect the length and the data volume of the independent branches, hence the number of redundant operations.

Based on these considerations, we tackled this multi-objective optimization problem (memory vs. computational overhead) by providing a simple heuristic, the restructuring algorithm. The restructuring algorithm can serve as an engine for different greedy optimization strategies (one of which will be discussed in the experimental section), as formulating the problem in a closed-form might be unfeasible, especially in the case of large networks.

Pseudocode

Algorithm 3 reports the pseudocode of the proposed restructuring algorithm, named RESTRUCTURE, with two input parameters: α and n_slices. α is a scalar value affecting the selection of the sub-graph to be restructured, while n_slices is a pair $\{w, h\}$ indicating the number of splits applied in the width and height dimensions of the tensors.

At first, the EXPLORE procedure finds the set of nodes to include in the critical sub-graph (line 2), i.e., those to be restructured. This procedure determines the placement of the fork- and join-points discussed in the previous sub-section. Then, the nodes are projected on the DFG, and the critical sub-graph is identified (line 3). Finally, the sub-graph is visited in reverse topological order to perform the graph-rewriting (lines 4). Specifically, the output tensors of the sub-graph are split according to the values in n_slices; then, during the backward traversal, the split operators are propagated from the output to the input tensors of each node applying the graph-transformation shown in Fig. 2.10.

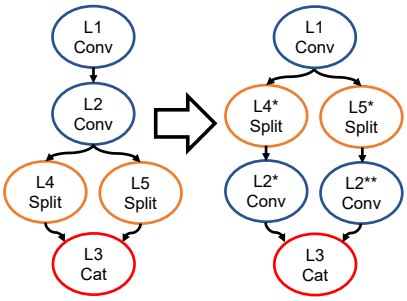


Fig. 2.10 Graph-rewriting that propagates the split operators in the DFG.

To compute the parameters of the split operators automatically, we developed a symbolic evaluation engine on top of a domain-specific language (DSL) that associates a functional specification to each tensor operator.

The EXPLORE procedure works as follows. First (lines 8-9), the DFG is scheduled, and the lifetime of each tensor is extracted to compute the memory

Algorithm 3: Daflow Restructuring Algorithm

```
1  Function restructure(DFG, α, n_slices):
2      critical_set = explore (DFG, α)
3      subDFG = extractSubgraph (DFG, critical_set)
4      reverseVisit (subDFG, n_slices)
5  Function explore(DFG, α):
6      schedule = topologicalSort (DFG)
7      lifetimes = getLifetimes (DFG, schedule)
8      memory_profile = getMemProfile (schedule)
9      peak_memory = max (memory_profile)
10     for node ∈ DFG.nodes do
11         criticality[node] = computeCriticality (
12                                 lifetimes, node, memory_profile)
13     end
14     critical_set = {node ∈ DFG.nodes,
15                         s.t. criticality[node] == peak_memory}
16     while ∃ node ∈ fanin(frontier(DFG, critical_set))
17                 s.t. criticality[node] ≥ α · peak_memory do
18         critical_set ← critical_set ∪ {node}
19     end
20     while ∃ node ∈ fanout(frontier(DFG, critical_set))
21                 s.t. criticality[node] ≥ α · peak_memory do
22         critical_set = critical_set ∪ {node}
23     end
24 return critical_set
```

profile and the peak value (lines 10-11). Each node is then labeled with a criticality index equal to the maximum amount of memory active during its processing (lines 12-15). All the nodes with a criticality index equal to the peak memory constitute the initial critical set (lines 16-17). The critical set is then extended using α as the driving parameter. Specifically, nodes at the boundary of the frontier — those critical nodes with in-/out-degree edges from/to nodes outside the critical set — can join the critical set if their criticality is greater or equal to $\alpha \cdot peak\_value$ (lines 18-25). The expansion ends if no other nodes can join the critical set, that is, when the remaining nodes have a criticality lower than $\alpha \cdot peak\_value$. Therefore, α directly affects the placement of the fork- and join-points, and hence the efficiency of the restructuring.

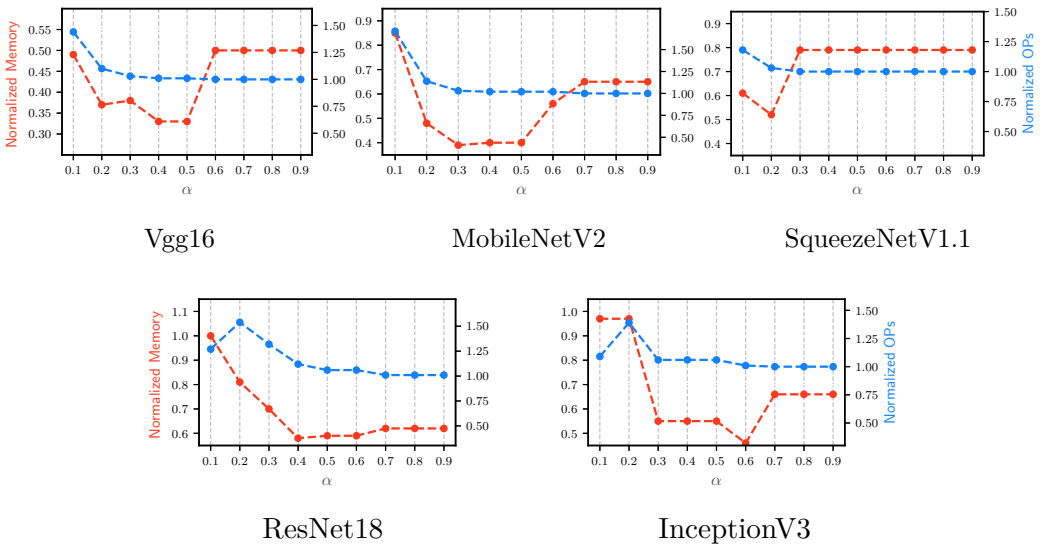


Fig. 2.11 Normalized memory and number of operations for $\alpha \in [0.1, 0.9]$ and n_slices=$\{2h, 2w\}$.

2.3.4 Experimental Results

We tested the proposed optimization on five state-of-the-art CNN architectures. The goal is to quantify the savings and prove the efficacy on models with different topologies. The selected benchmarks comprise VGG-16 as representative of large and computationally intensive networks with conventional single-layer connectivity; ResNet18 and InceptionV3 as representative models with irregular connectivity; MobileNetV2 and SqueezeNetV1.1 as representative models of small networks targeting mobile platforms. We provide a parametric analysis taking into consideration the two main parameters of the restructuring algorithm (Algoritm 3): α in the range $[0.1, 0.9]$ - step 0.1; n_slices in $\{[2, 4] \times [2, 4]\}$ - step 1 - along the two spatial directions height (h) and width (w), for a total of 9 permutations.

Fig. 2.11 reports the first set of collected results. The plots show the normalized peak memory (in red) and the normalized number of operations (in blue) as functions of α with n_slices set to a default value of $\{2h, 2w\}$. As a general trend, the memory usage gets smaller with α until it reaches a global minimum, indicated by α_{opt}. Tab. 2.8 collects a summary of the minimum points, reporting the memory savings achieved and the corresponding computational overhead. Memory savings reach on average 54.3%, with an

Table 2.8 Memory saving and computational overhead for $\alpha = \alpha_{opt}$ and $n\_slices = \{2h, 2w\}$.

Network	α_{opt}	**Memory Savings [%]**	**Computational Overhead [%]**
VGG-16	0.4	67.5	1.1
MobileNetV2	0.3	60.5	3.0
SqueezeNetV1.1	0.2	48.4	3.1
ResNet18	0.4	41.6	11.9
InceptionV3	0.6	53.5	1.4
Average		54.3	4.1

overhead of 4.1%. The technique performs best on VGG-16 (67.5% of savings, 1.1% overhead) thanks to a linear topology with no reconvergent paths. The memory reduction and computational overhead get slightly worse on ResNet18 (41.6% of savings, 11.9% overhead) due to residual blocks that complicate the topology, generating more overlap between the branches created during the restructuring process. In all networks but SqueezeNetV1.1, the new topologies obtained with α=0.9 already achieve a significant gain (60% on average). That is due to a highly irregular memory profile with peaks that fall steeply. However, larger memory savings require a proper selection of the critical sub-graph, especially for networks with a complex topology. In fact, the min-max distance of the memory savings is relatively high ($> 30\%$) for all the CNNs under analysis. This observation further motivates the importance of an optimal search.

The point α_{opt} is the break-even point under which the critical sub-graph (i) has a light frontier that enables enough data reuse, and (ii) is wide enough to catch several peaks of the model without resulting in a large number of additional operations. For α greater than α_{opt}, the critical sub-graph may get too small, and this may affect the memory savings negatively for two reasons. First, being the original global peak suppressed by the restructuring procedure, other local peaks outside the critical region of interest may now emerge as the new global ones. Second, the increase in the lifetime for large tensors at the fork- and join-points could overcome the benefits of lighter branches. For α lower than α_{opt}, the critical sub-graph gets larger than required, covering too many operators and thereby enlarging the chain of backward rewritings

Table 2.9 Computational overhead and memory saving for the optimal setting of *n_slice* when $\alpha = \alpha_{opt}$.

Network	α_{opt}	**n_slices**	**Memory Savings [%]**	**Computational Overhead [%]**
VGG-16	0.4	2h, 4w	75.0	2.3
MobileNetV2	0.3	3h, 4w	77.3	7.8
SqueezeNetV1.1	0.2	2h, 2w	48.4	3.1
ResNet18	0.4	3h, 3w	48.8	25.7
InceptionV3	0.6	3h, 3w	64.9	3.9
Average			62.9	8.6

substantially. This long chain is a source of redundancy which translates into higher memory demand and more arithmetic operations.

Tab. 2.9 completes the parametric analysis bringing the parameter *n_slices* into play. Specifically, it shows the optimal setting for *n_slices* when $\alpha = \alpha_{opt}$. A comparison between Tab.s 2.9 and 2.8 demonstrates that freeing the values in *n_slices* increases the memory savings (62.9% vs. 54.3% on average) at the cost of some computational penalty (8.6% vs. 4.1% on average). As a general trend, with more slices, the memory consumed by each branch gets smaller at the cost of redundant computations. However, the memory-vs-compute trade-off is more complex. If, after the graph restructuring, the new peak memory falls outside the critical sub-graph, increasing the values in *n_slices* does introduce more computational overhead without further savings. On the contrary, if the critical-sub graph still contains the peak dictating the total memory, then the effectiveness of varying *n_slices* over α is highly biased by the topology. How to infer the optimal setting is an open issue. However, this does not represent a major impediment since exploring various settings is fast and efficient. The sweep of the restructuring algorithm over nine α values completes in $82s$ for SqueezeNetV1.1 (fastest) and $375s$ for InceptionV3 (slowest), considering our single-core implementation on a machine powered by an Intel i7-8700K.

As a final remark, we further emphasize the orthogonality of the proposed graph-level restructuring to other optimizations. Rule-based approaches that leverage hand-written optimizations based on specific CNN structures are still

equally effective as on the initial graph and can be applied over the newly created branches. At the same time, it is still possible to exploit either automatic code synthesis methods [52] or vendor-specific libraries [53, 39], freeing the user from the considerable burden of developing additional operator-level optimizations for all possible target platforms.

2.3.5 Discussion

This second part of the chapter introduced a functionality-preserving optimization technique to lower the memory footprint of the feature maps of a Convolutional Neural Network. The technique operates a graph restructuring process that identifies the regions of the model contributing to the peak memory consumption and rewrites them as a series of more lightweight and independent sub-graphs. An extensive experimental assessment shows broad applicability of the proposed method on different CNN architectures, from linear topologies to more complex ones with residual branches, reporting remarkable memory savings (62.9% on average) with minimal computational overhead (8.6% on average). As it will be shown in the next section, the proposed restructuring process combined with other graph- and operator-level transformations can lead to more efficient tensor graph processing on embedded devices.

2.4 Sparse-Tiled Tensor Graph Processing

When targeting an MCU device, modern ML compilers [92] store the intermediate feature maps produced at run time in the on-chip SRAM, whereas the weights in the non-volatile memory, usually a FLASH memory. Weights can be, then, accessed at run time directly from the FLASH, as many MCUs have a fast direct access path to the FLASH [62]. Thus, when considering which memory optimization technique to apply, one important discriminant factor is the specific target memory and its effect on the accuracy of the model. For instance, on the memory side, weights pruning [33, 64] generally affects the FLASH memory footprint only, input resolution scaling [15] the SRAM, whereas quantization [40] and topology scaling [15] both the FLASH and the SRAM. On the accuracy side, quantization and pruning are fine-grain knobs

that allow the accuracy to be almost fully recovered with iterative re-training stages. In contrast, resolution and topology scaling are coarser knobs for fast and aggressive memory compression that usually induce non-recoverable loss.

This section presents a novel memory optimization pipeline that combines two techniques belonging to two different classes of methods: data-driven and data-independent. We adopt fine-grain weight pruning to reduce the weight footprint, i.e., FLASH occupation, with minimal accuracy loss, while we use the functionality-preserving dataflow restructuring procedure described in the previous section to reduce the activation footprint, i.e., SRAM occupation. The resulting graph obtained by such a pipeline is referred to as a *sparse-tiled tensor graph*.

One key feature of the proposed pipeline is that it integrates a data-independent technique, namely, the dataflow restructuring, early in the design and optimization flow. Such a choice differs from common methodologies [92] where data-independent strategies aimed at optimizing the resource allocations are applied too late in the pipeline, only after quantization, pruning, and topology scaling already set the trade-off between accuracy and memory consumption. Intuitively, this approach causes some regions of the optimization space to remain unexplored. Indeed, a thorough assessment of the overall pipeline on two versions (v1 and v2) of MobileNets shows that sparse-tiled graphs are dominant in the memory-accuracy space when targeting the NUCLEO-F767ZI board [62] hosting an ARM Cortex-M7 with 512KB/2MB of SRAM/FLASH. Compared to other compression pipelines that comprise either only data-driven techniques, i.e., sparsity, resolution, and topology scaling, or only data-independent techniques, i.e., dataflow restructuring, the proposed strategy enables the deployment of new optimal CNN configurations on MCU-powered devices.

2.4.1 Optimization Pipeline

An overview of the optimization pipeline is shown in Fig. 2.12. In this work, the pipeline targets MCUs of the ARM Cortex-M family; however, it can be easily generalized to other hardware architectures. The pipeline's input is a pre-trained CNN, and the output is a binary file ready to be deployed on the target device. The pipeline comprises two main stages. The *front-end* is where the

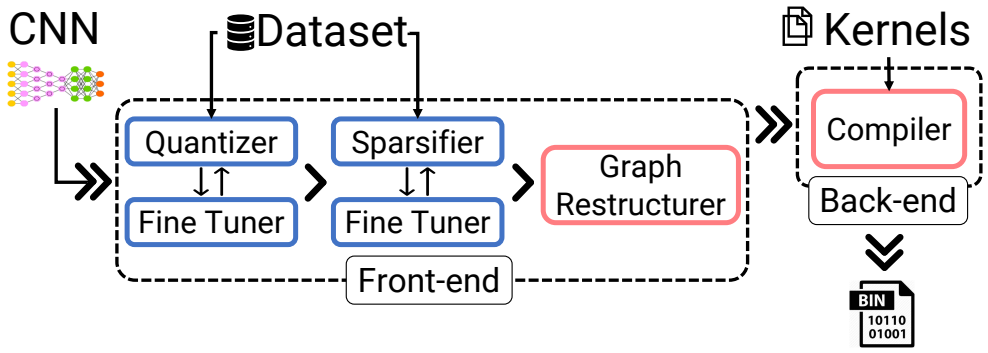


Fig. 2.12 The proposed optimization pipeline. The blue boxes indicate data-driven passes, while the red boxes data-independent passes.

memory optimization happens; the *back-end* is where the high-level description of the tensor graph is lowered to a low-level description, which can then be compiled to produce a binary executable file. The *front-end* includes two data-driven passes, pruning and quantization, and the data-independent dataflow restructuring procedure. The *back-end* lowers the high-level description of the obtained tensor graph by mapping each node to a low-level routine extracted from a library of operators optimized for the target device. The following subsections provide details on the data-dependent passes of the *front-end* stage as the dataflow restructuring algorithm was described in detail in the previous section (Sec. 2.3).

Data-Driven Passes

The tensor graph is gradually sparsified following the state-of-the-art structured pruning procedure proposed in [33, 34]. Convolutional layers are usually processed using a matrix-matrix multiplication operator (GEMM) thanks to the im2col transformation [39]. Thus, sparse convolutional layers are implemented by substituting the GEMM operator with a sparse matrix multiplication routine (spMM). As the Cortex-M datapath has a 2-lane SIMD unit, the weight matrix is sparsified in groups of two to efficiently vectorize the operation and reduce the indexing overhead. Specifically, the sparse weight matrices are encoded using a block Compressed Sparse Row format (CSR) as proposed in [33]. In particular, each sparse matrix is represented with a set of three 1D arrays: the *nnz-values* storing the value of the non-zero weights, the *j-idx* storing the first column index of each couple of non-zero elements, and the *i-idx* storing the number of

non-zero elements in each row. The *nnz-values* and *j-idx* arrays are accessed with a sequential streaming pattern, ensuring good cache locality. Moreover, by properly unrolling the innermost loop of the spMM operator, vector load instructions are used for accessing the *nnz-values* array, fully exploiting the available bandwidth to the L1 data cache. Storing the non-zero values and the metadata associated with the CSR format makes the memory footprint of the model approximately one and a half times the size of the non-zero elements. Note that this pruning strategy does not reduce the activation footprint, and hence it only affects the FLASH occupation of the model. The amount of sparsification enforced in the procedure may lead to accuracy loss; however, an 80% of sparsity represents a safe value, ensuring enough room to almost recover the accuracy of the dense model with a fine-tuning procedure [33, 34].

The quantizer reduces the arithmetic precision of weights and activations to 8-bits, using a linear symmetric scheme with power-of-two scaling [40, 39]. Such a design choice enables an efficient SIMD implementation of the tensor operators. While all the layers share the same bitwidth, the radix-points of activations, weights, and biases are assigned layer-by-layer. For the intermediate activations, the radix-points are set by minimizing the mean squared error between the original floating-point value distribution and the quantized one on a calibration set, namely, a subset of the training set (more on this in Sec. 3.3.3). A fine-tuner based on knowledge distillation is used to recover a possible accuracy loss [93]. The effect of the quantization step is a 4x reduction in both activation and model footprint.

2.4.2 Experimental Results

Benchmarks and Deployment

We tested the proposed optimization pipeline on MobileNetV1 [15] and MobileNetV2 [84], two state-of-the-art CNNs explicitly designed for mobile platforms. A set of pre-trained models is publicly available, including different configurations obtained through the width multiplier α and the input resolution ρ, the two architectural knobs proposed to scale the model (see Sec. 1.2). Tab. 2.10 provides a summary of the main features for $\alpha \in \{0.50, 0.75, 1.00\}$ and $\rho \in \{160, 192\}$; all the models are quantized to 8-bit with a layer-wise

Table 2.10 Baseline Characterization. Accuracy on ImageNet taken from tensorflow repositories[6].

Network	α	**Parameter count [M]**	ρ	**Top-1 Accuracy [%]**
MobileNetV1	1.0	4.24	192	69.2
			160	67.2
	0.75	2.59	192	66.1
			160	62.3
	0.50	1.34	192	60.0
			160	57.7
MobileNetV2	1.0	3.47	192	70.7
			160	68.8
	0.75	2.61	192	68.7
			160	66.4
	0.50	1.95	192	63.9
			160	61.0

binary scaling. Although other configurations are available, e.g., $\alpha \leq 0.25$ and ρ=$\{128, 224\}$, we decided not to report them because of their sub-optimality. For instance, ρ=224 takes +50% SRAM improving the accuracy by a mere 1% (w.r.t. ρ=192), while with $\alpha = 0.35$ ($\alpha = 0.25$) accuracy gets below <60% (50%). It is also worth emphasizing that our optimization pipeline does apply to other CNNs and tasks achieving similar results, not reported in this manuscript for the sake of space.

The tests were conducted on a NUCLEO-F767ZI board [62] hosting 512KB of on-chip SRAM and 2MB of FLASH. The MCU is an ARM Cortex-M7, operating frequency 216 MHz. We extended the CMSIS-NN library v.5.6.0 [39] with a modified in-house version of the SIMD-aware sparse matrix multiplication kernels presented in [33][5]. The graph tiling procedure does not require further modifications of the kernel library. It is a model- and hardware-agnostic graph-level method that works for any existing neural library [53, 39] and it does not prevent other low-level automatic code optimization, e.g. [52]. We adopted the GNU Arm Embedded Toolchain (version 6.3.1) for cross-compilation.

[5]No official open-source implementation was available at the time of writing.

[6]github.com/tensorflow/models/blob/master/research/slim/nets/mobilenet.md github.com/tensorflow/models/blob/master/research/slim/nets/mobilenet_v1.md

Pipeline Set-up

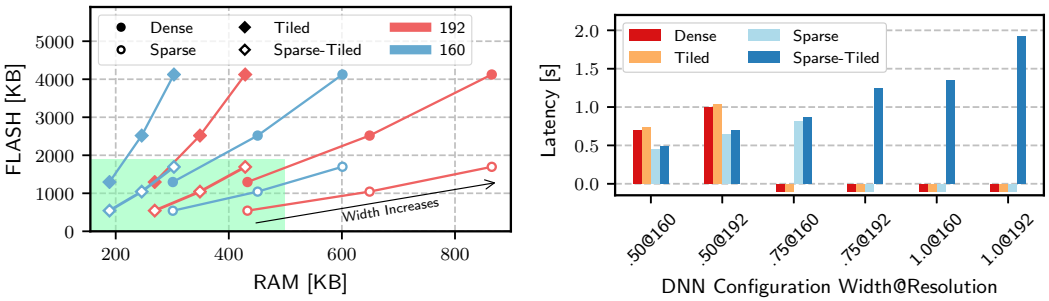
The sparse networks are obtained following the method proposed in [33] with 80% of sparsity distributed across all convolutional layers, except those containing depthwise convolutions that have been implemented with the dedicated dense operator available within the CMSIS-NN library. The choice of the sparsity level is driven by the empirical analysis provided in [33, 34], which demonstrates that 80% is a safe sparsity value, preserving most of the accuracy. Concerning the graph restructuring procedure, the threshold is set to $0.4 \cdot peak\_value$; namely, the restructuring region covers all the fan-in/out layers with an active memory $\leq 0.4 \cdot peak\_value$. Each tensor is split into four equally sized parts (2 slices along the width and height dimensions of the convolutional filters), originating four parallel branches for each tensor path.

Experimental Results

This section analyzes two main extra-functional metrics: memory (SRAM and FLASH) and latency.

Fig. 2.13a shows the RAM and FLASH consumed by MobileNetV1. The shaded area in the plot outlines the feasible region: configurations within this region meet both RAM and FLASH constraints and can be ported on the target hardware[7]. Sparse and tiled networks ($\diamond$) are those that meet both SRAM and FLASH constraints at full scale; hence, they can be deployed at full accuracy offering the highest quality with the minimum memory usage. The same is not for configurations obtained with the other knobs. Among the dense networks ($\bullet$), only two (.50@160 and .50@192) meet the memory budget at the cost of a large penalty in terms of accuracy (Tab. 2.10). Sparse networks ($\circ$) show a lower FLASH footprint thanks to the sparse matrix format; this lets a new configuration join the feasibility region (.75@160), while the remaining ones violate the SRAM constraint. The tiled networks ($\blacklozenge$) obtained with graph restructuring technique (w/o sparsification) show $\approx$50% lower SRAM usage and are compliant with the constraint. However, they exceed the available FLASH in most cases, which calls for aggressive width modulation

[7]Due to additional overhead to run the network, the flash constraint is set to 1.9MB and the RAM constraint to 500KB

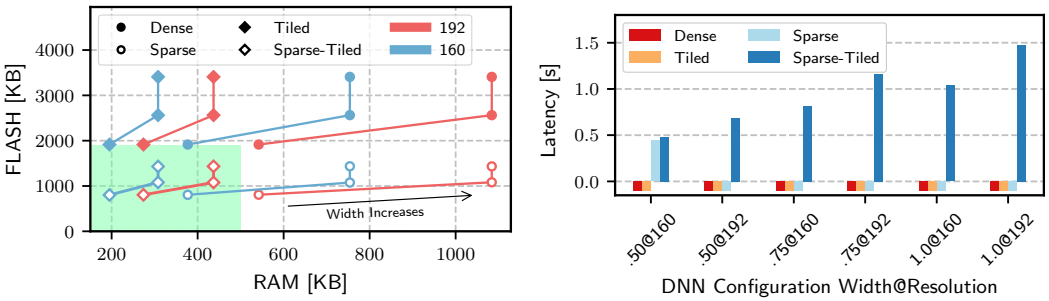


a) b)

Fig. 2.13 a) FLASH and RAM requirements for different configurations of MobileNetV1. Width values are 0.50, 0.75, 1.00. b) Latency measurements for MobileNetV1. Bars sorted for accuracy, from least accurate (left) to most accurate (right).

($\alpha \leq .50$). The bar chart in Fig. 2.13b shows the latency measured on the device; the negative bars are used to indicate the models not fitting the memory space. Sparse-tiled networks get much faster than dense only and tiled only nets; this can be observed for the first two networks on the left side of the chart ($\{.50@160, .50@192\}$). Moreover, they can be processed at any width ($\{.75@192, 1.0@160, 1.0@192\}$), ensuring the highest quality, which is a good option for non-time-critical tasks. Obviously, latency increases proportionally. As explained in Section 2.4.1, the restructuring procedure adds overhead due to redundant computations: ≈10% penalty compared to dense and sparse (as shown for the two configurations $\{.50@160, .50@192\}$). The three sparse configurations ($\{.50@160, .50@192, .75@160\}$) dominate the sparse-tiled version, which, however, could still be adopted in case additional RAM is occupied by concurrent applications running on the MCU, e.g., a sensor sampling procedure.

Similar conclusions can be inferred for MobileNetV2, whose results are reported in Fig. 2.14a and 2.14b showing even greater benefits. The sparse-tiled configurations are those that fit into memory. Unlike MobileNetV1, none of the dense networks is compliant with the memory constraints, mainly due to the larger activations. Aggressive scaling factors, both for the width multiplier and the input resolution, are needed to push one configuration within the feasible region (.50@160), resulting in a high accuracy drop. Tensor tiling is mandatory to bring SRAM below the threshold, but still not enough due to the model size that gets close to the boundaries of the feasible region for highly scaled widths.



a) b)

Fig. 2.14 a) FLASH and RAM requirements for different configurations of MobileNetV2. Width values are 0.50, 0.75, 1.00. b) Latency measurements for MobileNetV2. Bars sorted for accuracy, from least accurate (left) to most accurate (right).

A side comment raises from the observation that shrinking α from 1.0 to .75 does not reflect a reduction of the activations. The main reason is that the first layers of the 1.0 and the .75 configuration have the same hyper-parameters; hence, since the activation memory peak is achieved while processing the first layers of the network, they share the same activation footprint. In terms of latency, we observed the same trend reported for MobileNetV1, except for the non-monotonic behavior recorded for one configuration (1.0@160), where the speed-up by sparsity accumulates in a specific way due to the topology of the network and the width-resolution ratio.

2.4.3 Discussion

This work introduced a memory optimization pipeline combining data-driven DL techniques with a data-independent dataflow restructuring optimization. The weights of the model are block-pruned by a sparsification procedure, while the subgraphs of the model contributing to the peak memory value are substituted with functional-equivalent but smaller independent sub-graphs. The overall effect is a reduction of both RAM and FLASH footprint. The collected results show the effectiveness of the proposed pipeline, as several configurations of MobileNets can be deployed on an ARM M7 core with 512KB/2MB of SRAM/FLASH. We expect further investigations of joint data-driven and data-

independent optimizations to enable more efficient tensor graph computing, meeting the needs of embedded intelligent applications.

2.5 Conclusions

This chapter focused on how to build *small* CNNs. Specifically, it first reviewed and quantitatively assessed on a large benchmark suite the main algorithms adopted to tackle the problem of memory allocation in DL compilers. As the main outcome, it demonstrated the need to revisit the dominant trends, as the commonly adopted heuristics can introduce a substantial memory penalty, whereas a MILP method, which was disregarded by previous works due to its complexity, can always find the optimal solution in a reasonable run time. The second part of the chapter introduced a novel functionality-preserving graph restructuring procedure to reduce the memory footprint of the intermediate activations of a CNN. The technique achieves remarkable memory savings at the cost of small computational overhead when tested on a wide variety of CNN architectures. As a key feature, the proposed approach can work on any CNNs without specialized code or hardware components, resulting in a widely applicable and highly effective technique. The last part of the chapter presented a new compression pipeline, which combines weight pruning with graph restructuring to deploy more accurate CNNs on tiny MCU devices. The experimental results revealed that the combination of graph- and operator-level transformations could push further the efficiency of tensor graph computing on small embedded devices.

Chapter 3

Enabling Monocular Depth Estimation on Low Power Devices

3.1 Introduction and Motivation

Embedded applications are usually characterized by a performance constraint, which corresponds to a minimum throughput, and by a set of hardware constraints, dictated mainly by cost and power consumption limitations. For this reason, an end-to-end evaluation of possible optimizations is crucial to assess different design choices and find the most suitable trade-off. Thus, in this chapter, we focus on a specific application use case to highlight the importance of the vertical optimization approach proposed in this thesis. Specifically, the framework presented in this chapter is assessed in the context of monocular depth estimation, one of the most important components of embedded computer vision applications.

Depth perception is a low-level vision kernel adopted in many real-world applications, like autonomous or assisted driving and robotics. While effective active technologies, such as Time-of-Flight (ToF) and Light Detection and Ranging (LiDAR), can be used for depth estimation, they are expensive, significantly sized, and only provide sparse depth measurements. A more attractive method involves inferring depth from images acquired using conventional image

sensors, namely, integrated cameras, which are usually already available in many embedded vision systems. For this reason, depth estimation from images represents an important problem in computer vision, and various approaches have been extensively investigated over time [94].

Estimating depth from two calibrated cameras, namely, a stereo acquisition pipeline, has recently achieved remarkable improvements thanks to the adoption of CNNs. However, since energy consumption, form-factor, and assembling cost severely constrain the design space of an embedded system, there is an increasing interest in using a more ubiquitous and cheaper setup, where dense depth predictions are inferred from a *single* image. An acquisition setup requiring only one camera reduces implementation cost, improves portability, and minimizes energy consumption, enabling the deployment of emerging applications, like augmented reality and virtual reality (AR/VR), on portable devices and intelligent automation systems.

Despite the recent improvements in monocular techniques based on CNNs, the problem of estimating depth from a single image is theoretically ill-posed, requiring a huge computational power to bridge the accuracy gap with geometry-aware methods like stereo [94]. The CNNs used for monocular depth estimation are, in fact, resource-demanding, both in terms of computational complexity and memory footprint, leading the authors of previous works [95] to rely on power-hungry accelerators like high-end GPGPU cards. Thus, deploying CNNs for monocular depth estimation on embedded platforms while preserving accuracy and performance represents an open research question.

To this end, this chapter briefly reviews the literature related to monocular depth estimation (Section 3.2). Then, it presents design methods and optimization techniques for implementing a monocular depth estimation pipeline on a low-cost CPU powered by an ARMv7a architecture (Section 3.3). Finally, it introduces the novel lightweight CNN architecture, referred to as μPyD-Net, as well as the algorithmic-and operator-level optimization strategies we proposed to push depth estimation within the constraints of the MCUs adopted as end-nodes of the IoT (Section 3.4).

The content of this chapter is an extended and improved version of our previous publications found in [96–99].

3.2 Deep Learning for Depth Estimation

Early methods have relied on the geometrical (*epipolar*) constraint to estimate the depth from two images acquired by two aligned and synchronized cameras. With this setup, the depth can be inferred by mean of *triangulation* once the correspondence between pixels in the two views is addressed. Due to the importance of depth estimation in computer vision, many algorithms have been then proposed in the literature [100–102], and, more recently, DL methods have achieved better results compared to classical ones [103, 104],

Unfortunately, the adoption of depth-from-stereo may be limited by the requirements of two calibrated cameras and synchronized image acquisition, which could be not available (e.g., on existing installations), too expensive, or cumbersome. Moreover, stereo vision also suffers from unreliability issues in the case of miscalibration or when large dis-occlusions occur between the two cameras. Therefore, inferring depth using only a single camera has recently gained more interest. Despite monocular depth estimation being theoretically a strongly *ill-posed* problem, in the last few years, CNNs have been able to learn how to exploit clues such as shadows, occlusions, and relative scales between objects to estimate the depth from a single image.

Early approaches for monocular depth estimation powered by CNNs have used ground truth depth supervision [105, 106], achieving unmatched accuracy compared to previous works in the field [107, 108]. Unfortunately, collecting large amounts of images with depth annotations is extremely expensive, and it requires additional sensors and hand-made post-processing. To overcome the need for ground truth data, CNNs can be trained by casting depth estimation as an image reprojection across different viewpoints [109, 95], thus in a self-supervised manner. To this end, there are two prominent (not mutually exclusive) strategies usually adopted to obtain different viewpoints of the same scene: taking multiple images acquired with a single but moving camera [109–111] or using a stereo camera [112, 95, 113, 114]. Godard et al. [95] achieved better results than previous works combining the (sub-)differentiable bilinear sampler mechanism proposed in [115] with a left-right consistency constraint, a better neural architecture design, and a more robust appearance matching loss function [116]. Concurrently, [113] and [117] improved the results achieved by stereo supervision using, respectively, a novel trinocular paradigm and adver-

Table 3.1 Evaluation metrics. y denotes the predicted depth, y^* the ground-truth depth. N represents the amount of valid pixels in the ground-truth depth map.

Notation	**Definition**	**Equation**
Abs Rel	Absolute Relative Error	$\frac{1}{N}\sum_{i=1}^{N}\frac{\lvert y_i - y_i^*\rvert}{y_i^*}$
Sq Rel	Squared Relative Error	$\frac{1}{N}\sum_{i=1}^{N}\frac{\lVert y_i - y_i^*\rVert^2}{y_i^*}$
RMSE	Root Mean Squared Error	$\sqrt{\frac{1}{N}\sum_{i=1}^{N}\lVert y_i - y_i^*\rVert^2}$
RMSE log	Logarithmic Root Mean Squared Error	$\sqrt{\frac{1}{N}\sum_{i=1}^{N}\lVert \log y_i - \log y_i^*\rVert^2}$
a_n	Prediction Accuracy	% of y_i *s.t.* $\texttt{max}\left(\frac{y_i}{y_i^*}, \frac{y_i^*}{y_i}\right) < 1.25^n$, $n \in \{1, 2, 3\}$

sarial loss during training. Other strategies based on stereo pairs have relied on semi-supervised training data [118] or enforcement of temporal consistency [119]. Finally, a more unconstrained strategy used to infer depth from a single image consists in using unlabeled monocular videos for training [111, 110].

Unfortunately, despite their effectiveness, the CNN architectures used in these works are pretty complex, they require high computing capabilities (e.g., those available on high-end GPGPUs), and they occupy a large memory footprint during the inference phase. In order to reduce compute and memory requirements, FastDepth [120] and EDA [121] proposed novel compact architectures tailored for mobile GPUs, like the NVIDIA Jetson TX2 board powered by a mobile version of the Pascal architecture. However, both methods rely on a standard supervised training procedure, which, as stated before, requires a full annotated dataset rarely available in real-life applications.

This chapter targets the deployment of *self-supervised* monocular depth estimation on *low-power CPU-based* devices (Section 3.3) and on *MCU-based* devices (Section 3.4).

Evaluation

For the sake of clarity, this subsection reports the definition of the standard metrics adopted to evaluate the accuracy of depth estimation methods [105]. The Absolute Relative Error (Abs Rel), the Squared Relative Error (Sq Rel),

the Root Mean Squared Error (RMSE), and the Logarithmic Root Mean Squared Error (RMSE log) are commonly used to compute the error (the lower, the better) between the predicted and ground-truth depth. Additionally, an accuracy score (the higher, the better) α_n is adopted to compare different depth estimations methods. α_n is defined as the percentage of predicted depth values whose ratio and inverse ratio with the ground truth is below a given threshold of 1.25^n. We summarized the definition of each metric in Tab. 3.1.

3.3 Enabling Depth Estimation on ARMv7a CPUs

This section presents our end-to-end framework aimed at enabling monocular depth estimation on ARMv7a platforms. Two key factors are needed to make the high-end GPGPU to low-power CPU shift successful for CNNs targeting monocular depth estimation. First, a network topology capable of achieving high accuracy while complying with the computational and memory resources of an off-the-shelf low-power CPU. Second, the availability of a vertical optimization stack for code optimization and deployment to real hardware. To this end, we borrow the *Pyramidal Depth Network* (PyD-Net) recently introduced in [122], achieving close to state-of-the-art accuracy and almost real-time performance on commercial high-end CPUs. Then, we propose an end-to-end optimization framework to reduce the complexity of the network and to deploy the compiled network on the target device. Specifically, the front-end of the optimizer performs an algorithmic-level optimization through quantization to 16-bit and to 8-bit fixed-point; the method is fast, highly accurate, and compliant to the hardware characteristics of off-the-shelf CPUs. The back-end is in charge of code compilation and of mapping the model on the target ARMv7 architecture by exploiting integer neural kernels designed in-house to maximize the utilization of the Single Instruction Multiple Data (SIMD) units available in the Cortex-A architecture.

The obtained results show that the quantized versions of PyD-Net (16- and 8-bit) have marginal accuracy losses, substantial speed-up w.r.t. the floating-point (32-bit) version, and remarkable memory savings.

3.3.1 Related Works

Quantization of CNNs

From a technical viewpoint, quantizing a network consists of searching the fixed-point format that minimizes the accuracy loss of the quantized network compared to the full precision network. Specifically, defining the fixed-point format requires choosing the bit-width and the radix-point position used for both the activation and the weight tensors of the network. As discussed in section 1.2, existing quantization techniques are based on classic methods from digital signal processing theory and differ in the strategy used to scale the radix-point, namely, linear or non-linear scaling, symmetric or asymmetric mapping, and in the granularity of the quantization, namely, per-network, per-layer, or per-channel. Note that a one-size-fits-all solution does not exist as the best method depends on the combination of the CNN architecture and on the characteristics of the underlying hardware.

Fixed-Point CNNs with ARMv7

Efficient processing of fixed-point CNNs requires optimized convolutional operators for integer operands. ARM released *Computing Library* [54] an open-source repository of low-level routines that support all the basic building blocks of neural models (e.g., Activation, Convolution, Fully-Connected, Normalization, and Pooling layers). Our preliminary analysis of version 18.0.5 of this library revealed that fixed-point convolutions with 8-bit performed slower than with 32-bit (single precision) floating-point. The same was confirmed at the time by another technical report [123], where authors claimed a 25% performance overhead.

3.3.2 PyD-Net Design and Training

Fig. 3.1 depicts an abstract view of PyD-Net, the CNN architecture adopted in this work. PyD-Net [122] is a Convolutional Neural Network designed to run efficiently on CPUs. Inspired by the most popular networks for vision applications, it relies on an encoder-decoder architecture. The encoder is an

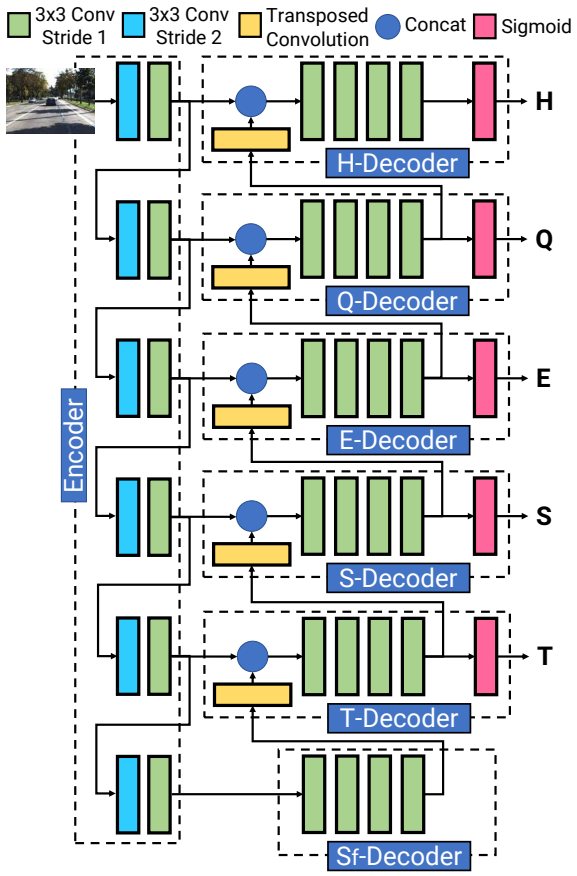


Fig. 3.1 PyD-Net architecture. H stands for $\frac{1}{2}$ of the input resolution, Q for $\frac{1}{4}$, E for $\frac{1}{8}$, S for $\frac{1}{16}$, T for $\frac{1}{32}$.

extremely lightweight pyramidal encoder that extracts high-level low-resolution features; the decoder has a modular architecture composed of multiple branches, generating high-resolution and dense disparity maps from the extracted features.

The pyramidal encoder halves the resolution of the feature maps at each level such that each decoder branch operates at a different spatial resolution. In this way, the top decoders capture the fine-grain details of the input image, while the bottom decoders the general context. A deconvolutional layer upsamples the disparity map obtained at a lower level before using it as an input for the higher-resolution branches.

The encoder comprises two 3×3 convolutional kernels at each level (from H to Sf), the first having a stride 2 for down-sampling, the second a stride 1 for feature extraction. The number of filters increases when moving from the

top to the bottom of the pyramid: 16, 32, 64, 96, 128, and 192. The decoder comprises six branches, producing depth maps at six different resolutions, from $\frac{1}{2}$ (H) to $\frac{1}{64}$ (Sf) of the input resolution. The decoder branches are structurally equivalent, having four 3×3 convolutional layers with 96, 64, 32, and 8 output channels and a leaky ReLU as the activation function. Each decoder branch processes the concatenation of the upsampled depth map from the previous level with the pyramidal features of the same level. Thus, the decoder branches on the bottom require fewer operations and occupy a smaller RAM footprint than those on the top, as they process tensors with a lower spatial resolution. The tensor produced by the sigmoid operator of the decoder branches at *half* H, *quarter* Q, and *eighth* E resolution constitute the three possible outputs of the network. By choosing which output decoder is used as the main output of the network, it is possible to trade-off accuracy and energy during inference. For instance, working at low-resolution E, the two topmost depth decoders are disabled, reducing the number of convolutional layers to be processed. This topic will be analyzed in greater detail in chapter 4.3. When compared with the more complex state-of-the-art encoder-decoder architecture of [95], PyD-Net [122] is about 16× smaller in size and 5× faster[1] (at half resolution H), yet achieving comparable depth accuracy.

Following [95], the network is trained on binocular stereo pairs, processing a single frame as input and reprojecting the other according to the estimated disparity. The following three paragraphs detail the loss terms composing the loss function, namely, appearance, smoothness, and left-right consistency, and describe all the steps of the reprojection process.

The appearance term measures the photometric difference between the input image I^l and the reprojected one $\tilde{I}^l$ obtained by warping the corresponding right image I^r, available during training, according to the estimated d^l:

$$\mathcal{L}_{ap}^{d^l} = \alpha \frac{1 - SSIM(I^l, \tilde{I}^l)}{2} + (1 - \alpha)||I^l - \tilde{I}^l|| \tag{3.1}$$

where $SSIM$ represents the Structural Similarity Index Measure [116] adopted to measure the similarity between the two images in combination with a standard ℓ_1 loss.

[1] On an Intel Core i7-6700K CPU (4.2Ghz) using 32-bit floating-point [122]

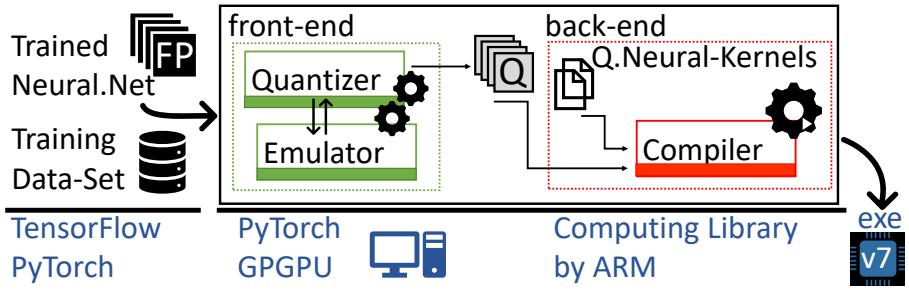


Fig. 3.2 Optimization and deployment flow targeting ARMv7a cores.

The smoothness term discourages large discontinuities in the estimated depth map, this term penalizes the difference between neighboring pixels in d^l, except where strong edges occur in I^l:

$$\mathcal{L}_{ds}^{d^l} = |\delta_x d^l| e^{-||\delta_x I^l||} + |\delta_y d^l| e^{-||\delta_y I^l||} \tag{3.2}$$

where $\delta_x(.)$ and $\delta_y(.)$ are gradients in horizontal and vertical direction, respectively.

The left-right consistency term [95] enfornces consistency between d^l and reconstructed $\tilde{d}^l$ obtained by warping d^r, a second output of the network aligned to I^r, according to d^l:

$$\mathcal{L}_{lr}^{d^l} = |d^l - \tilde{d}^l| \tag{3.3}$$

The three loss terms can be computed over d^r as well, by replacing $d^l, I^l, \tilde{I}^l$ with $d^r, I^r, \tilde{I}^r$. Finally, the single-scale loss is obtained by summing the three terms computed over both d^l and d^r. Note that supervision is provided to each output decoder by downsampling all terms to the resolution of the decoder.

3.3.3 Optimization Framework

Fig. 3.2 shows the end-to-end framework developed for the optimization and the deployment of quantized CNNs on ARMv7a cores. The flow comprises two main stages. The front-end translates a CNN trained with 32-bit floating-point numbers (FP32 in the figure) into a fixed-point model (16- or 8-bit). The back-end maps the high-level description of the CNN on an optimized low-

level code that can run on the target hardware platform. As efficient integer convolutional kernels were missing in publicly available libraries, such as the ACL [54] by ARM [123], we developed a new set of integer kernels (*Q.Neural-Kernels*) optimized for 16- and 8-bit inference. Such kernels are tailored to the Cortex-A architecture to fully exploit the single-instruction multiple-data (SIMD) unit embedded in modern ARM CPUs.

Model Optimization: Fixed-Point Quantization

Quantization to fixed-point arithmetic relies on an affine mapping of integers Q (represented with 16 or 8 bits) to floating-point numbers V:

$$V = K \cdot (Q - Q_0) \tag{3.4}$$

where K is scale factor, and Q_0 the quantized value corresponding to the floating-point value 0.

In this work, we resorted to a linear quantization with symmetric binary scaling: Q_0 is fixed to 0 (symmetric scaling); the scale factor K is a power of 2, i.e., $K = 2^{-FL}$, where FL is the fraction length, i.e., the position of the radix-point in Q (binary scaling). Thus, the affine mapping can be rewritten as

$$V = Q \cdot 2^{-FL} \tag{3.5}$$

Moreover, we adopted a hybrid quantization scheme, where the bit-width (either 16-bit or 8-bit) is fixed for all layers, whereas the radix-point is assigned per layer. Other quantization strategies, e.g., asymmetric scaling [40], may achieve higher accuracy but at the cost of additional computational stages. For instance, an asymmetric method, e.g. [40], requires additional output pipeline stages that negatively affect the performance as demonstrated in [123], while floating-point scaling requires complex data-type conversions instead of the simple shift operation required by the binary scaling. For the specific case of PyD-Net, our linear symmetric binary scaling quantization achieves almost the same accuracy of floating-point, making other more complex schemes irrelevant.

Given a set of real values, e.g., the weights and activations of PyD-Net, the bitwidth of the integer representation and the value of FL affect the accuracy

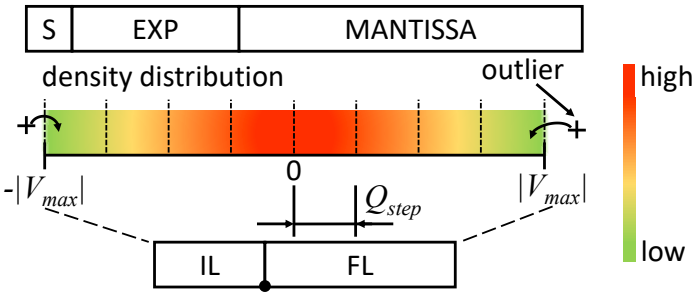


Fig. 3.3 Floating-Point to Fixed-point quantization.

loss due to quantization. This trade-off is pictorially shown in Fig. 3.3, where the gradient bar refers to the density distribution of the floating-point values, $|V_{max}|$ is the largest value representable with the fixed-point number, and Q_{step} represents the quantization step. Specifically, a small value of FL increases Q_{step} but also $|V_{max}|$, enlarging the range of representable numbers. The decision of which constraint to guard more ($|V_{max}|$ or Q_{step}) mainly depends on the distribution of the original values and their importance in the neural model.

Since the bit-width is defined by the available hardware (16 or 8 bits for the ARMv7a), the problem reduces to searching the optimal $|V_{max}|$ value. Then, the FL can be computed from $|V_{max}|$ using the following equation:

$$FL = \left\lfloor \log_2 \left(\frac{2^{BW-1} - 1}{|V_{max}|} \right) \right\rfloor \tag{3.6}$$

To choose the value of FL, we implemented an optimization procedure that minimizes the L2 distance between the original 32-bit floating-point values X and the quantized values Q. The procedure is applied to both activations and weights independently, and it works as follows. First, it performs a value distribution analysis of weights and activations; for the latter, a subset of the training set is used (referred as *calibration set*). Second, it computes the lower-bound and upper-bound of the fraction length: FL_{lb}, FL_{ub} using the following equation:

$$V_{max} = \begin{cases} \texttt{max}(|X_{min}|), |X_{max}|) & \text{for } FL_{lb} \\ \texttt{max}(|X_{min}|), |X_{max}|)/K & \text{for } FL_{ub} \end{cases} \tag{3.7}$$

with K an arbitrary large integer[2]. Third, it samples multiple values of FL in the range $[FL_{min}, FL_{max}]$ and it sets the FL_{opt} as the one minimizing the L2 error. Half-even rounding is used for the quantization of trainable parameters.

To emulate integer arithmetic during the training and validation stages, we built an in-house emulation engine that leverages the acceleration of GPGPUs. The engine implements the fake-quantization method introduced in [40]. During inference, a software wrapper converts activations and weights (stored in fixed-point) to the 32-bit floating-point. Once processed, data are converted back in fixed-point and adjusted with auxiliary transformations (e.g., saturation, truncation, binary-shift) that replicate the behavior of the fixed-point units of the ARMv7 core (e.g., the saturation of the accumulator register, the set-up of the radix-point position).

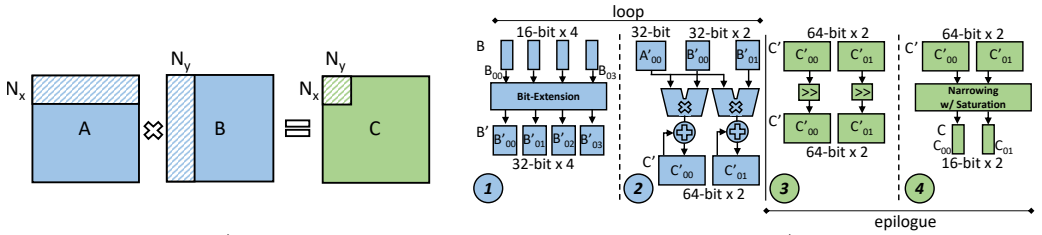
Finally, a fine-tuning stage is operated to recover the possible accuracy loss introduced by the quantization process. The fine-tuning works as follows: (i) the forward-propagation is run with fake-quantization; (ii) the gradients are back-propagated using the straight-through estimator method [124]; during this step, weights are kept in a floating-point format to allow small weight updates; (iii) weights are quantized at the end of each epoch using stochastic rounding. Compared to standard floating-point training, the execution time increases by 20%. For the backpropagation, we leveraged knowledge distillation by setting the quantized model as the student and the original floating-point network as the teacher. The training loop is driven by a multi-scale loss function that minimizes the mean squared error between the disparity maps inferred by the two actors (teacher d^{F} and student d^{Q}) at the different output resolutions:

$$\mathcal{L} = \sum_{s \in [\frac{1}{2}, \ldots, \frac{1}{32}]} ||d_s^{\mathrm{Q}} - d_s^{\mathrm{F}}||^2 \tag{3.8}$$

Fixed-point Neural Kernels

The common belief that fixed-point representations reduce energy consumption due to less complex arithmetic is not exactly true. Indeed, the correct processing of a fixed-point 2D convolution requires additional instructions not needed in floating-point, such as data extension and arithmetic shifts.

[2]We empirically verified K=100 is an optimal choice for PyD-Net



a) b)

Fig. 3.4 Q.Neural-Kernel: execution flow for 16-bit fixed-point.

The Cortex-A CPU targeted in this work hosts the NEON Media Processing Engine, a programmable Single-Instruction Multiple Data (SIMD) architecture that relies on multiple arithmetic units to accelerate parallel workloads, like CNN inference. The NEON architecture supports both parallel floating-point and integer instructions. The register file can be configured to host 8-, 16-, 32-, 64-, or 128-bit data, while the integer data-path supports 8-, 16-, 32- or 64-bit operations.

The proposed Q.Neural Kernels leverage a custom implementation of a GEneral Matrix-Multiplication (GEMM) algorithm [125], tailored to the NEON unit. A 2D convolution is then mapped to a GEMM through an im2col operation. In an optimized GEMM implementation, the input matrices are iteratively split into regular tiles to maximize data reuse across the memory hierarchy. Among all possible tiling choices, we resorted to an output stationary dataflow [126] (Fig. 3.4a), where the output matrix is divided in tiles of shape $N_{\mathrm{x}} \times N_{\mathrm{y}}$ and each output pixel stays stationary in a dedicated register of the register file till the end of all accumulations. As explained later in the text, the adopted dataflow is the most efficient choice in fixed-point convolutions.

The processing of each output pixel involves a sequence of MAC operations. Multiple output pixels can be computed in parallel depending on the precision adopted, 2 for 16-bit and 4 for 8-bit. A detailed example is reported in the schematic representation of Fig. 3.4b. It illustrates the parallel calculation of two outputs C_{00} and C_{01}. In general, $C_{ij} = \sum_k A_{ik} \cdot B_{kj}$, with $i \in [0, N_{\mathrm{x}}), j \in [0, N_{\mathrm{y}})$. The example is for 16-bit fixed-point (the same holds for 8-bit, yet with doubled parallelism). The flow is as follows:
(1) the 16-bit (8-bit) input operands, A_{ik} and B_{kj} are extended to 32-bit (16-bit) obtaining A'_{ik} and B'_{kj} (Fig. 3.4b refers to B_{kj} only);

(2) the input operand A'_{ik} is broadcasted to all arithmetic units to exploit data-reuse, while the other input B'_{kj} is streamed across the units; two (four) fused MAC operations are executed in parallel; the result is stored into a 64-bit (32-bit) register C'_{ij};
(3) at the end of the loop, the two (four) results are ready to be packed and then stored in the main memory; an output processing stage (highlighted in green) performs a simple binary shift based on the input and output radix-point position;
(4) the result is shrunk to the original bit-width, i.e., 16-bit (8-bit), and eventually saturated.

The bit-extension of step **(1)** guarantees 32- (16-) guard-bits for the accumulation. This operation is crucial as it avoids overflow/underflow during accumulation. Bypassing this stage may achieve twice the parallelism, but the results are highly inaccurate. Considering the larger bit-width of the partial sums, adopting an output stationary flow is of paramount importance, as it reduces the number of bytes moved across the memory hierarchy.

Concerning the shape of the output tiles, we empirically found that ($N_{\mathrm{x}} = 6, N_{\mathrm{y}} = 4$) for 16-bit and ($N_{\mathrm{x}} = 6, N_{\mathrm{y}} = 8$) for 8-bit achieve a good balance between computing and memory intensity.

Thanks to the adopted implementation choices, the benefit of precision scaling is twofold. First, the number of elements in the output tile doubles at lower precision, halving the data movement across the memory hierarchy, therefore reducing memory energy. Second, the number of MAC operations processed by a single instruction doubled from 2 (16-bit) to 4 (8-bit), enabling faster processing, hence improving energy efficiency.

As a side note, we point out that the parallelism of floating-point is 4. Moreover, floating-point requires fewer operations in the inner-loop as the additional output stage (steps (3) and (4) in Fig. 3.4b) is not needed. Despite that, the performance of fixed-point CNNs are better than the floating-point version. This gain is due to the following factors: (i) enhanced utilization of memory bandwidth; (ii) smaller memory footprint for storing weights and partial results, hence less RAM usage; (iii) higher hit-rate in caches. This analysis is confirmed by the experimental results discussed next.

3.3.4 Experimental Results

Experimental set-up

PyD-Net infers disparity maps at different resolutions. The three options for the output resolution, i.e., H, Q, and E, have been explored to assess the relationship between depth accuracy and the non-functional properties, namely, binary footprint, RAM space, throughput, and energy efficiency. All experiments were repeated for the three arithmetic precision configurations, namely, 32-bit floating-point, (**FP32**), 16-bit fixed-point (**FX16**), and 8-bit fixed-point (**FX8**). The baseline is PyD-Net with output resolution H and FP32 precision (i.e., H@FP32).

KITTI raw [56] is the reference dataset in this field [95, 122]. It collects 23297 images split, according to the standard protocol proposed by Eigen et al. in [105], into a training-set (22600 stereo pairs) and a test-set (697 images) with sparse ground-truth labels acquired with a LiDAR for the evaluation. The disparity maps obtained through the inference stage are transformed into depth maps following the methodology introduced by [95]. The baseline and the starting point of our work is the pre-trained PyD-Net model [122]. It was trained, as described in [122], for 200 epochs on batches of 8 images resized to 512×256.

For the quantization stage, we used a calibration set containing 5000 images randomly selected from the training set. Note that there is no intersection between the testing and calibration sets. The fine-tuning stage (applied after the quantization) consists of 25 training epochs over the full training set using the Adam optimizer [127]. We used the following hyper-parameters: learning rate 1.0e-7, $\beta_1 = 0.9$, and $\beta_2 = 0.999$, weight decay = 0.

The proposed GEMM-based Q.Neural-Kernels are written in C++ and inline assembly code. They are integrated into the ARM Compute Library version 18.05 built with `scons` ver. 2.4.1 and the *gcc-linaro* toolchain ver. 6.4.0-2018.05.

The embedded platform used as a test-bench is the Raspberry PI 3B with a 32-bit Ubuntu Mate 16.04 as the operating system. The board hosts a quad-core BCM2837 chip-set. The board consumes 3.5 W under full utilization

Table 3.2 Experimental results concerning depth estimation accuracy. Comparison between original PyD-Net [122] (FP32) and optimized architectures at different resolutions.

			Lower is better		Higher is better		
Config.	Abs Rel	Sq Rel	RMSE	RMSE log	$a1$	$a2$	$a3$
H@FP32	0.146	1.298	5.859	0.241	0.802	0.927	0.968
H@FX16	0.147	1.331	5.925	0.243	0.801	0.926	0.967
H@FX16-ft	0.147	1.302	5.945	0.244	0.798	0.925	0.967
H@FX8	0.177	1.893	6.621	0.272	0.768	0.911	0.958
H@FX8-ft	0.148	1.337	6.018	0.246	0.795	0.924	0.967
Q@FP32	0.149	1.350	6.128	0.246	0.795	0.923	0.966
Q@FX16	0.149	1.365	6.155	0.246	0.794	0.923	0.966
Q@FX16-ft	0.149	1.342	6.176	0.248	0.790	0.921	0.966
Q@FX8	0.183	2.364	7.457	0.270	0.766	0.908	0.957
Q@FX8-ft	0.158	1.427	6.290	0.257	0.778	0.917	0.964
E@FP32	0.162	1.699	7.141	0.266	0.768	0.907	0.959
E@FX16	0.165	1.770	7.327	0.271	0.762	0.904	0.957
E@FX16-ft	0.162	1.712	7.163	0.266	0.768	0.907	0.958
E@FX8	0.193	2.758	8.507	0.288	0.745	0.893	0.950
E@FX8-ft	0.171	1.829	7.430	0.276	0.751	0.901	0.956

(4 cores active and maximum utilization); moving from FP32 to FX16 or FX8 has a negligible effect on the total power consumption, dominated by memory accesses. Thus, energy consumption is inversely proportional to latency.

Accuracy vs. Quantization

Tab. 3.2 reports an evaluation of several variants of the original PyD-Net architecture. As the model by Godard et al. [95] has 15× the parameters of PyD-Net (30 vs. 1.9 million), it is not suited for embedded devices, and so we did not use it as a benchmark for non-functional properties in the following subsection. Here, we discuss in detail the effects of the data-type (single-precision floating-point, 16 or 8 bit fixed-point, referred to as FP32, FX16, and FX8) on the accuracy of the network at the different output resolution (H, Q, E) and optional fine-tuning (-ft) carried out after quantization. Specifically, we analyze error and accuracy metrics commonly adopted for evaluating depth-from-mono performance [105], assuming a 80m cap distance [95] (see section 3.2 for further details on the metrics). Tab. 3.2 reveals a similar trend for the three resolution H, Q, and E, for all the seven metrics. Specifically, the 16-bit fixed-point quantization FX16 introduces a negligible accuracy drop compared to the

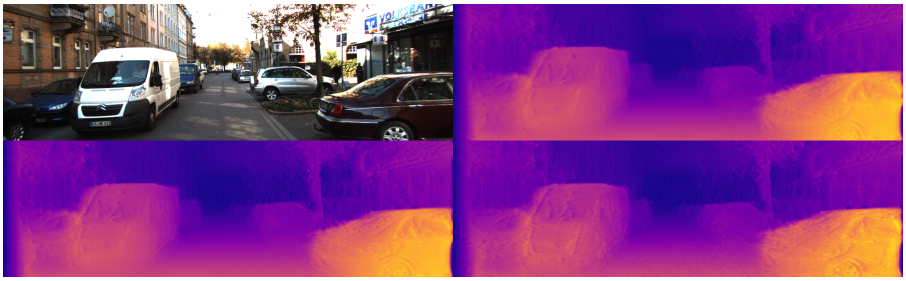


Fig. 3.5 Top row: Input image from KITTI dataset (left) and depth map H@FP32 computed by the original PyD-Net network [122] (right). Bottom row: depth maps H@FX16-ft (left) and H@FX8-ft (right).

original PyD-Net FP32; thus, the additional fine-tuning (-ft) brings a marginal accuracy gain. The 8-bit fixed-point quantization (FX8) leads to a more substantial accuracy loss, especially for Q and E output resolutions. However, fine-tuning the network (-ft) remarkably improves performance, closing the gap with FX16 quantization and, most notably, with the original FP32 strategy. Fig. 3.5 reports a qualitative comparison between the depth estimated at different resolutions and quantization levels.

Performance, Memory Space and Energy Efficiency

Tab. 3.3 reports the hardware-related metrics measured during the test-run, namely, memory space (for weights storage), RAM usage (during PyD-Net processing), and energy efficiency (average Frames per J). The throughput (Frames/second) can be derived by multiplying energy efficiency (Frames/J) by the total average power consumption (3.5 W).

As expected, energy efficiency improves when working at lower resolutions: the upper branches of the decoders are disabled, highly reducing the computational burden. For instance, considering the FP32 configuration, the improvement from high (H@FP32) to low resolution is 5.64× thanks to the reconfigurable topology of PyD-Net. When fixing the output resolution, energy efficiency increases with smaller data representations. For instance, at high resolution, the gain from H@FP32 to H@FX8 is 49%; at low resolution, the gain grows up to 63.7%. To notice that, as previously outlined, moving from floating-point to fixed-point affects accuracy only marginally. The combined action of resolution scaling and precision scaling enables even larger optimiza-

Table 3.3 Non-functional metrics of PyD-net at different resolutions and precisions on ARMv7-A

Resolution	Precision	Space (MB)	RAM (MB)	Frame/J
H	FP32	7.6	206	0.141 (×**1.00**)
	FX16	3.8	118	0.156 (×**1.11**)
	FX8	1.9	62	0.210 (×**1.49**)
Q	FP32	7.2	60	0.386 (×**2.74**)
	FX16	3.6	34	0.420 (×**2.99**)
	FX8	1.8	19	0.635 (×**4.51**)
E	FP32	6.8	53	0.794 (×**5.64**)
	FX16	3.4	28	0.922 (×**6.55**)
	FX8	1.7	14	1.299 (×**9.23**)

tion: from H@FP32 (0.141 Frames/J) to E@FX8 (1.299 Frames/J) the energy efficiency increases by 9.23×.

This first analysis gives clear evidence of the scaling properties of the quantized PyD-Net model, the benefits of multiple precision configurations, and the effectiveness of the porting flow on the target architecture. A sensing technology with such energy-quality scaling characteristics represents a practical option for those embedded applications that can tolerate lower accuracy for higher energy efficiency by tuning resolution (coarse-knob) and precision (fine-knob). For a more detailed discussion, the reader could refer to chapter 4.3.

Concerning memory, both the binary space and the RAM usage are important metrics as they reflect the efficiency of the proposed implementation. As expected, the space for storing network parameters reduces linearly with the precision, e.g., from 7.6 MB with H@FP32 to 1.9 MB with H@FX8. At low resolution and low precision, the memory space is just 1.7 MB (E@FX8), which brings the overall savings w.r.t. H@FP32 up to 4.5×. Even more interesting is the analysis of the RAM. Its utilization is dramatically reduced with an overall scaling factor of 14.7×: from H@FP32 (206 MB) to E@FX8 (14 MB).

As a final remark, Fig. 3.6 provides a technology comparison among different hardware options: a GPU (Titan X Maxwell), a high-end CPU (Intel i7-6700K CPU), and the ARMv7 at FP32, FX8, and FX16. The bar chart shows the energy efficiency (Frames/J) for all the possible permutations of output resolution, precision, and hardware; the labels refer to the normalization w.r.t. high resolution (H) for each hardware option separately. Moving from H to E

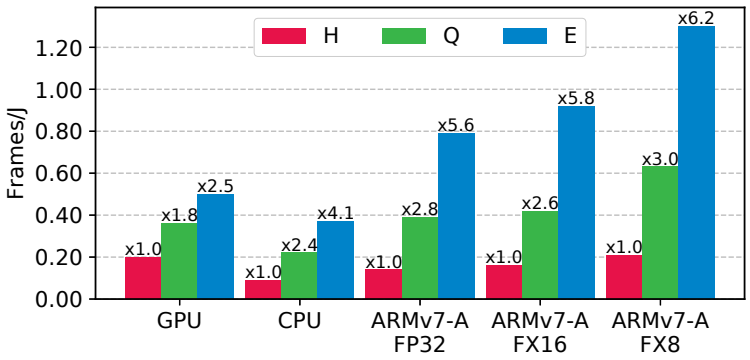


Fig. 3.6 Energy efficiency vs. Resolution using GPU, CPU and the ARMv7-A.

with the ARMv7@FX8 improves energy by 6.2×. A more interesting aspect is that the lower the arithmetic precision, the larger the gain brought by resolution scaling. While for the GPU and the desktop-CPU, the gain from H to E is limited to 2.5× and 4× respectively, the energy gain increases to 5.6× with the ARMv7@FP32 and 6.2× with the ARMv7@FX8.

3.3.5 Discussion

This work introduces a comprehensive design and optimization framework to improve the energy efficiency of depth perception on low-power embedded devices. To assess the effectiveness of our proposal, we conducted an extensive evaluation on an embedded system powered by the ARMv7a, a widely adopted RISC architecture. The collected experimental results showed that the joint co-operation between (*i*) the design of the tiny yet reconfigurable PyD-Net and (*ii*) the optimization enabled by the hardware-friendly fixed-point quantization allows achieving acceptable accuracy with energy efficiency beyond the state-of-the-art. These features pave the way to the widespread deployment of monocular depth sensing in applications constrained by stringent energy requirements.

3.4 Enabling Depth Estimation on Microcontrollers

Enabling monocular depth estimation on tiny end nodes powered by MCUs may offer interesting opportunities in the Internet-of-Things (IoT) space. Specifically,

distributed visual sensors can evolve from data collectors to smart hubs capable of inferring depth locally with higher quality-of-service and increased users privacy. Unfortunately, MCUs are orders of magnitude less performing than embedded CPUs and GPUs, so despite the efforts carried out in our previous work and by other active researchers in the community, deploying monocular depth estimation on such devices is still extremely challenging. Even the smallest CNNs [122, 120], in fact, are designed for platforms with multi-core high-frequency CPUs and large RAMs, thus consuming 3.5 to 10 W. Instead, common MCUs run at a much lower frequency (hundreds of MHz vs. 1–2 GHz) and have very few on-chip memory resources (hundreds of kB vs. 2–8 GB). Meeting such constraints represents a challenge in computer vision and depth perception, which, in our opinion, calls for a paradigm shift: quality is no longer the only objective as other extra-functional metrics need to be considered at the algorithmic level and during the whole compilation flow.

To this end, the first assumption made in this work is that processing high-resolution images is not feasible nor beneficial for practical use cases. Hence in this work, differently from the traditional high-quality vision systems, we adopt low-resolution images to meet the stringent hardware constraints of MCUs. Processing low-resolution tensors, in fact, reduces the number of operations to be processed and the memory footprint to store hidden features maps. However, the in-depth analysis conducted in Sec. 3.4.5 demonstrates that resolution scaling alone is not enough to fit current models on MCUs and that achieving such a goal requires additional design and optimization efforts. On the neural architectural side, we propose a novel lightweight architecture referred to as μPyD-Net designed explicitly for processing low-resolution images (i.e., 48×48 or 32×32) on MCUs. The internal topology of μPyD-Net was optimized with hardware-conscious techniques to maximize the savings brought by inputs resolution scaling. At the algorithmic level, we present a supervised training flow from images and full-resolution disparity maps obtained with SGM. The training process is characterized by low requirements but equivalent performances of more costly strategies. At the operator level, we perform a highly-accurate 8-bit quantization and an efficient mapping on low-level routines specialized for the target architecture.

The two main achievements of the work presented in this section are:

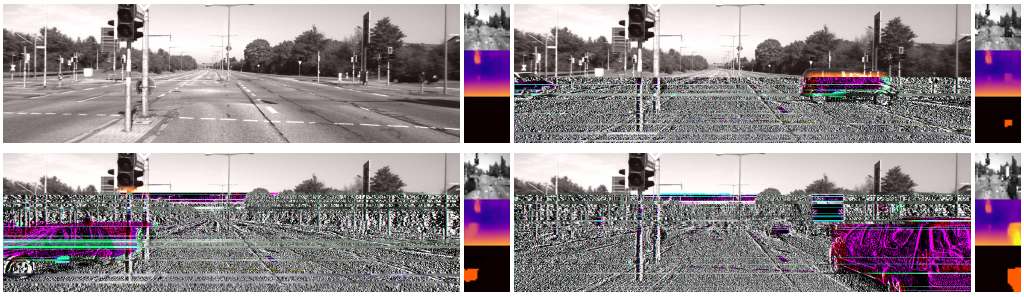


Fig. 3.7 Example of traffic monitoring system based on μPyD-Net.

- enabling for the first time monocular depth estimation on low-power MCUs, such as the Arm Cortex-M7 CPU;
- demonstrating how to obtain a meaningful coarse depth representation from a low-resolution image, as low as 48×32 and 32×32 pixels.

An extensive experimental assessment shows that, despite the low input resolution, μPyD-Net achieves, on the standard KITTI dataset [56], a depth accuracy comparable to seminal works [105, 128], although not on par with the current state-of-the-art [129, 114, 130]. However, this is not surprising as such methods use higher input resolution and much more complex models, unfeasible requirements for the devices used at the edge of the IoT.

3.4.1 Practical Use Cases

Processing high-resolution images to produce high-resolution dense depth maps is paramount to meet the quality constraint of some vision tasks, such as 3D reconstruction and SLAM. However, a coarse low-resolution depth estimate may be sufficient in other common edge applications, such as object/people counting [131, 132], pose estimation [133], action recognition [134], and vehicle detection [135]. Indeed, millimetric depth estimations are not strictly required to accomplish such tasks. To this end, in this section, we qualitatively assess the use of coarse depth estimation on two applications reported in Fig. 3.7 and Fig. 3.8.

The first one is a simple traffic monitoring system for counting, for instance, cars. Fig. 3.7 shows a traffic monitoring system based on μPyD-Net. For each example, we show the high-resolution frame, followed by the 32×32 image

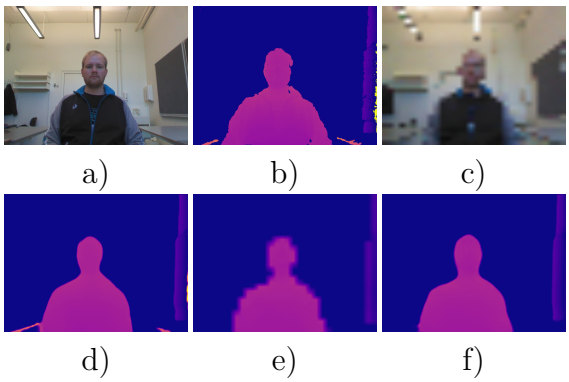


Fig. 3.8 Results on a testing image of the VAP dataset [136]. a) RGB original frame, b) ground-truth depth acquired using Kinect, c) RGB input image resized to 32×32, d) maps predicted by PyD-Net, e) maps predicted by μPyD-Net, and f) outcome of the super-resolution network fed with the μPyD-Net map.

processed by the network (top-right corner) and its output (mid-right corner). In the bottom-right corner, for each example, we show the differences in the coarse 3D structure of the scene with respect to the structure of the environment itself, which has been acquired in absence of vehicles (top-left example). The highlighted regions depict the changes in the depth maps inferred by μPyD-Net when a vehicle enters the scene. It is worth noticing that, even with small input images, the low operating frequency and parallelism of MCUs prevent real-time performance (i.e., >30 FPS). However, the focus here is on those applications with no strict timing constraints, like traffic congestion monitoring, and not on those requiring fast decision-making, such as autonomous driving.

The second application concerns a simple privacy-preserving monitoring system capable of performing a remote video analysis without revealing the user identity [134]. The idea is that user privacy can be enforced by performing the analysis in the depth domain. Fig. 3.8c reveals that low-resolution images can only partially hide distinctive features of the person as some clues can still be inferred. Instead, by moving the image to a pure depth domain as reported in Fig. 3.8 d, it is possible to hide personal details while keeping the relevant information required for the high-level task. Enabling these sensing capabilities on tiny devices can reduce the design cost and the power consumption of systems compared to using RGB-D cameras (e.g., the Kinect in (b)) or standard CNNs for monocular depth estimation (d) while achieving similar accuracy. Indeed, the map inferred by μPyD-Net (e) and post-processed by a super-resolution

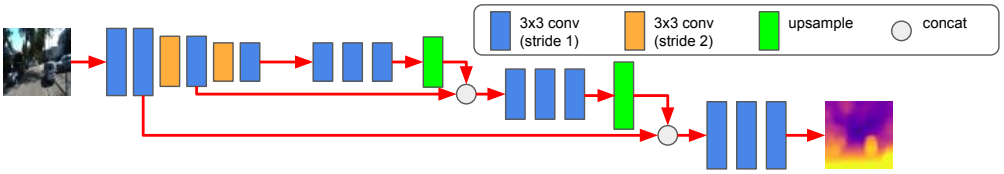


Fig. 3.9 μPyD-Net architecture.

network running on the cloud (f) achieves results comparable to the two baselines.

3.4.2 μPyD-Net architecture

Fig. 3.9 sketches the architecture of μPyD-Net. It follows the encoder-decoder architecture of PyD-Net 3.3.2, but it has a different topology and different pooling and stride parameters to work with low-resolution inputs and to reduce the computational complexity at higher resolutions. Specifically, a shallow encoder extracts a three-level pyramid of features using six 3×3 convolutional layers with leaky ReLU as activation function ($y = \alpha \cdot x$ if $x > 0$ else x, with α=0.125), producing 8, 8, 16, 16, 32, and 32 output channels. The decoder comprises three branches composed of three convolutional layers, followed by leaky ReLU (except the last one). Each decoder branch processes one pyramid level, producing 32 features. The output of the last layer of each decoder branch is up-sampled through a 2×2 transposed convolution layer. This extremely compact architecture, having only 100K parameters, is designed to process tiny resolution images, and thus it is tailored to low power devices such as MCUs. The topology of the network, its low parameter count, and the use of a low image resolution, i.e., 48×48 and 32×32, allow the model to fit within the 512 kB of RAM and to break the 1 FPS barrier on off-the-shelf MCUs, as it will be shown in detail in the experimental results section. Adding more layers either to the features extractor or to the decoders would make one or both the requirements not met.

Achieving a reasonable accuracy in the depth estimation with such small architecture is not straightforward. Specifically, in addition to the issues induced by processing low-resolution images, e.g., loss of details, providing supervision at shallow resolution is challenging. In particular, when the annotation is

sparse like in the KITTI dataset [56], downsampling the sparse depth data to the input resolution of μPyD-Net would make labels no longer reliable due to the interpolation process. For this reason, during training, we rely on proxy-supervision [114] obtained through Semi-Global Matching (SGM), a traditional stereo algorithm [100].

3.4.3 Proxy Supervision

Similar to other seminal works in the field [109, 95], we replace the need for accurate ground truth depth annotations during training using view synthesis. In particular, we exploit a pair of synchronized images acquired by a stereo camera to formulate a re-projection loss for supervision [95]. Formally speaking, given a stereo pair made of images L and R, the network is trained to infer an inverse depth map (i.e., the disparity map) D^L starting from L. Then, $\tilde{R}$ is obtained by warping the R view according to D^L. Finally, we compute a photometric loss $\mathcal{L}_{ap}^{l}$ between L and the warped image $\tilde{R}$ as proposed in [95] and reported in the following equation:

$$\mathcal{L}_{ap}^{l} = 0.85 \cdot \frac{(1 - \text{SSIM}(L, \tilde{R})}{2} + 0.15 \cdot |L - \tilde{R}| \tag{3.9}$$

To further enforce the self-supervision, we train the network also to infer a synthetic disparity map D^R for the right image R such that a consistency loss can be computed between the two disparity maps. Specifically, an equivalent $\mathcal{L}_{ap}^{r}$ signal can be obtained comparing R with warped image $\tilde{L}$.

However, due to the low resolution of the input images, a photometric loss alone cannot provide sufficiently reliable supervision. Hence, inspired by other recent works [137, 94, 114], which have leveraged noisy disparity estimations produced by traditional stereo algorithms for supervision, we use the SGM algorithm [100] to generate dense proxy labels from stereo pairs [114]. Specifically, for each pixel p and disparity hypothesis d, a Hamming matching cost $C(p, d)$ is computed between 9×7 census transformed images and then refined through multiple scanline optimizations as follows:

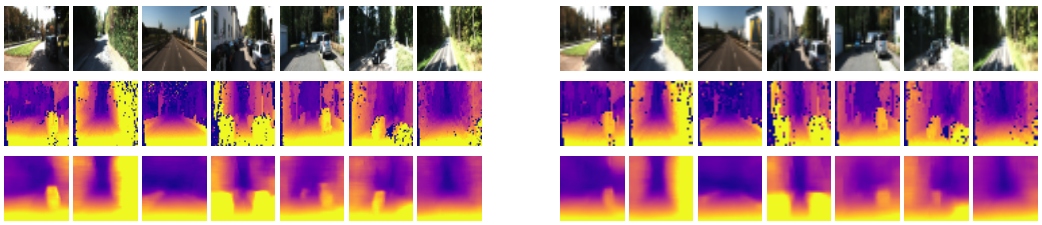


Fig. 3.10 Examples of self-sourced proxy labels on 48×48 (left) and 32×32 (right) images. From top to bottom, reference images, disparity maps produced by SGM [100], and predictions by μPyD-Net.

$$\begin{aligned} E(p,d) =& C(p,d) + \min_{q>1}[C(p',d), C(p',d \pm 1) + P1, \\ & C(p',d \pm q) + P2] - \min_{k<D_{max}}(C(p',k)) \end{aligned} \tag{3.10}$$

where P_1 and P_2 are two smoothness penalties, which discourage disparity gaps between p and previous pixel p' along the scanline path. A *winner-takes-all* strategy is applied after summing up the outcome of each optimization phase. Finally, the *left-right consistency* constraint is used to filter out the outliers as follows. We first compute the disparity maps D^L and D^R with SGM, respectively assuming as reference left and right images, then we invalidate the pixels having different disparities across the two maps higher than a certain threshold. The method is reported in the following equation:

$$D(p) = \begin{cases} \tilde{d}(p) & \text{if } |D^L(p) - D^R(p - D^L(p))| \leq \varepsilon \\ -1 & \text{otherwise} \end{cases} \tag{3.11}$$

To effectively scale the proxy labels to the resolution of the network inputs, SGM is run on the images at the original resolution $W \times H$, then the proxy is downsampled respectively to 48×48 and 32×32 using nearest-neighbor interpolation, and the disparity values are properly scaled by $\frac{48}{W}$ and $\frac{32}{W}$. As outliers are filtered out, enforcing the left-right consistency constraint, the produced labels are not fully dense. Nonetheless, most points survive this process, and each valid value available in the inverse depth map is obtained without any interpolation from nearby points.

The obtained labels are then used to provide supervision to μPyD-Net employing a reverse Huber (berHu) loss [138]:

$$\mathcal{L}_{ps} = \frac{1}{N} \sum_{p} berHu(d(p), \tilde{d}(p), c) \tag{3.12}$$

$$berHu(d(p), \tilde{d}(p), c) = \begin{cases} |d(p) - \tilde{d}(p)| & \text{if } |d(p) - \tilde{d}(p)| \leq c \\ \frac{|d(p) - \tilde{d}(p)|^2 - c^2}{2c} & \text{otherwise} \end{cases} \tag{3.13}$$

where $d(p)$ and $\tilde{d}(p)$ are, respectively, the predicted disparity and the proxy annotation for pixel p while c is set as $\alpha \texttt{max}_p |d(p) - \tilde{d}(p)|$, with $\alpha = 0.2$.

Fig. 3.10 shows, from top to bottom, some qualitative examples of low-resolution images (48×48), followed by proxy labels generated by SGM and disparity maps estimated by μPyD-Net. μPyD-Netaccurately reproduces inverse depth estimations consistent with the self-sourced annotations.

Finally, the loss function is obtained summing the proxy-supervision with the contribution given by photometric loss:

$$\mathcal{L}_{init} = \alpha_{ap}(\mathcal{L}^l_{ap} + \mathcal{L}^r_{ap}) + \alpha_{ps}(\mathcal{L}^l_{ps} + \mathcal{L}^r_{ps}) \tag{3.14}$$

We tuned α_{ap} and α_{ps} following [114]. Although we verified that SGM is highly effective, other sources of proxy labels can be adopted as well, such as stereo networks trained in a self-supervised manner with photometric losses [139], or in a supervised manner at the cost of requiring ground truth labels. The effect of the different strategies on the accuracy of μPyD-Net will be assessed in the experimental results section.

3.4.4 Optimization Stack

The optimization stack designed for the deployment of μPyD-Net into Cortex-M MCUs is depicted in Fig. 3.11. It is similar to the stack presented in section 3.3.3, and it comprises two main stages. The *front-end* quantizes the model to an 8-bit

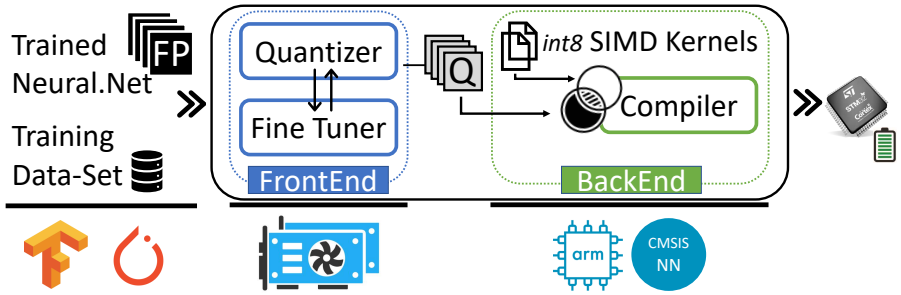


Fig. 3.11 Optimization framework.

fixed-point representation; the *back-end* translates the high-level description of the quantized network into low-level routines optimized for the target device.

Given the tight memory constraints and limited computational resources of low-power MCUs, quantization, in this case, represents a must rather than an optimization option. For instance, for the case of the Arm Cortex-M MCU family, widely used in low-cost and low-power embedded systems, the FLASH memory is limited to $1-2$ MBs, while the SRAM to $16-512$ KBs. The FLASH memory stores the application binary file and the network weights, copied into the SRAM at initialization time. The SRAM stores the intermediate feature maps of the network, whose size is not negligible in deep CNNs like μPyD-Net. Intuitively, with such minimal memory resources, 8-bit quantization is highly desirable, as it brings a $4\times$ memory footprint reduction compared to 32-bit values. Second, the floating-point unit is optional for many chip-sets of the Cortex-M architectural family, and, when available, it only supports scalar operations. On the other hand, the integer instruction set provides a 2-way Single-Instruction Multiple-Data (SIMD) data path (available on the M7 core), which double the theoretical throughput achievable by optimized integer kernels.

Similar to the work presented in Section 3.3, we adopt a linear quantization scheme with power-of-two scaling. The radix-point of both feature maps and weights is assigned layer-by-layer, and the bitwidth is fixed at 8-bits. To calculate the optimal radix-point, we use the same heuristic described in section 3.3.3. The accuracy loss due to quantization is recovered through a post-training *fine-tuning* stage based on knowledge distillation [93].

After quantization and fine-tuning, the network is deployed on the target device. To this end, we use the CMSIS-NN library developed by ARM, as it includes a series of efficient handwritten routines for integer CNNs. However, we had to develop some operators as the CMSIS-NN version available at the time of this project was missing a 2D transposed convolution (also called deconvolution) and the leaky ReLU activation function. For the transposed convolution, we implement it as a two stages operator: first, the input features are upsampled using a factor equal to the stride, then convolved with unit stride. For the leaky ReLU, we implement it as a simple masked shift operation by constraining the slope α to be a power of two. We observed that this choice achieves better performance without affecting the network accuracy.

3.4.5 Experimental Results

This section reports the results of the exhaustive experimental evaluation aimed at assessing the accuracy and the non-functional figures of merit, namely, latency and memory footprint, of μPyD-Net.

Dataset & Training

The KITTI stereo dataset [56] is a collection of rectified stereo pairs, comprising 61 scenes (more than 42K stereo frames) related to driving scenarios. It is the standard dataset for evaluating monocular depth estimation methods. The average image resolution is 1242×375. The depth annotations were obtained through a LiDAR device, mounted and calibrated in proximity to the left camera. We divided the overall dataset into two subsets, composed respectively of 29 and 32 scenes, according to the *Eigen split* [105, 95]. We used 697 frames of the first group as the test set and 22600 frames of the second as the training set.

The CityScapes dataset [140] contains stereo pairs capturing scenes from 50 cities in Germany taken from a moving vehicle in various weather conditions. It consists of $22,973$ stereo pairs with a resolution of 2048×1024 pixels. It is often adopted only for pre-training [95, 113, 114] the network since no ground truth maps are provided. As in [95] the lowest 20% of each stereo pair is discarded at training time.

The Make3D dataset [107] consists of a set of images and depth maps from a custom-built 3D scanner, collected during daytime in a diverse set of urban and natural areas in the city of Palo Alto and its surrounding regions. It contains 534 images at 1704×2272 resolution. We run experiments on the 134 testing images without retraining as in [95, 58].

Hardware Set-up

The proposed μPyD-Net is tested and validated on a NUCLEO-F767ZI[62] development board by ST-Microelectronics. It is powered by a chip-set featuring an Arm Cortex-M7 CPU clocked at 216 MHz and hosting 512 kB of SRAM and 2 MB of FLASH memory. As reported in the data-sheet [141], the current consumption is $\approx$100 mA for a data-intensive application run in similar operating conditions of our experiments. Hence, the power consumption is $<$400 mW. The `.C` description of the optimized μPyD-Net model is compiled using the *GNU Arm Embedded Toolchain*, version 6.3.1, and flashed into the board using the *mbed-cli* toolchain.

Evaluation – Functional metrics

We evaluate predictions according to standard functional metrics [105, 95]: *Abs rel*, *Sq rel*, *RMSE* and *RMSE log* represent error measures ($\Downarrow$, the lower the better), while $\alpha < K$ the percentage of predictions whose maximum between ratio and inverse ratio with respect to the ground truth is lower than a threshold K ($\Uparrow$, the higher the better). The detailed formulation of each metric can be found in sec. 3.2.

Since the performance of μPyD-Net is limited to the accuracy of the proxy labels used for supervision, we study the effect of the strategy used to obtain the labels on the accuracy of the network. Specifically, we consider labels obtained by SGM and by distillation [139] from two state-of-the-art CNNs, namely, UnOS by Wang *et al.* [142] and DispNet-CSS by Ilg *et al.* [143], respectively trained with self-supervision and ground truth. For both, we use the weights made available by the authors, respectively trained with the photometric loss on the full KITTI dataset (UnOS) or with ground truth on the SceneFlow dataset [103] and fine-tuned on KITTI 2015 training set (DispNet-CSS).

Table 3.4 Proxy labels accuracy on the test set of KITTI dataset [56] using the split of Eigen et al. [105], maximum depth set to 80m.

		Lower is better ⇓				Higher is better ⇑		
Method	Resolution	Abs Rel	Sq Rel	RMSE	RMSE log	α <1.25	$\alpha < 1.25^2$	$\alpha < 1.25^3$
SGM [100]	native	0.064	0.584	3.700	**0.149**	**0.951**	**0.976**	**0.986**
UnOS [142]	native	0.064	0.582	3.690	0.169	0.932	0.964	0.979
DispNet-CSS [143]	native	**0.060**	**0.442**	**3.543**	0.167	0.940	0.967	0.981
SGM [100]	48×48	0.415	10.963	11.836	0.481	0.539	0.785	0.881
SGM [100]	$\downarrow 48 \times 48$	0.107	1.594	5.556	0.199	0.906	0.960	0.978
UnOS [142]	$\downarrow 48 \times 48$	0.102	1.145	5.140	0.197	0.901	0.955	0.976
DispNet-CSS [143]	$\downarrow 48 \times 48$	**0.089**	**0.723**	**4.420**	**0.183**	**0.909**	**0.961**	**0.980**
SGM [100]	32×32	0.652	21.745	15.319	0.638	0.370	0.667	0.797
SGM [100]	$\downarrow 32 \times 32$	0.133	2.083	6.448	0.222	0.873	0.949	0.974
UnOS [142]	$\downarrow 32 \times 32$	0.131	1.622	6.040	0.221	0.867	0.945	0.972
DispNet-CSS [143]	$\downarrow 32 \times 32$	**0.110**	**0.940**	**4.948**	**0.199**	**0.882**	**0.955**	**0.978**

Tab. 3.4 collects results concerning a comparison between the three, conducted on the Eigen test split. We first report the accuracy of the disparity maps processed at full resolution to highlight the importance of spatial resolution. We highlight the almost equivalent performance of SGM and UnOS on error metrics, whereas DispNet-CSS produces better results. Concerning alphas, SGM produces better accuracy. Considering the 48×48 resolution, we report four main experiments respectively evaluating disparity map obtained by running SGM on 48×48 images and by either SGM, UnOS, and DispNet-CSS at full resolution and then downsampled using nearest-neighbor interpolation ($\downarrow 48 \times 48$), i.e., to the resolution used to train μPyD-Net. We cannot run either UnOS or DispNet-CSS at 48×48 because of their high compression factor ($\frac{1}{64}$), requiring larger images. It is important to notice that running SGM directly on 48×48 images leads to terrible performance. The high quantization of pixels at this resolution, in fact, makes this solution unreliable for training CNNs using these labels. Conversely, full-resolution images downsampled to 48×48 maintain acceptable performance, with DispNet-CSS labels resulting in a more performant training. In general, SGM and UnOS are close in performance, with the former resulting slightly more accurate and thus preferable to train μPyD-Net. The same behavior can be observed by running experiments at 32×32 resolution. Although DispNet-CSS labels show much higher accuracy compared to SGM and UnOS, they need ground truth labels to be obtained. Next, we will highlight that training μPyD-Net on DispNet-CSS rather than SGM achieves only minor improvements, thus making SGM better suited for practical applications.

Table 3.5 Ablation study on the test set of KITTI dataset [56] using the split of Eigen et al. [105], maximum depth set to 80m.

				Lower is better ⇓		Higher is better ⇑		
Supervision	Res.	Abs Rel	Sq Rel	RMSE	RMSE log	α <1.25	$\alpha < 1.25^2$	$\alpha < 1.25^3$
Photo	48×48	0.260	3.386	9.416	0.391	0.593	0.805	0.903
SGM	↓ 48×48	0.200	2.107	7.144	0.295	0.707	0.871	0.943
UnOS	↓ 48×48	0.200	2.095	7.224	0.298	0.701	0.870	0.944
DispNet-CSS	↓ 48×48	**0.191**	**1.825**	**6.697**	**0.286**	**0.716**	**0.881**	**0.949**
Photo	32×32	0.315	4.984	11.007	0.451	0.539	0.764	0.879
SGM	↓ 32×32	0.221	2.547	7.625	**0.312**	**0.681**	**0.858**	**0.935**
UnOS	↓ 32×32	0.221	2.625	7.623	0.313	0.677	0.853	**0.935**
DispNet-CSS	↓ 32×32	**0.217**	**2.195**	**7.171**	0.314	0.680	0.855	0.934

Finally, we study the effectiveness of μPyD-Net variants trained with different sources of self-supervision. Tab. 3.5 collects results of 48×48 and 32×32 models trained respectively with image reprojection losses [95], proxy labels sourced through SGM algorithm, UnOS and DispNet-CSS. At first, we point out how the supervision from photometric losses performs much worse than using proxy labels. Although this approach is extremely popular [95, 122, 113], we argue that, intuitively, the image content at such low resolution is much lower compared to the one available at the original resolution, thus leading to poor supervision. Exploiting the guidance from accurate disparity maps at training time boosts the accuracy achieved by μPyD-Net. Nevertheless, although the proxy labels show different accuracy according to Tab. 3.4, in particular comparing rows 4, 5,6 and 8, 9, 10 sourcing supervision from SGM algorithm results slightly better than using UnOS, with a small margin compared to DispNet-CSS, although DispNet-CSS needs ground truth for training. Thus, we prefer the SGM solution for practical applications because of the smaller accuracy gap than DispNet-CSS but greater flexibility.

In Tab. 3.6 we compare μPyD-Net with state-of-the-art solutions for monocular depth estimation. The upper portion of the table contains complex architectures with millions of trainable parameters, suited only for high-end GPUs (e.g., the NVIDIA Titan XP). On the other hand, the lower portion of the table lists networks requiring much less computational and memory requirements compatible with a broader range of devices. Moreover, we also report the resolution of the input image for each network. At first glance, we can notice the large gap between the amount of information processed by μPyD-Net and other proposals. However, shrinking the image using this

extreme factor (down to $\frac{1}{38}$ for width, $\frac{1}{11}$ for height in case of 32×32 images) has a non-negligible impact on the input image fed to μPyD-Net. This degradation is particularly evident for small objects at a longer distance or thin structures as poles, causing higher errors than the ground truths acquired at full resolution. For this reason, Tab. 3.6 also includes the lightweight PyD-Net [122] network processing much larger 256×512 images. Nonetheless, it is important to notice that, even stretching the input of other proposals to either 48×48 or 32×32, they would not run on the targets hardware device due to excessive memory requirements. Moreover, PyD-Net [122] would not be compatible with such tiny image sizes since its pyramidal structure is too deep. However, as pointed out by previous studies in other fields [144], the image content encoded in such tiny images is still enough to estimate a coarse estimation of the scene, comparable to state-of-the-art techniques proposed just a few years ago [105, 128], with hundred times fewer parameters and computational requirements. Not surprisingly, 48×48 input images yield better results than 32×32.

Table 3.6 Quantitative evaluation on the test set of KITTI dataset [56] using the split of Eigen et al. [105] with maximum depth set to 80m. Methods with * run post-processing [95].

				Lower is better ⇓				Higher is better ⇑		
Method	Resolution	Params	MCU	Abs Rel	Sq Rel	RMSE	RMSE log	α <1.25	$\alpha < 1.25^2$	$\alpha < 1.25^3$
Eigen et al. [105] Fine	172 × 576	54.2M	No	0.203	1.548	6.307	0.282	0.702	0.890	0.958
Liu et al. [128]	-	40.0M	No	0.201	1.584	6.471	0.273	0.680	0.898	0.967
Zhou et al. [109]	128 × 416	34.2M	No	0.198	1.836	6.565	0.275	0.718	0.901	0.960
MonoDepth [95] ResNet50*	256 × 512	48.0M	No	0.114	0.898	4.935	0.206	0.861	0.949	0.976
3Net [113] ResNet50*	256 × 512	65.0M	No	0.111	0.849	4.822	0.202	0.865	0.952	0.978
MonoDepth2 (S) [58]	192 × 640	11.0M	No	0.109	0.873	4.960	0.209	0.864	0.948	0.975
DSVO [129]*	256 × 512	96.2M	No	0.097	0.734	4.442	0.187	0.888	0.958	0.980
MonoResMatch [114]*	384 × 1280	42.5M	No	**0.096**	**0.673**	**4.351**	**0.184**	**0.890**	0.961	**0.981**
DepthHints [130]*	320 × 1024	34.5M	No	**0.096**	0.710	4.393	0.185	**0.890**	**0.962**	**0.981**
PyD-Net [122]	256 × 512	1.9M	No	0.146	1.291	5.907	0.245	0.801	0.926	0.967
μPyD-Net	48 × 48	0.1M	Yes	0.193	2.312	6.952	0.277	0.735	0.890	0.953
μPyD-Net	32 × 32	0.1M	Yes	0.215	2.395	7.252	0.301	0.696	0.866	0.939

Table 3.7 Quantitative evaluation on the test set of KITTI dataset [56] using the split of Eigen et al. [105] with maximum depth set to 80m.

			Lower is better ⇓				Higher is better ⇑		
Method	Resolution	Params	Abs Rel	Sq Rel	RMSE	RMSE log	α <1.25	$\alpha < 1.25^2$	$\alpha < 1.25^3$
MonoDepth2 [58]	↓ 32 × 32 ↑	34.5M	**0.188**	**1.724**	7.447	0.318	0.686	0.869	0.938
MonoResMatch [114]	↓ 32 × 32 ↑	42.5M	0.191	2.103	**6.670**	**0.279**	**0.742**	**0.896**	**0.953**
μPyD-Net	32 × 32	0.1M	0.215	2.395	7.252	0.301	0.696	0.866	0.939

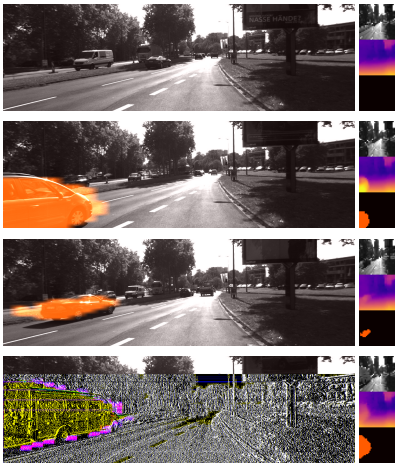


Fig. 3.12 Qualitative results concerning traffic monitoring. For each example, we show the high-resolution frame, followed by 32×32 images processed by μPyD-Net.

To further support that μPyD-Net is effective at extracting most of the knowledge available from low-resolution images, we show the performance achieved by two state-of-the-art networks, respectively MonoDepth2 [58], and MonoResMatch [114] when processing 32×32 images. Being these latter not able to process such tiny images because of architectural limitations, we simulate low-resolution images by downsampling the inputs to 32×32 ($\downarrow 32 \times 32$) and then upsampling ($\uparrow$) them back to the original resolution. Tab. 3.7 collects the outcome of this evaluation, showing how μPyD-Net places in between the two competitors for most metrics (*i.e.*, Sq Rel, RMSE, RMSE log, $\alpha < 1.25$, $\alpha < 1.25^3$) although counting two order of magnitude less parameters. This supports the fact that μPyD-Net itself is enough to extract most of the information available from low-resolution content while keeping low complexity. This latter property is crucial for deployment on the target microcontrollers, over which MonoDepth2 and MonoResMatch parameters alone would not fit into the available memory.

We stress the fact that the farther points in the scene are those most affected by the degradation introduced by processing tiny images since each pixel senses a larger portion of the real scene. Therefore, we will assess the accuracy of μPyD-Net when sensing at different ranges the scenes included in the datasets. Tab. 3.8 reports a detailed comparison between μPyD-Net and its optimized counterpart considering different depth ranges, from 0 m to 15, 25, 50 and 80 m. The upper part shows the results obtained by μPyD-Net

Table 3.8 Evaluation of μPyD-Net and quantized variants at different ranges. Comparison with state-of-the-art [114] on the same ranges.

Res.	Range	Precision	Lower is better ⇓ Abs Rel	Sq Rel	RMSE	RMSE log	Higher is better ⇑ α <1.25	$\alpha < 1.25^2$	$\alpha < 1.25^3$
32 × 32	0-80m	float32	0.215	2.395	7.256	0.302	0.695	0.865	0.939
		int8	0.498	11.712	15.133	0.656	0.439	0.714	0.850
		int8-ft	0.219	2.478	7.379	0.307	0.687	0.861	0.937
	0-50m	float32	0.206	1.865	5.710	0.287	0.710	0.875	0.946
		int8	0.421	6.004	9.769	0.529	0.461	0.751	0.889
		int8-ft	0.209	1.928	5.809	0.292	0.702	0.872	0.943
	0-25m	float32	0.172	0.929	3.155	0.238	0.764	0.910	0.965
		int8	0.308	2.023	4.717	0.354	0.546	0.854	0.944
		int8-ft	0.174	0.939	3.179	0.240	0.758	0.908	0.964
	0-15m	float32	0.136	0.448	1.800	0.189	0.822	0.939	0.979
		int8	0.219	0.761	2.438	0.248	0.718	0.919	0.972
		int8-ft	0.138	0.457	1.815	0.191	0.817	0.938	0.978
48 × 48	0-80m	float32	0.193	2.308	6.943	0.277	0.736	0.890	0.953
		int8	0.322	5.919	10.648	0.394	0.615	0.837	0.925
		int8-ft	0.193	2.252	6.922	0.276	0.735	0.890	0.954
	0-50m	float32	0.182	1.685	5.308	0.261	0.751	0.901	0.960
		int8	0.285	3.329	7.266	0.343	0.636	0.860	0.942
		int8-ft	0.182	1.656	5.299	0.260	0.750	0.901	0.960
	0-25m	float32	0.149	0.748	2.843	0.212	0.804	0.932	0.975
		int8	0.220	1.208	3.629	0.261	0.716	0.914	0.967
		int8-ft	0.150	0.744	2.854	0.212	0.803	0.931	0.975
	0-15m	float32	0.118	0.348	1.601	0.168	0.856	0.956	0.986
		int8	0.166	0.501	1.967	0.199	0.821	0.948	0.983
		int8-ft	0.119	0.347	1.608	0.168	0.855	0.955	0.986
[114]	0-80m		0.096	0.673	4.351	0.184	0.890	0.961	0.981
[114]	0-50m		0.092	0.504	3.336	0.174	0.899	0.965	0.984
[114]	0-25m		0.078	0.242	1.799	0.141	0.925	0.977	0.990
[114]	0-15m		0.067	0.119	1.027	0.111	0.949	0.986	0.994

Table 3.9 Quantitative evaluation on Make3D dataset [107].

		Lower is better ⇓			
Method	Resolution	Abs Rel	Sq Rel	RMSE	RMSE log
MonoDepth [95] ResNet50	256 × 512	0.451	7.299	10.139	0.223
3Net [113] ResNet50	256 × 512	0.407	5.060	8.558	0.203
MonoDepth2 [58]	192 × 640	0.375	3.694	**8.218**	0.204
MonoResMatch [114]	384 × 1280	0.375	4.072	8.859	0.213
DepthHints [130]	320 × 1024	**0.350**	**3.385**	8.242	**0.200**
PyD-Net [122]	256 × 512	0.510	9.106	10.538	0.225
μPyD-Net	48 × 48	0.531	7.607	9.726	0.226
μPyD-Net	32 × 32	0.607	10.687	10.252	0.237

processing 32×32 images and in the middle processing with the same network 48×48 images. At the very bottom of the same table, we also report for comparison results yielded by state-of-the-art [114]. We can notice in general how, independently of the input resolution and evaluation range, introducing the quantization dramatically drops the performance of μPyD-Net (float32 vs. int8 entries), as already observed in [96]. However, by fine-tuning the model after quantization (int8-ft entries), the original performance is restored for most metrics and sometimes even improved. Focusing on how the metrics change across the different evaluation ranges, we can perceive how, on nearby measurements, the gap between μPyD-Net and much more complex state-of-the-art [114] gets lower. For instance, by looking at the RMSE metric, we can observe how the difference in terms of average error is about 3 when considering the full evaluation range 0-80 m, while it drops to about 0.6 and 0.8 respectively for 48×48 and 32×32 images when dealing with the 0-15 m range. This behavior suggests that μPyD-Net might not be particularly suited for long-range depth measurements. However, close-range depth-sensing provides a valid alternative when a low-power budget is paramount. For instance, Fig. 3.12 shows a qualitative example of a traffic monitoring system processing images from the KITTI dataset [56] downsampled to 48×48 resolution. We show four images acquired from a static point of view to simulate a monitoring camera placed on a crossroad. For each one, we report on the right their downsampled counterpart, the estimated disparity map sourced by μPyD-Net and a segmentation map detecting objects on the scene over imposed to the original KITTI image. To this aim, given the depth layout estimated for the

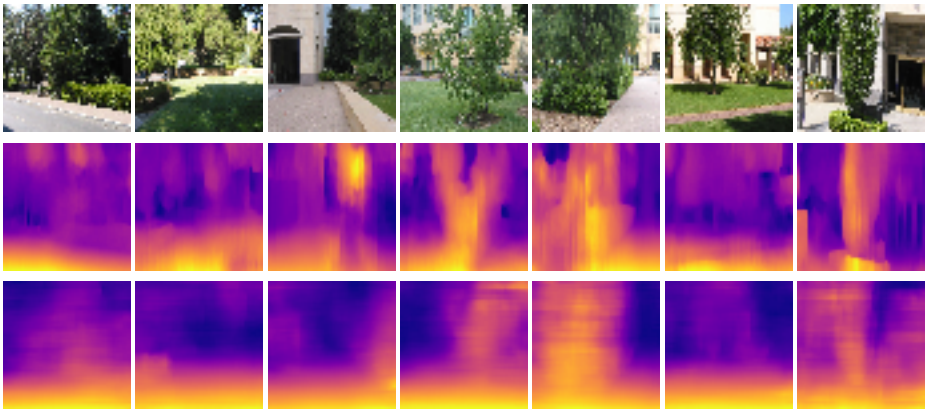


Fig. 3.13 Qualitative results on Make3D. From top to bottom, reference images, inverse depth maps by MonoResMatch [114] and by 48×48 μPyD-Net.

empty scene (i.e., in the absence of vehicles) estimated by μPyD-Net, a simple change-detection algorithm in the depth domain is sufficient to detect nearby cars reliably. Although this application allows for simple traffic monitoring, the depth cue provided by μPyD-Net can be exploited for other purposes (e.g., 3D tracking) and replace other sensors. Therefore, such information could be used in place of other sensors or to enrich other image-based cues such as object detection or semantic segmentation.

In order to assess the generalization properties, in Tab. 3.9 we report results on the Make3D dataset [107] following the evaluation proposed in [58], on a center crop of 2×1 ration and without applying median scaling (not required when training on stereo pairs). We point out how state-of-the-art networks suffer from huge drops when moved to unseen environments as well. μPyD-Net can still provide meaningful predictions, very close to those by MonoDepth when running at 48×48. Fig. 3.13 show some qualitative examples, showing in particular how the coarse disparity maps by μPyD-Net are often less affected by artifacts with respect to the predictions by MonoResMatch.

Evaluation – Hardware-related metrics

The shift from high-performance GPUs to ultra-low-power MCUs encompasses the evaluation of hardware-related metrics besides accuracy, *i.e.* latency and memory, in order to assess the portability and efficiency. As shown in Tab. 3.6, the number of parameters of standard monocular networks exceeds by far the

Table 3.10 Extra-functional metrics of μPyD-Net at different input resolutions on the NUCLEO-F767ZI board.

Resolution	RAM	Execution Time
32×32	208 kB	290 ms
48×48	337 kB	651 ms

memory constraints of commercial MCUs, preventing the deployment on the edge and therefore a direct comparison with μPyD-Net. For this reason, this section focuses on the hardware characterization of μPyD-Net, demonstrating that the adopted architectural choices are mandatory to guarantee compliance with the limited resources of the hosting system.

Tab. 3.10 reports the hardware-related metrics measured at run-time on the NUCLEO-F767ZI board: RAM usage and execution time (averaged on 100 inference runs). The proposed μPyD-Net reaches a throughput of 3.4 FPS, which can be considered a remarkable result considering the limited power budget of the adopted device. Moreover, the collected results demonstrate that input resolution is an effective knob in the accuracy-latency-memory space: a resolution of 32×32 enables 38% of memory savings and 2.24× higher throughput compared to 48×48. This comes at the cost of some accuracy loss (as already shown in Tab. 3.8). However, this might be a false problem as errors can be masked by subsequent processing stages. The resolution is a design choice indeed, and it should be weighted depending on the requirements of the downstream application.

Even though the limited computational resources of the hosting MCU prevent real-time processing even for such a compact network, the measured performance meets the requirements of the applications described in Section 3.4.1. However, if higher power and area budgets are available, μPyD-Net can be ported to more powerful systems and its application extended to other use-cases. To assess the scalability of μPyD-Net from the IoT to the embedded segment, we tested its performance on the mobile CPUs (ARM Cortex-A53) adopted in our previous work [96]. In this system, μPyD-Net processes up to 320 frame/s, a 94× boost that comes at the cost of 10× power consumption ($\sim$4 W).

It might seem like the high efficiency brought by μPyD-Net is simply due to the input rescaling, and hence our proposal may seem a relatively naive

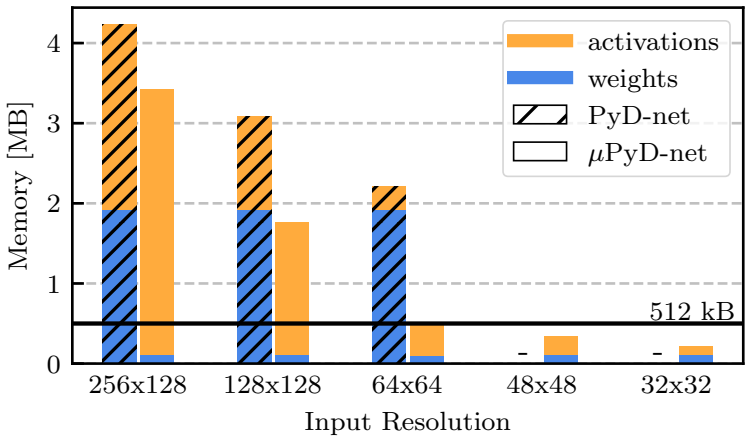


Fig. 3.14 Memory breakdown of PyD-Net and μPyD-Net at different input resolutions. The dash (-) indicates that the resolution is not compliant with the network topology.

approach. A more detailed analysis reveals that the design of μPyD-Net goes beyond this simplistic analysis. On the one hand, it is correct that a lower input space reduces the memory footprint as all the inner feature maps get intrinsically smaller. On the other hand, what makes μPyD-Net smaller and faster, hence less energy-hungry and able to fit tiny MCUs with marginal accuracy loss, lies in the topology of the network. Input resolution scaling alone is not enough, but, when jointly applied on the structure of μPyD-Net it enables design options that would not be possible otherwise. Like other pyramidal architectures, μPyD-Net applies a coarse-to-fine strategy where information is processed hierarchically. Since features of higher semantic level are inferred layer-by-layer traversing the pyramid bottom-up, it is intuitive to understand that the lower the resolution of the input image, the lower the number of layers needed to achieve a certain accuracy. This is a general trend also recognized in other CNNs, but it has a much higher impact on the specific case of μPyD-NetẆith smaller inputs, it is possible to compress the topology by reducing the number of encoders and decoders, not just their size, thus achieving aggressive RAM reduction.

To support this analysis, the bar chart in Fig. 3.14 shows the memory footprint vs. input resolution of PyD-Net (hatched bars) and μPyD-Net (plain bars), both quantized to 8-bit; the comparison is made by splitting the contributions of weights (blue) and inner features (orange). The horizontal black line marks the RAM constraint (512 kB). For both the networks, the

dimensionality of the internal activations is re-scaled according to the input size. It is worth noticing that the minimum input resolution of PyD-Net is 64×64 since the input image is down-sampled by a factor of 2^6 across the pyramidal encoders. For such reason, the results below are not reported. We can infer the following considerations. First, PyD-Net runs out of space and cannot fit into MCUs because the size of the weights does not scale with the input resolution. Even using the smallest input size (*i.e.* 64×64), the weight storage is about 2 MB, namely $4\times$ the RAM on-board. Second, μPyD-Net shows an activations/weights ratio larger than PyD-Net. For instance, with the highest input resolution (256×128) the RAM taken by the features significantly increases, from 2.3 MB (PyD-Net) to 3.2 MB (μPyD-Net). The reason is that, in PyD-Net, the size of the feature maps processed by the topmost decoder, which is the most energy-hungry layer, is half of the input resolution, while in μPyD-Net it is not. Therefore, μPyD-Net is less suited for high resolution. However, as inputs are re-scaled to 48×48, the activation footprint scales well, meeting the memory constraint. Working with 32×32 images ensures some free space for other background applications or tasks.

These findings support our claim: μPyD-Net does work not just because of the lower cardinality of the input space but precisely because it has been tailored to adapt to the requirements of tiny applications.

3.4.6 Discussion

Depth is of paramount importance in many practical computer vision applications and the terrific results recently achieved by monocular methods have dramatically increased the interest in this topic. Unfortunately, in most cases, these methods rely on high-end GPGPUs or sufficiently capable embedded devices, precluding their practical deployment in those application contexts involving ultra low-power devices such as MCUs. Thus, in this section, we proposed a two-fold strategy to enable monocular depth estimation on MCUs. First, we designed an ultra-lightweight Convolutional Neural Network based on a pyramidal architecture, which is trained in a semi-supervised manner leveraging proxy-supervision obtained through a conventional stereo algorithm. Then, we quantized the network to an 8-bit fixed-point representation, and we mapped the high-level description of the network to low-level routines optimized for the

target hardware. Exhaustive experimental results and an in-depth evaluation performed on an Arm Cortex-M7 MCU confirm that obtaining monocular depth cues is feasible even on devices characterized by ultra low-power constraints.

3.5 Conclusions

This chapter focused on how to build *fast* CNNs targeting *low-power* devices. Specifically, it introduced a comprehensive design and optimization framework to accelerate the inference of quantized, lightweight CNNs on ARM Cortex-A CPUs and ARM Cortex-M MCUs. The optimization framework combines a hardware-friendly fixed-point quantization method with integer neural kernels custom-tailored to the target platform. The first part of the chapter discussed the acceleration of PyDNet, a lightweight CNN capable of achieving close to state-of-the-art accuracy in the challenging task of monocular depth estimation. The experimental evaluation performed on a multi-core ARM Cortex A53 CPU proves the efficiency of the proposed framework, as the 16-bit and 8-bit quantized versions of PyDNet suffer from negligible to marginal accuracy loss compared to the floating-point version while achieving remarkable latency and memory savings. The second part of the chapter introduced the design and development of μPyD-Net, a lightweight Convolutional Neural Network specifically designed to perform monocular depth estimation on MCUs. The experimental assessment showed that, despite meeting the strict constraints of tiny MCUs, μPyD-Net achieves a depth accuracy comparable to seminal works on the KITTI dataset. As the primary outcome, this chapter demonstrated that combining a hardware-aware neural design process, an efficient quantization method, and computational kernels custom-tailored to the target architecture can remarkably improve the efficiency of challenging CNN inference tasks on low power devices.

Chapter 4

Energy-Quality Scalable Convolutional Neural Networks

4.1 Introduction and Motivation

As discussed in previous chapters, the optimization of CNNs for low-power embedded devices is a well-established problem in many application contexts. The recent literature has proposed several solutions, from design methodologies of hardware-aware neural architectures [15, 84] to model compression strategies, like weight pruning [35] and quantization [40], and low-level code optimizations [145]. Although such methods have enabled a remarkable reduction of the complexity of a CNN, they allow embedded designers to tune the optimization process for the worst-case constraints, producing a *static* CNN capable of working in only one performance-quality point. An embedded system relying on such a *static* CNN spends the same computational effort in all circumstances, neglecting the changes in external conditions, quality-of-service demands, and the requirements of other applications running concurrently on the same device. In this context, satisfying the worst-case constraints corresponds to either reducing the CNN accuracy or reducing its performance.

An alternative solution relies on what is known in the approximate computing field as *"energy-quality" scaling*. By leveraging the error resilience of many real-life applications, the quality-of-result can be gracefully degraded at run time to achieve higher energy efficiency, depending on the specific task,

the external context, or the battery level. In this scenario, the worst-case constraints can be satisfied only when needed, optimizing for energy efficiency in all other cases. For instance, a mobile surveillance system can reduce the prediction quality to consume less battery on average and switch to a more accurate but more energy costly mode only when something suspicious is detected. In a depth estimation application, a less accurate estimate could still be sufficient to accomplish simple tasks, such as object and people counting, action recognition, and vehicle detection. However, the system can switch to an accurate configuration for a more precise 3D scene reconstruction or when the scene is particularly cluttered and "complex". More in general, when the scene analyzed is "easy" and requires less effort to be understood, the computational effort of the system should be scaled accordingly to release system resources or to reduce energy consumption. Beyond these context-driven speculations, the run time environment of the system could force the inference process of a CNN to run under more stringent latency and energy constraints in certain intervals of time, to allocate more time and resources for other applications running concurrently. Therefore, the availability of energy-quality (EQ) scalable CNNs capable of efficiently trading-off at run time accuracy with computational cost represents an extremely valuable tool in embedded applications.

Building an energy-quality scalable CNN encompasses the availability of proper knobs that can be tuned at run time to set the most appropriate working point. Several knobs have been proposed in the literature: to mention a few, width modulation [15, 24] and layer skipping [22, 23] play with the topology of the network, resolution scaling [15, 146] with the size of the intermediate feature maps, precision scaling [147, 148] with the complexity of the arithmetical operations. Nevertheless, previous works have been focused only on image classification tasks based on CNNs [149], but not, or very marginally, in more complex applications, such as monocular depth estimation. Moreover, how to implement an energy-quality scalable system on tiny devices powered by MCUs is still an open research challenge, as the scalable knobs must be compatible with the minimal memory resources available on device, namely, $1-2$ MBs of FLASH and tens to hundreds of KBs of RAM.

This chapter first reviews the main knobs proposed in the literature to implement energy-quality scalable CNNs (Section 4.2). Then, it introduces the design and characterization of an efficient dynamic system for energy-

quality scalable depth-sensing called EQPyD-Net (Section 4.3). Finally, the last part of the chapter presents an end-to-end optimization flow comprising a training methodology, a novel compressed sparse storage format, and purposely built sparse operators to deploy energy-quality scalable CNNs on tiny MCUs (Section 4.4).

The content of this chapter is an extended and improved version of our previous publication found in [98].

4.2 Energy-Quality Scaling Knobs

This section reviews the most common knobs proposed in the literature to scale the energy-quality working point of a CNN at run time.

Resolution scaling

High-resolution features help the model achieve high prediction accuracy at the cost of a remarkable increase in the computational cost. Therefore, tuning the resolution depending on the actual constraints of the system represents an effective way to scale the energy-quality point of a CNN. Resolution scaling can be achieved by changing the spatial resolution of the network inputs [15] or, in the case of multi-scale architectures [146], by selecting the branches of the network to process.

Width Modulation

Static solutions scale uniformly the number of filters of the convolutional layers of the network through a scaling ratio, called the width-multiplier [15]. More advanced solutions [24] share the weights across the different configurations (i.e., for different values of the width-multiplier) to avoid the overhead of storing multiple weight-sets.

Depth Modulation

The depth of the network, i.e., the number of layers traversed during the inference pass, can be selected at run time by the user or by the system. Alternatively, the depth can be modulated automatically by attention modules, gating blocks [22], or early-exit branches [23]. Multi-scale networks like PyD-Net combine resolution scaling with layer skipping.

Arithmetic precision

The data-type of weights and activations can reduce the computational effort and alleviate the memory bandwidth requirements [40]. The data-type scaling can be performed per layer or per network. The latter is usually preferred in the case of general-purpose cores, such as those targeted in this work, thanks to its simple implementation.

4.3 Energy-Quality Scalable Monocular Depth Estimation on Embedded CPUs

This section introduces the design and characterization of an efficient energy-quality scalable system for depth sensing. When designing an EQ system, the first implementation choice is the selection of the underlying CNN model that has to be enhanced with dynamic features. In the case of a monocular depth estimation pipeline, PyD-Net [96] is an excellent candidate. Indeed, its modular structure allows the resolution of the predicted depth maps to be set at run time, skipping the most demanding computations, hence saving energy when running at lower output scales. Moreover, its tiny memory footprint enables the storage of multiple weight-sets at different arithmetic precisions. These different configurations can then be dynamically selected at run time without any latency overhead to tune the energy-accuracy working point of the system. Therefore, building upon our previous work [96] described in Sec. 3.3, we present an energy-quality scalable system named EQPyD-Net.

The dynamic energy-quality trade-off attainable with EQPyD-Net is the result of the following optimizations orchestrated across all levels of the design and deployment stack:

- at training time, the adoption of a multi-scale self-supervised training procedure to infer high-quality depth maps at five different resolutions;
- at optimization time, the use of a quantization procedure to convert PyD-Net into 16- and 8-bits fixed-point configurations;
- at run time, the adoption of energy-proportional neural kernels for fixed-point tensor operators custom-tailored for the ARMv7a architecture.

A thorough assessment of EQPyD-Net on a COTS system powered by an ARM Cortex-A53 CPU shows that it can cover a wide range of energy-quality space with a Pareto front comprising five operating points. In fact, on the KITTI dataset, EQPyD-Net can be as accurate as 82.2% processing 0.4 Frame/J, or it can save up to 92.6% of energy savings with only 6.1% of accuracy loss. Nevertheless, EQPyD-Net requires only 5.2 MB for the weights and 38.3 MB (worst-case) for the processing, representing thus a promising solution for embedded computer vision applications.

4.3.1 Training and Optimization Flow for EQPyD-Net

Neural Network Architecture

As we adopt the same neural architecture of our previous work, the reader could refer to section 3.3.2 for a more detailed description. For the sake of clarity, Fig 4.1 shows the abstract view of PyD-Net. In this section, we only highlight the key elements that make PyD-Net an excellent candidate for an energy-quality scalable system. Specifically, PyD-Net is characterized by the peculiar combination of a super lightweight encoder with decoder branches that require a computational effort proportional to the quality demand. Such a design choice is motivated by observing that the encoder is always processed regardless of the final resolution; thus, it is designed to be as fast as possible. On the other hand, the high-resolution decoders are active only when a high

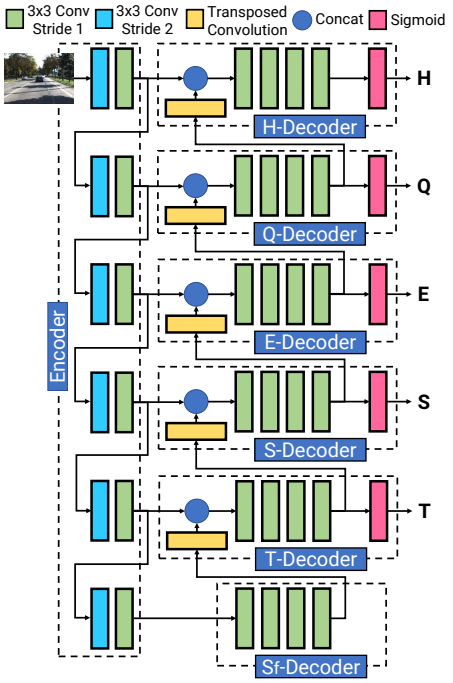


Fig. 4.1 PyD-Net architecture. H stands for $\frac{1}{2}$ of the input resolution, Q for $\frac{1}{4}$, E for $\frac{1}{8}$, S for $\frac{1}{16}$, T for $\frac{1}{32}$.

output resolution is set as the network output, and so, they are designed to be energy-quality proportional. For comparison with common architectures used as the encoder in other monocular networks, Tab. 4.1 reports the number of MAC operations and the number of parameters of the encoder of PyD-Net and of VGG16, ResNet-18, and MobileNetV2. The encoder of PyD-Net requires at least 3.4x less space and at least 1.93x less MAC operations.

The Training Procedure

We follow the state-of-the-art methodology described in sec. 3.3.2 to train our network in a self-supervised manner, assuming that stereo data is available during training. The model is trained to minimize a multi-scale combination of appearance, smoothness, and consistency loss terms as proposed in [58]. Specifically, we compute each single-scale loss signal at full resolution by upsampling each estimated depth d_s at scale s to the full resolution map $d\uparrow_s$. Then, the final loss signal $\mathcal{L}$ is obtained as the sum of each single-scale loss $\mathcal{L}^{d\uparrow_s}$. This approach differs from what we did in our previous work, where

Table 4.1 Number of parameters (# Params) and multiply&accumulate operations (# MACs) of the most common encoders and PyD-net encoder.

Network	**# Params (M)**	**# MACs (G)**
VGG16	138.36	40.37
ResNet-18	11.69	4.76
MobileNet v2	3.50	0.85
PyD-net Encoder	1.02	0.44

the I^l, I^r are first downsampled at each scale to compute $\mathcal{L}^{d_s}$ [95, 122], and then used to provide supervision at the different decoders. Fig. 4.2 gives a preliminary qualitative comparison between the results obtained by the original training adopted in [122] for PyD-Net and the new one deployed in this work for EQPyD-Net. The figure shows that the former can only provide meaningful estimations down to $\frac{1}{8}$ of the input resolution while EQPyD-Net allows for scene understanding even at a resolution as low as $\frac{1}{32}$ of the input.

Optimization Flow

We adopt the same optimization flow presented in section 3.3.3. Here, we highlight only the two main benefits of precision scaling in reducing energy consumption. First, at lower precision, the number of elements fitting in the lower levels of the memory hierarchy, namely, caches and register files, doubles, reducing the data movement across the hierarchy and so the energy consumption. Second, the number of MACs processed by a single instruction doubles from 2 (16-bit) to 4 (8-bit), enabling faster processing.

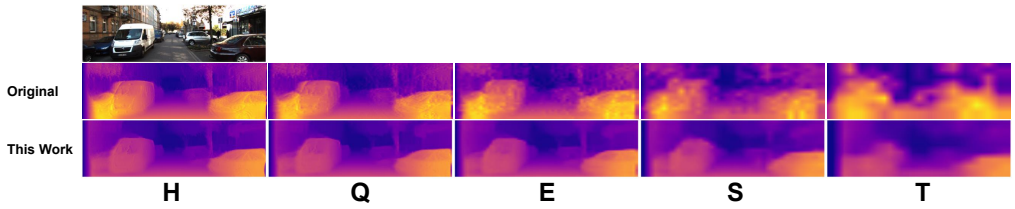


Fig. 4.2 Depth estimated at different output resolutions for an input taken from the KITTI dataset. On top [122], on the bottom EQPyD-Net.

4.3.2 Experimental Results

This section aims at assessing the performance of EQPyD-Net, according to several figures of merit, both functional, namely, error and accuracy metrics, and extra-functional, namely, energy consumption and memory footprint. The analysis reported at the end of the section describes in-depth the effect of the arithmetic and neural topology knobs on the energy-quality scalability of EQPyD-Net.

Training set-up

We extensively tested the energy-quality scalability of EQPyD-Net on the KITTI raw dataset [56], a widely adopted dataset for depth estimation. It comprises 42K rectified stereo image pairs framing driving scenarios. It was produced starting from 61 video sequences collected by a driving car. Besides stereo pairs, the dataset also contains metric depth measurements produced by a LiDAR sensor mounted on the car. Following previous works [105], [95], We split the dataset into two subsets according to the Eigen split [105], i.e., the set of 61 sequences is split into 29 and 32 sequences, and 697 frames are picked from the first split and 22600 from the second group for test and training purposes respectively.

To train the network, we adopted the schedule described in [122] to compare with the original PyD-Net version [122, 96]. Specifically, we first pre-trained the network for 50 epochs on the CityScapes dataset [140], then we trained for 200 epochs on the Eigen training split. For both both cases, we adopted a batch size of 8 images with resolution 512×256. We used the Adam optimizer [127] with $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\varepsilon = 10^{-8}$, a learning rate of 10^{-4} for the first 60% epochs, halved every 20% epochs until the end of the training process. During training, we performed the following augmentations with a 50% probability to be applied: random horizontal image flipping, random color augmentation in terms of gamma correction, brightness modification, and color shifting. Gamma, brightness, and color shift values are sampled from a uniform distribution in [0.8,1.2], [0.5,2.0], and [0.8,1.2] ranges respectively.

The post-quantization fine-tuning was performed for 5 epochs with the Adam optimizer, learning rate 1.0e−5 on the Eigen training split.

Table 4.2 Error metrics and accuracy scores on the KITTI raw data using the Eigen split [105] at different scales and precision options. For each resolution, the first row refers to PyD-Net trained with the single-scale loss [96]. The best results at each resolution are highlighted in bold, while the absolute bests in red.

			Lower is better		Higher is better		
Config.	Abs Rel	Sq Rel	RMSE	RMSE log	a_1	a_2	a_3
H@FP32 [96]	0.146	1.298	5.859	0.241	80.2%	92.7%	96.8%
H@FP32	**0.135**	1.154	**5.550**	**0.234**	**82.1%**	**93.2%**	**96.9%**
H@FX16	0.136	1.157	5.556	0.235	**82.1%**	**93.2%**	**96.9%**
H@FX8	0.200	1.820	6.797	0.329	69.8%	86.9%	93.2%
H@FX8-ft	0.141	**1.152**	5.634	0.239	80.9%	92.9%	**96.9%**
Q@FP32 [96]	0.149	1.350	6.128	0.246	79.5%	92.3%	96.6%
Q@FP32	**0.135**	**1.134**	**5.505**	**0.233**	82.1%	**93.3%**	**97.0%**
Q@FX16	**0.135**	1.136	5.506	**0.233**	**82.2%**	**93.3%**	**97.0%**
Q@FX8	0.196	1.918	6.735	0.297	72.9%	88.8%	94.6%
Q@FX8-ft	0.146	1.164	5.608	0.241	80.5%	93.0%	96.9%
E@FP32 [96]	0.162	1.699	7.141	0.266	76.8%	90.7%	95.9%
E@FP32	**0.137**	**1.151**	**5.546**	**0.233**	**81.7%**	**93.1%**	**97.0%**
E@FX16	**0.137**	1.153	**5.546**	**0.233**	**81.7%**	**93.1%**	**97.0%**
E@FX8	0.264	2.964	8.252	0.324	61.1%	86.8%	94.4%
E@FX8-ft	0.145	1.157	5.675	0.241	80.1%	92.8%	96.9%
S@FP32 [96]	0.221	2.768	8.960	0.347	64.3%	84.5%	92.5%
S@FP32	**0.143**	**1.235**	**5.679**	**0.235**	**80.4%**	**92.8%**	**97.0%**
S@FX16	**0.143**	1.238	5.680	**0.235**	**80.4%**	**92.8%**	**97.0%**
S@FX8	0.215	2.176	7.062	0.291	70.6%	89.0%	95.2%
S@FX8-ft	0.155	1.335	5.843	0.240	79.6%	92.7%	**97.0%**
T@FP32 [96]	0.416	9.184	12.384	0.502	45.6%	71.8%	84.7%
T@FP32	**0.165**	**1.514**	**5.990**	**0.243**	**77.8%**	**92.2%**	**97.0%**
T@FX16	**0.165**	1.517	5.993	**0.243**	**77.8%**	**92.2%**	**97.0%**
T@FX8	0.199	1.982	6.816	0.285	70.6%	89.1%	95.5%
T@FX8-ft	0.178	1.667	6.161	0.249	76.1%	92.1%	**97.0%**

Device set-up

We adopted the Raspberry Pi 3B (RPI3B) powered by the BCM2837 System-on-Chip by Broadcom, hosting a quad-core ARM Cortex-A53 CPU and $1GB$ of DRAM. The CPU can run at a maximum clock frequency of 1.2 GHz. The Q.Neural Kernels (see Section 3.3.3) were integrated into the ARM Compute Library (ACL) [54] version 18.05, which collects the floating-point operators used as baselines in our analysis. The code was cross-compiled with *gcc-linaro* toolchain version 6.4.0-2018.05. We adopted Ubuntu Mate 16.04 (32-bit) as the operating system. We used Google benchmark version 1.5.0 [150] to collect the performance statistics of the inference stages at the maximum operating frequency.

Functional Metrics

Tab. 4.2 reports a quantitative assessment of EQPyD-Net under the different error measures (Abs Rel, Sq Rel, RMSE, RMSE log) and accuracy scores (a_1, a_2, a_3), assuming a 80 m cap distance [95] (see sec. 3.2 for further details). The table is organized in five sections, one for each resolution (H, Q, E, S, T); results are reported for different data types (32-bit floating-point, 16- or 8-bit fixed-point, referred to as FP32, FX16, and FX8) with fine-tuning (-ft) performed post-quantization. The first row of each section reports the metrics of the original version of PyD-Net [96].

The collected results show the higher accuracy achieved by EQPyD-Net thanks to the multi-scale loss. Scaling down to S, EQPyD-Net outperforms the original version at H resolution in all metrics, obtaining better results while being faster and requiring fewer resources. Moving to T, it still achieves competitive accuracy ($a_1 = 77.8\%$) with only 4.3% loss related to H, whereas, in the original version, the accuracy drops down to 45.6%. The benefits can also be perceived qualitatively from Fig. 4.3. Moreover, as the Q configuration of EQPyD-Net achieves better results than H for all metrics, we removed the H-decoder from the network at deployment time with substantial energy and memory savings, as will be described in the following subsections.

Tab. 4.2 shows that the FX16 configuration achieves equal or better results than the FP32, whereas switching to FX8 causes a substantial accuracy drop (the worst case is -20.6% for a_1 in E@FX8). Luckily, fine-tuning the network reduces the accuracy gap with FX16 and FP32, as shown qualitatively in Fig. 4.3. The 3D structure of the scene is well preserved even at the lowest scales independently from the adopted data type, with a slight deterioration at T resolution only. Moreover, it can be noticed how the FX8 configuration after fine-tuning performs similarly to standard FP32.

Overall, these findings demonstrate the efficiency of the multi-scale loss and the effectiveness of the quantization procedure for all output resolutions.

Comparison to state-of-the-art techniques

In Tab. 4.3, we compare the accuracy of our best configuration Q@FP32 with the accuracy of other state-of-the-art monocular depth strategies. Compared to the PyD-Net, the new training procedure based on the multi-scale loss further improves the depth estimation accuracy closing the gap with larger self-supervised models such as MonoDepth [95], MonoDepth2 [58], and 3Net [113]. On the other hand, the methods leveraging other forms of supervision such as SGM as done in DepthHints [130] and MonoResMatch [114], still produce better depth estimations. However, the higher accuracy comes at the cost of a huge computational and memory burden, making the use of high-performance CPUs or GPUs mandatory. EQPyD-Net, instead, is highly compact, and so it represents an excellent candidate for embedded applications.

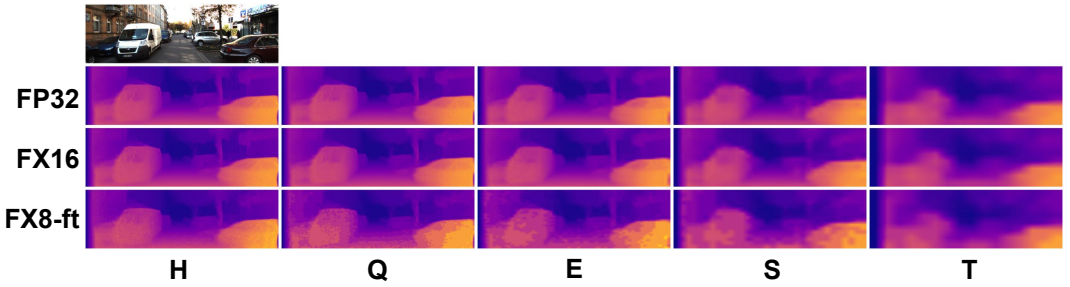


Fig. 4.3 Depth images obtained for each value of precision and output resolution for an input taken from the KITTI dataset. The last row illustrates depth images inferred after fine-tuning (-ft).

Table 4.3 Quantitative evaluation of KITTI test set using the split of Eigen et al. [105] with maximum depth set to 80m.

		Lower is better				Higher is better		
Method	SGM	Abs Rel	Sq Rel	RMSE	RMSE log	δ <1.25	$\delta < 1.25^2$	$\delta < 1.25^3$
DepthHints [130]	✓	**0.096**	0.710	4.393	0.185	**0.890**	**0.962**	**0.981**
MonoResMatch [114]	✓	**0.096**	**0.673**	**4.351**	**0.184**	**0.890**	0.961	**0.981**
MonoDepth [95]	-	0.114	0.898	4.935	0.206	0.861	0.949	0.976
3Net [113]	-	0.111	0.849	4.822	0.202	0.865	0.952	0.978
MonoDepth2 [58]	-	0.109	0.873	4.960	0.209	0.864	0.948	0.975

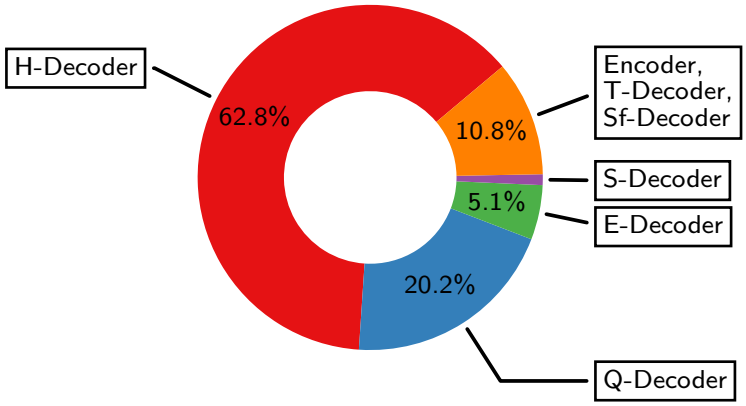


Fig. 4.4 Energy breakdown of different modules of EQPyD-Net at FX16. Similar values have been observed also for FX8.

Energy-Quality Scaling

This section presents an extensive characterization of the energy-quality characteristics of EQPyD-Net. Specifically, the aim is to dissect the effect of the adopted scaling knobs, namely, resolution and precision, on energy and accuracy scaling. As the quality metric, we selected the a_1 score, which is commonly adopted to assess the prediction accuracy in the case of CNNs for monocular depth estimation [120]. To evaluate the energy consumed during inference, we measured the average Frame/J over 100 runs, assuming a constant power consumption of 3.5 W. Although this is a worst-case assumption, as the adopted scaling knobs could slightly lower the average power consumption, the latency scaling represents the main factor affecting the energy efficiency on a general-purpose CPU.

Fig. 4.4 shows the energy breakdown of the different modules composing EQPyD-Net. Although the pie-chart reports only the number for the FX16 configuration, similar percentages hold for the other precision options. The main advantage of EQPyD-Net lies in the limited contribution (only 10.8%) of the modules that are always executed regardless of the selected resolution (orange area). Thanks to its lightweight pyramidal encoder, EQPyD-Net pushes the computational efforts to those decoder branches processed exclusively when high quality is needed. Moreover, the improved training procedure allows us to remove the H-decoder, which is the most consuming block with 62.8% of energy.

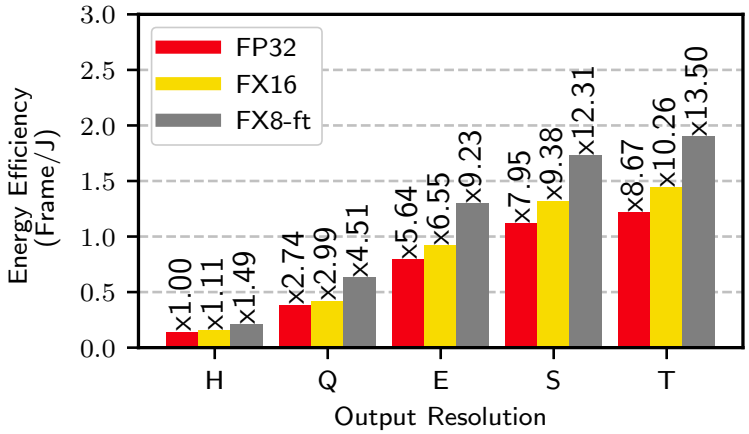


Fig. 4.5 Energy efficiency at different output scales and precision configurations. Annotations indicate the relative improvement with respect to H@FP32 (0.141 Frame/J).

The cooperation with precision scaling further extends the achievable savings, as depicted in Fig. 4.5. The bar-plot illustrates the energy gains (normalized to H@FP32) achieved through fixed-point quantization at each scale. The observed trends validate the proposed Q.Neural Kernels (see Section 3.3.3), since the energy efficiency lowers with the precision. Most importantly, FX16 consistently outperforms FP32 while keeping the same accuracy. A more in-depth analysis reveals interesting aspects of the nature of the two knobs. The savings achieved by resolution scaling gets lower at smaller scales, ranging from 2.69× (H→Q at FX16) to 1.09× (S→T at FX16), whereas the savings brought by reduced bit-width keeps almost constant, around 1.38× (FX16→FX8-ft at constant resolution). Indeed, precision scaling operates as a coarse-grain knob and improves efficiency by reducing the complexity of all the network layers.

A more intelligible representation of this concept is reported in the Pareto analysis of Fig. 4.6. The plot places each configuration of EQPyD-Net in the energy vs. accuracy space. The Pareto curve connects the five non-dominated points (orange line). This analysis reveals that switching across high-resolution configurations (Q, E, S) brings significant energy savings with a small accuracy loss. Instead, when moving towards the lowest resolution (S→T), the accuracy degradation gets substantial (-2.6%) with limited energy savings (-9.38%). Conversely, precision scaling (S@FX16→S@FX8-ft) achieves 31.2% savings, yet with negligible accuracy drop (only 0.8%).

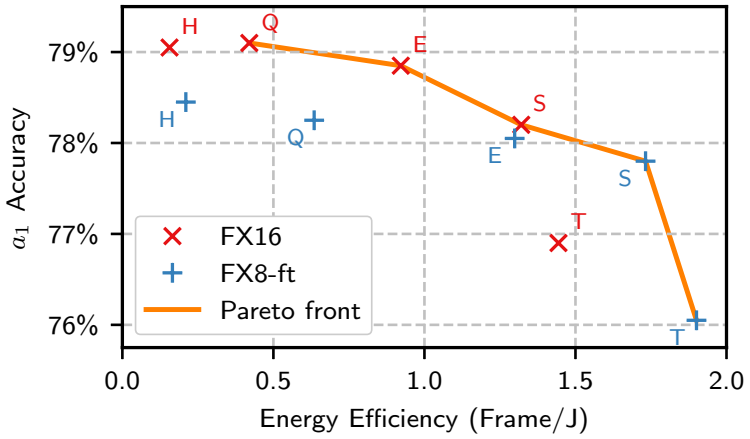
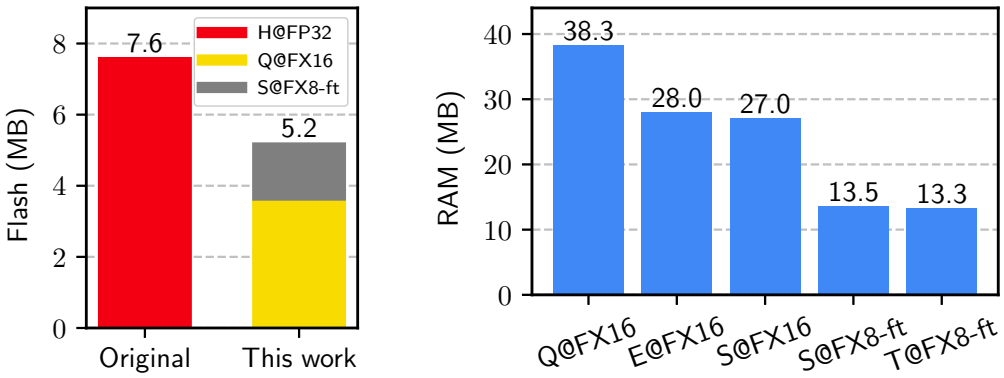


Fig. 4.6 Energy and accuracy at different output resolutions (from left to right H→T) and precision configurations (FX16 and FX8-ft). The orange curve connects Pareto-optimal solutions.

Memory

This subsubsection aims to evaluate the cost of storing the Pareto optimal configurations of EQPyD-Net (those of Fig. 4.6), enabling energy-quality scaling at run time. The overall memory footprint is the sum of two main contributions: the network weights, stored on the flash memory and block-loaded into RAM at initialization time, and the partial activations produced by the intermediate layers during inference. The latter are stored on the RAM through a time-shared buffer to reduce peak memory usage. At a given arithmetic precision, different resolution options share the same network weights since resolution scaling implies layer skipping (see Section 3.3.2). By contrast, configurations centered on different arithmetic precisions need the availability of two separate weight sets, i.e., 16-bit and 8-bit. Nevertheless, as depicted in the barplot of Fig. 4.7a, the weights of the original PyD-Net (H@FP32) occupies 7.6 MB (red bar), whereas the sum of the two configurations Q@FX16 and S@FX8-ft takes only 5.2 MB (yellow and gray bars), 44% less memory.

Concerning the RAM requirements, the decoder branches determine the peak usage due to their large number of channels (up to 96). Therefore, the RAM depends on the selected scale, as can be inferred from Fig. 4.7b. Besides that, precision scaling allows further savings with a linear reduction of RAM resources. Indeed, the transition S@FX16→S@FX8-ft halves the memory utilization. In all cases, the resources needed are limited to few tens of MBs, from 38.3 MB



(a) (b)

Fig. 4.7 Flash (a) and RAM (b) requirements for Energy-Quality scaling with EQPyD-Net.

to 13.3 MB, instead of 206.0 MB required by the original PyD-Net (H@FP32), making EQPyD-Net extremely suitable for several embedded platforms.

4.3.3 Discussion

This section presented a thorough assessment of EQPyD-Net, an energy-quality scalable system capable of performing monocular depth estimation on low-power devices. The energy-quality scalability of EQPyD-Net comes from the use of two knobs acting at different optimization levels: the resolution scaling enabled by the reconfigurable topology of the network and the multi-scale training; the precision scaling enabled by network quantization and low-level code optimizations. Experimental results collected on the Cortex-A53 CPU validate the efficiency of the adopted strategy. Future works could investigate the feasibility of a similar energy-quality scalable solution also for binocular [94] or multi-view stereo setup.

4.4 Energy-Quality Scalable CNNs on Tiny Devices via Nested Sparsity

As extensively highlighted in the previous sections, building a dynamic CNN encompasses the choice of proper knobs that enable the latency-quality trade-off

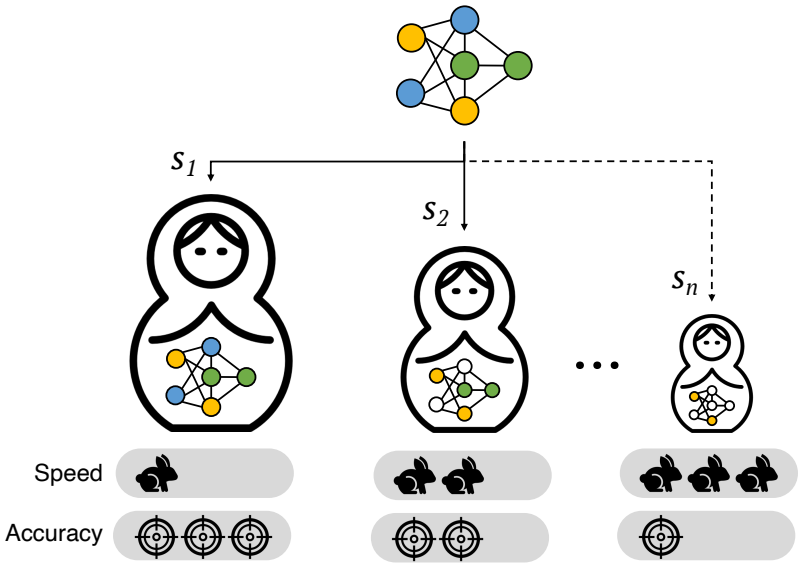


Fig. 4.8 A pictorial representation of a *Nested Sparse CNN*

at run time. In the case of MCUs, such a choice is very challenging as it must be compatible with the minimal storage and memory resources available on the device, namely, a few MBs of FLASH (1-2MB) and hundreds of KBs of RAM ($\leq 512kB$). Architectural-level knobs recently proposed, like the network depth [22] or the layers width [24], only offer a coarse-grain control for latency (hence energy) and accuracy, and they do not alleviate the pressure on the storage memory as the full model configuration, i.e., the one at the maximum width or maximum depth, might still be too large to fit into the FLASH memory. The availability of fine-grain control knobs capable of modulating the latency but also keeping the model footprint minimal is highly desirable here, and model *sparsity* is a good candidate. Sparse training is less prone to accuracy loss, and sparse models can be compressed via sparse encoding formats [28]. However, how to leverage the weight sparsity as the dynamic knob on compact CNNs, e.g., the MobileNets [15], and how to deploy dynamic sparse CNNs efficiently on general-purpose cores are still open issues.

To address such issues and so to enable the use of model sparsity as a dynamic knob on tiny MCUs, we propose a new class of dynamic CNNs called *Nested Sparse CNNs*. A Nested Sparse CNN is a neural model with a single weight-set capable of working in N different configurations of increasing sparsity. It can be viewed as a super-network containing N sparse sub-networks with

nested weight-sets (like the *Matryoshka doll* depicted in Fig. 4.8). A low sparsity value corresponds to high accuracy, whereas a high sparsity value results in a faster inference process at lower accuracy. A Nested Sparse CNN is the result of a novel end-to-end optimization pipeline, covering different levels of the design stack:

- at training time, we introduce a gradient masking technique that properly routes the learning signals between the nested sparse networks guaranteeing convergence and high accuracy;
- at compile time, we propose a new sparse matrix compression format that fruitfully exploits the nested structure of the weight-sets to minimize the storage footprint;
- at run time, we present sparse compute kernels capable of performing tensor operations using the compressed weights without expensive decoding stages and ensuring no additional latency cost when switching among different sparse configurations.

4.4.1 Related Works

In this section, we briefly review the main compressed sparse storage formats adopted for sparse CNNs and the work of Wu et al. [151] as it is the one closer to our proposal and deserves more attention.

Compressed Sparse Storage Format

In the case of a sparse tensor, substantial memory savings can be achieved by storing only the value and the position of the non-zero entries. Many different sparse storage formats exist in literature [152], and the optimal one to use depends on the sparsity level, the structure, and the access pattern needed, e.g., random, streaming, or transposed access. For example, to maximize the compression efficiency, a simple bitmap is preferable for low sparsity regimes, whereas coordinate-offset schemes (COO) are more suitable in high-sparse regimes [28]. Sparse storage formats like Compressed Sparse Row (CSR) or Columns (CSC) [33] enables fast row access and hence they can be used to efficiently perform sparse-matrix-vector and sparse-matrix-matrix operations.

Dynamic Sparsity

Following the theoretic principle *the higher the sparsity, the lower the latency*, the authors of [151] introduced a learning method for deep neural models capable of working at different sparsity levels but using a unique weight-set. The training loop was tested only on Recurrent Neural Networks (RNNs) for Automatic Speech Recognition (ASR), known to be more redundant and hence more reliable to pruning [153]. As an anticipation of the experimental sections, we observed substantial accuracy loss when the technique is applied to lightweight CNNs for image classification tasks. Most importantly, the resulting storage footprint and the deployment on a real processing core are open issues that received no attention in prior works.

4.4.2 Building Nested Sparse CNNs

Training

Training a Nested Sparse CNN means learning N sub-networks with increasing sparsity within a single weight-set θ concurrently. Collecting and composing the learning contributions coming from (and directed to) the many sparse sub-networks is not a trivial task, mainly because the learning of the shared weights must be appropriately balanced.

Let's begin with the standard sparse training procedure of static CNNs. While pioneering works, e.g., [29], prevented the pruned weights from getting gradient updates, more recent works [27] have proposed pruning-while-training strategies that *regrow* lost connections and achieve better results. This procedure is the state-of-the-art approach for sparse training. Managing the *regrowth* mechanism for a Nested Sparse CNN is not straightforward, as the "state" of a single weight (i.e., pruned or not-pruned) might differ among the N sub-networks. To handle the many constraining states that may bubble up, we propose *gradient masking*, a novel method conceived to route the learning signals among the sub-networks.

A representation of the dynamics governing the training is reported in Fig. 4.9. At each training step, the sub-networks are sequentially evaluated with increasing order of sparsity value, from low to high, while the weight-set

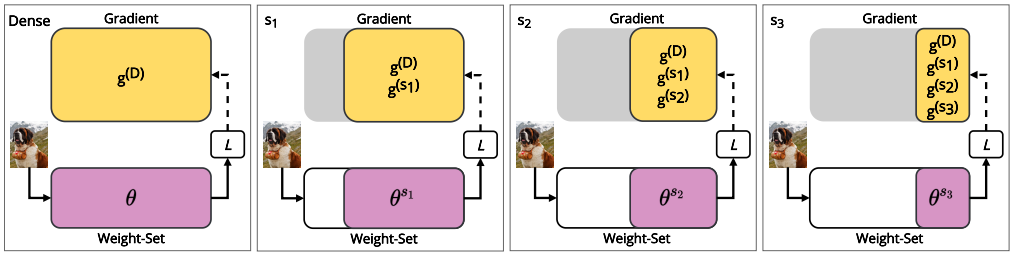


Fig. 4.9 Training step: full weight-set (θ) and the sub-nets (θ^{s_i}) sorted with an increasing order of sparsity (i.e $s_1 < s_2 < s_3$).

θ is updated only at the end. Each frame in the picture shows a consecutive evaluation of a different sub-network, consisting of forward (solid line) and backward (dashed line) passes. L indicates the training loss. The first frame on the left (labeled as Dense) is for the full weight-set θ (i.e., sparsity s_0=0%), while the following ones refer to sub-networks with increasing order of sparsity (i.e., $s_{i+1}>s_i$), from left to right. Note that the dense network is utilized only during the training phase for stability but discarded at inference time. With *gradient masking*, the weights pruned for a specific sparsity value s_i no longer contribute to the following stages, neither to the forward nor to the backward propagation; this is graphically depicted in Fig. 4.9 by shadowed gray regions. For instance, the gradient computation from the sub-network with sparsity level s_2 (i.e., $g^{(s_2)}$) does not interfere with the gradients previously computed for the less sparse sub-networks. This flow allows the entire weight-set θ to evolve during the training process while guaranteeing that each sparse sub-network receives the correct gradient, i.e., only the one related to its active weights.

Formally, to extract the N nested weight-sets $\theta^{(s_i)}$ for $i \in \{1, ..., N\}$ from a single weight-set θ we use a set of binary masks $M = \{M^{(s_1)}, ..., M^{(s_N)}\}$ such that $\theta^{(s_i)} = \theta \circ M^{(s_i)}$[1]. Each mask M^{s_i} is obtained by ranking the weight-set θ by magnitude, then selecting the weights to prune according to the desired sparsity level s_i. Such a strategy enforces the nesting of all weight-sets θ_i, which can be formalized as:

$$s_1 < s_2 < ... < s_N \;\Rightarrow\; \theta^{(s_1)} \supset \theta^{(s_2)} \supset ... \supset \theta^{(s_N)}. \tag{4.1}$$

[1] $\circ$ indicates the Hadamard product between two matrices.

Algorithm 4: Nested Sparse Training

```
1  Function main(steps, S, block_shape, optimizer):
2      for t in steps do
3          optimizer.zero_grad() // Ĝ = 0
4          soft_labels = forward (θ)
5          Ĝ += backward (θ)
6          if pruneStep (t) then
7              for s in S do
8                  M^s = getMask (θ, s, block_shape)
9                  θ^s = θ ∘ M^s
10                 forward(θ^s, soft_labels)
11                 ĝ^s = backward (θ^s)
12                 Ĝ += M^s ∘ ĝ^s // Gradient Masking
13             end
14         optimizer.step() // Ĝ update
15     end
16     M = { getMask (θ, s, block_shape) for s in S }
17     return θ, M
18 Function getMask(θ, s, block_shape):
19     blocks = groupBlocks (θ, block_shape)
20     idx = rankBlocks (blocks, s)
21     mask = ones_like (θ)
22     mask[idx] = 0
23 return mask
```

The pseudocode of the proposed training loop is reported in Algorithm 4. It follows the skeleton of the state-of-the-art iterative pruning procedure [154] where a model is gradually pruned until a target sparsity level is reached. The procedure is fed with the set of sparsity levels $S = \{s_1, ..., s_N\}$ and the *block_shape* ($m \times n$); it returns the weight-set θ and the set of masks M, one for each sparsity level $s \in S$.

The training loop alternates dense and sparse training epochs, according to a fixed scheduler (line 6). At the beginning of each epoch, the gradient is zeroed (line 3), then the forward and backward passes are performed with the entire weight-set θ (lines 3-5). The weights are directly updated using the gradient value (line 14) during the dense steps. During the sparse steps, for each sparsity level s (line 7), the `getMask` function generates a mask M^s (line 8). This mask

is point-wise multiplied with θ to extract the sparse sub-network θ^s (line 9), which is used to complete the forward and backward passes (lines 10-11). Notice that the predictions of the dense model (line 4) are used as soft labels for the sparse sub-networks (line 10), as a form of in-place distillation [155]. As the last step, the local gradient $\hat{g}^s$, relative to the sub-network θ^s, is masked and merged with the previous gradient contributions (line 12). The effect of the *gradient masking* is twofold: first, it allows less sparse (and possibly more accurate) sub-networks to influence the weights of the more sparse and weaker ones; second, it shields the more sparse (and hence less accurate) sub-networks, preventing abrupt changes in the learning curve. Once the contributions of each sub-network are accumulated in the global gradient $\hat{G}$, the weight-set θ is updated (line 14). At the end of the training phase, the weight-set and the set of nested masks M are returned (lines 16-17), ready to be used during the inference stage.

The `getMask` function used to extract a binary mask given the sparsity value s_i works as follows. First, weights are grouped into blocks of shape $m \times n$ (line 19), where m is in the output-channels axis. Second, blocks are ranked according to their magnitude (L^2-norm) through the `rankBlocks` function, which returns the position (idx) of the sorted weights in descending order. Finally, the less important $s_i \cdot |\theta|$ weights are pruned (line 20) by setting to zero their value in the binary mask M_{s_i} (lines 21-22).

Compression

Fig. 4.10 illustrates an example of the proposed sparse matrix compression format, named *NestedCSR*, for a nested model trained with three generic sparsity levels $s_1 < s_2 < s_3$ and using a 1×2 block shape. It is worth emphasizing that the compression format is general and can be used with any number of sparsity levels or block sizes. At the lower sparsity level (s_1), the matrix comprises the red, green, and blue non-zero blocks; at the medium sparsity level (s_2), the red and green blocks; at high sparsity level (s_3), the red blocks. As shown in the figure, the three configurations can be seen as a composition of three disjoint sparse matrices, and this is precisely the property exploited by *NestedCSR*. Each sparse sub-set is compressed using a block CSR format [33]: the *nz-values* array stores the values of the non-zero blocks in

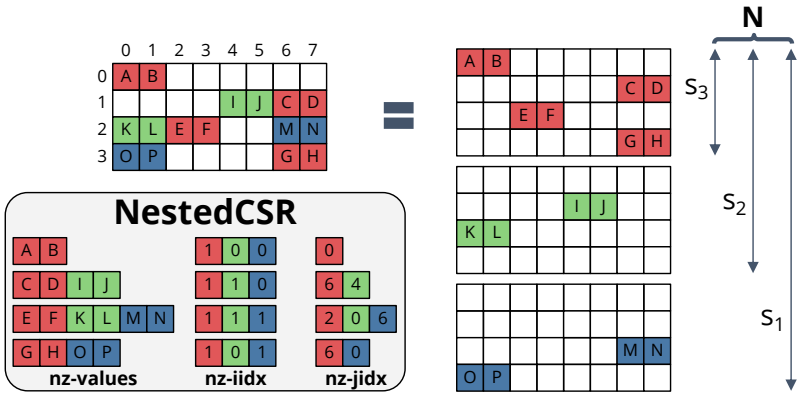


Fig. 4.10 Example of the proposed NestedCSR format applied to a 1×2 block sparse matrix W that can work in three sparsity levels $\{s_1, s_2, s_3\}$.

row-major order, the *nz-iidx* the number of non-zero blocks on each row, and the *nz-jidx* the column position of each non-zero blocks. The three arrays of each sparse sub-set are concatenated row-wise, from the most sparse to the least sparse (from red to blue in Fig. 4.10).

The footprint of a block-sparse matrix W of dimensions $R \times C$ encoded with *NestedCSR* depends on the block shape $m \times n$ and on the number of sparsity levels (N). The size of the arrays is reported in the following equations:

$$\begin{aligned} |\textit{nz-values}| &= (1-s_{min}) \cdot R \cdot C \\ |\textit{nz-iidx}| &= N \cdot R \\ |\textit{nz-jidx}| &= (1-s_{min}) \cdot \frac{R \cdot C}{n \cdot m} \end{aligned} \tag{4.2}$$

The main advantage of such a format is that the amount of storage memory is weakly affected by the number of nested configurations. In fact, the number of sparse configurations (N) only affects the size of *nz-iidx*, which, however, is usually negligible compared to that of the other two arrays. Thus, the overall footprint is mainly set by the smallest adopted sparsity value (s_{min}).

To accelerate the processing of a nested and compressed sparse layer on a general-purpose core, we implemented a custom compute kernel that performs a matrix multiplication $C = A \cdot B$ between a sparse matrix (A) encoded using the *NestedCSR* format and a dense matrix (B), as shown in Fig. 4.11a. The kernel

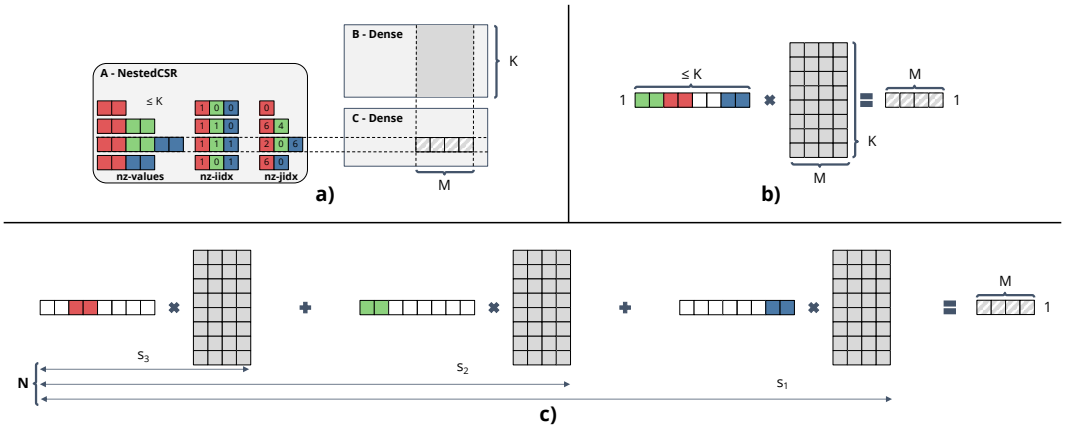


Fig. 4.11 Example of the proposed compute kernel performing a sparse matrix-matrix multiplication between a 1×2 block sparse matrix A encoded using the NestedCSR format and a dense matrix B.

can be used to process both fully-connected and convolutional layers, where, for the latter, a convolution-as-GEMM implementation is adopted [33, 39].

Similar to CSR-based sparse matrix multiplication, the whole operation is carried out as a sequence of small matrix operations between M columns of the dense matrix and 1 row of the sparse matrix as shown in Fig. 4.11b. Such implementation choice amortizes the cost of the indirection process needed to access one element of the sparse matrix across multiple multiply-and-accumulate operations. Specifically, we experimentally found out that M=4 represents a good trade-off between data-reuse and register pressure on small MCUs. Since in the NestedCSR format, a single row of the sparse matrix is encoded as N sparse components, also the multiplication operation is decomposed into at most N sparse operations as shown in Fig. 4.11c. Then, based on the sparsity value s_i selected at run time, only some operations are performed while the others are skipped, exploiting the higher sparsity level to effectively reduce the overall computational complexity. Notice that there is no additional cost from switching the sparsity level, as the kernel implementation can be specialized at compile time and then simply called at run time based on the input s_i of the procedure.

4.4.3 Experimental Results

Experimental Set-up

The proposed method was evaluated on image classification (IC) and object detection (OD) using the following data-sets.
Cifar-10/100 (IC) [156]: $60k$ 32×32 RGB images annotated with 10/100 labels and split into $45k$ samples for training, $5k$ for validation, and $10k$ for testing.
PASCAL VOC (OD) [157]: 15870 RGB images picked from the 2007 and 2012 PASCAL Visual Object Classes Challenge, counting of 37813 objects annotated with 20 different labels. As suggested in [158], VOC07 and VOC12 *trainval* data were used for training, while VOC07 was dedicated to testing. We limited the number of different classes to the top-10 classes recognized by the full-scale model. The image resolution was re-scaled to 160×160 with a bi-linear interpolation; this is mandatory due to the strict memory constraints of the target MCU ($512KB$ of RAM, $2MB$ of FLASH).

As CNN benchmarks we opted for lightweight architectures suitable for the IoT segment: a 9-layer ResNet (ResNet9) [1] for IC on Cifar-100; MobileNetV1 [15] for IC on Cifar-10; MobileNetV2 [84] as backbone for a Single Shot Detector (SSD) [158].

The training procedure for the IC task was driven by the SGD optimizer (momentum 0.9, weight decay 0.0005) for 300 epochs with batch size 128. The learning rate followed a cosine annealing schedule starting from 0.05. The same procedure was adopted for the SSD training, except for the batch-size set to 32. Images were flipped and rotated for data augmentation on the IC task, whereas, for OD, we followed the same strategy presented in [158]. Each training experiment was run three times with different seeds, and the average accuracy is reported. In order to train the sparse networks, both single and nested, the block shape is set to 1×2, and a uniform sparsity value is used for all layers, except for the first layer, which is kept dense (following [34]). After 8 warm-up epochs, the sparsity levels start to increase with a polynomial decay schedule [154]. The training algorithm was implemented within the PyTorch framework ($v1.5.1$) and accelerated with a single consumer graphic card by NVIDIA (Titan Xp).

Table 4.4 Accuracy results. Best results for each sparsity level are highlighted in bold.

a) MobileNetV1 on CIFAR-10.

Training	Sparsity [%]	Accuracy Top-1 [%]			
		w=1.00	w=0.75	w=0.50	w=0.25
Dense	0	90.08	89.35	88.32	85.31
Single Sparse	70	89.70	**88.56**	87.27	**83.32**
	80	89.02	88.13	**87.04**	73.22
	90	**88.81**	86.02	75.20	57.88
DSNN [151]	70	86.30	86.21	84.09	78.84
	80	86.42	85.96	83.69	76.10
	90	85.49	84.62	81.78	72.22
Ours	70	**89.90**	88.48	**87.55**	83.29
	80	**89.20**	**88.24**	86.95	**82.12**
	90	88.50	**87.03**	**85.86**	**78.20**

b) ResNet9 on CIFAR-100.

Training	Sparsity [%]	Accuracy Top-1 [%]			
		w=1.00	w=0.75	w=0.50	w=0.25
Dense	0	73.78	72.24	69.66	63.05
Single Sparse	70	72.93	71.09	68.29	**58.90**
	80	72.61	70.90	67.72	**57.40**
	90	**72.15**	**69.98**	65.04	52.15
DSNN [151]	70	72.9	70.48	63.38	45.25
	80	72.83	69.70	62.48	44.69
	90	71.62	67.56	60.15	40.92
Ours	70	**73.56**	**72.04**	**68.82**	58.70
	80	**72.94**	**71.05**	**68.38**	57.30
	90	71.19	69.59	**65.92**	**52.93**

The set of sparsity levels S used to collect the results cover three values: {70%, 80%, 90%}. Finding the optimal set S to achieve the best accuracy, latency, and storage trade-off is out of the scope of this work.

In the remaining sections we refer to *Dense* as the dense baseline network, *Single Sparse* as the model optimized for a single sparsity level [29], *Nested Sparse CNNs* for our proposal, *Slimmable* as the dynamic model obtained by layers width scaling [24], and *DSNN* as the dynamic sparse model [151]. For *Slimmable*, we adopted the official implementation[2], whereas for *DSNN* we used an in-house implementation as no open-source code was available at the time of writing.

The inference latency was measured on a NUCLEO-F767ZI powered by an ARM Cortex-M7 MCU operating at $216MHz$. The board hosts 512KB of on-chip SRAM and $2MB$ of FLASH. The CMSIS-NN library v.5.6.0 [39] was extended by the sparse matrix multiplication kernels described in the previous section, with a block-shaped set to 1×2 to exploit the Single Instruction Multiple Data media accelerator of the M7 core [33]. To comply with the arithmetic requirements of the CMSIS-NN library, the CNNs under analysis were quantized to 8-bit using a layer-wise symmetric binary scaling [40]. We adopted the GNU Arm Embedded Toolchain (version 6.3.1) for cross-compilation.

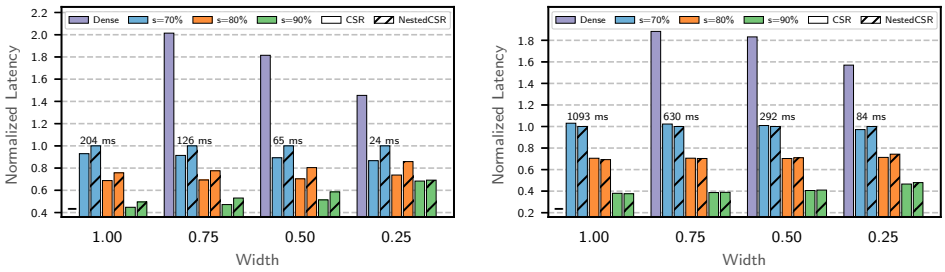
[2]https://github.com/JiahuiYu/slimmable_networks

Training Evaluation

To assess the quality and generalization properties of the proposed nested training, we analyzed the accuracy achieved over the IC tasks by CNN architectures of decreasing capacity, that is, reduced by a width scaling factor $w \in \{1.00, 0.75, 0.50, 0.25\}$. Such scaling should not be confused with the dynamic width scaling of [24], which is discussed later (Section 4.4.3).

The main results are collected in Tab. 4.4a and Tab. 4.4b. Training a network for a single sparsity level should be considered as the best case because the parameters are optimized for that specific sparsity level. Contrarily, the training of a Nested Sparse CNN has to concurrently optimize multiple sub-networks while trying to drive each of them towards the highest accuracy. Despite the multi-objective nature of the optimization, Nested Sparse CNNs actually perform better than individually trained sparse models in several cases. The gap is rather small when they achieve inferior accuracy: the (worst case) accuracy drop is 0.31% for MobileNetV1 and 0.96% for ResNet9. These results prove that the gradient masking technique enables less sparse sub-networks to boost the accuracy of more sparse ones. For instance, the single sparse MobileNetV1@w0.25 with s=90% shows a drastic quality drop, whereas the Nested Sparse model is 20.32% more accurate, closing the gap with the less sparse configurations. The joint training benefits configurations with higher sparsity values, but it also improves the least sparse ones due to proper usage of the dense model in the training loop. Nested Sparse ResNet9@w0.75 at the lowest sparsity ratio (s=70%) is $\approx$ 1% more accurate than the single sparse model, and hence much closer to the dense model.

Tab. 4.4 collects the accuracy obtained with DSNNs [151]. While the training of DSNNs has proven effective on RNNs for ASR, our results reveal they get worse on tiny CNNs for IC tasks: 3.40% less accurate than the single sparse configuration on MobileNetV1 and 13.65% on ResNet9. Except for ResNet9@w1.00 with s=90%, Nested Sparse CNNs outperform DSNNs, improving for compact networks with a lower width and a higher sparsity level.



a) MobilenetV1 - Cifar10 b) ResNet9 - Cifar100

Fig. 4.12 Latency values normalized for each width to the NestedCSR@s=70%. The latency of the dense model at w=1.00 is not shown as it exceeds the FLASH memory of the adopted device (2MB).

Compression Pipeline Evaluation

Tab. 4.5 reports the storage profiles for ResNet9 and MobileNetV1, showing that Nested Sparse CNNs achieve substantial savings: three nested sparse configurations require as low as $1016kB$ (54% smaller than the dense baseline) in the case of ResNet9@w1.00, while $1464kB$ (53% smaller) in the case of MobileNetV1@w1.00. Interestingly, a Nested Sparse CNN takes almost the same storage of its least sparse configuration. For instance, encoding a 70% sparse model with block CSR [33], calls for $1014KB$ in the case of ResNet9@w1.00 (only $2kB$ less than NestedCSR) and $1458kB$ for MobileNetV1@w1.00 (only $6kB$ less than NestedCSR). The models centered on the other widths follow similar trends, confirming the effectiveness of the NestedCSR format across a wide set of topology configurations.

With the aid of custom-designed compute kernels, not just the memory but also the latency take advantage of the NestedCSR format. Fig. 4.12 reports a comparative analysis for ResNet9 and MobileNetV1, both dense and sparse, using a classical CSR [33] and the proposed NestedCSR. As expected, the sparse kernels introduce a remarkable speed-up w.r.t. the dense versions, but even more remarkable, they allow Nested Sparse CNNs to reach comparable performance to single sparse CNNs. For ResNet9, the multi-sparse kernels perform slightly better than single sparse kernels (1.83% on average) at high width (w=1.00 and w=0.75), and show more overhead at low width (4.04% in the worst case). For MobileNetV1, the multi-sparse kernels perform moderately worse (10.91% slower on average), and the overhead increases more notably

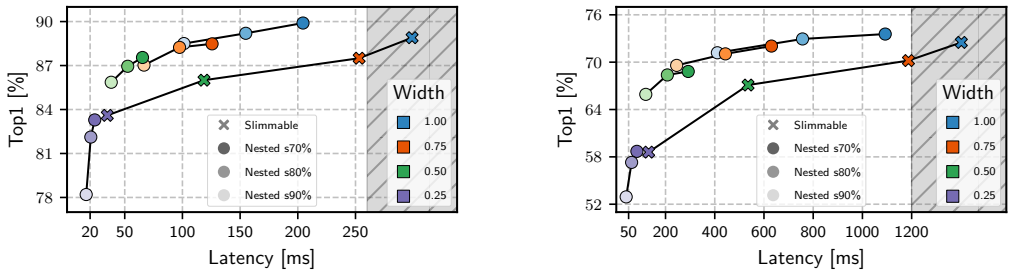
Table 4.5 Storage footprint of ResNet9 trained on Cifar100 and MobileNetV1 trained on CIFAR10.

Model	**Method**	**Sparsity** [%]	**Storage [KB]** w=1.00	w=0.75	w=0.50	w=0.25
MobileNetV1	Dense	0	3132	1774	800	208
	Single	70	1458	834	384	106
	Nested	$\{70, 80, 90\}$	1464	839	387	108
ResNet9	Dense	0	2232	1259	562	143
	Single	70	1014	575	260	68
	Nested	$\{70, 80, 90\}$	1016	576	260	68

(up to 14.08% in the worst case) for more sparse and smaller networks. The architectural differences between ResNet9 and MobileNetV1 originate from the differences in the performances of the sparse and multi-sparse kernels. In MobileNetV1, there are many convolutional layers, but only the point-wise 1×1 are sparsified. Whereas, in ResNet9, there are fewer convolutional layers, but they are all sparse, and they have a higher number of channels with larger kernels (3×3). Although the use of multi-sparse kernels incurs such a latency penalty, it still preserves the latency gain brought by sparsity while occupying much lower storage. In fact, multiple sparse networks would require storing all weight-sets on-device, an unfeasible requirement for tiny end-nodes.

Latency-Quality Scaling

Fig. 4.13 depicts the latency vs. accuracy trade-off achievable by Nested Sparse CNNs. The best dynamic behavior is observed at the highest width. Looking at MobileNetV1@w1.00, an increase of sparsity from 70% to 90% has a minimal effect on accuracy (1.4%), but the speed-up is remarkable: 51% of latency reduction. ResNet9@w1.00 follows the same trend, where a higher sparsity level improves latency by 62% with a moderate effect on accuracy (2.37% loss). Reducing the model width makes the trade-off slightly worse as smaller CNNs are less resilient to pruning. As a result, the accuracy gap increases when the model gets smaller, and the latency speed-up lowers. Still, for the smallest nets ($width$=0.25) an accuracy drop of 5.09% (5.77%) for ResNet9 (MobileNetV1) corresponds to a latency savings of 52% (31%).



a) MobileNetV1- Cifar10 b) ResNet9 - Cifar100

Fig. 4.13 Latency-accuracy scaling for Slimmable CNNs and Nested Sparse CNNs. Grey area shows the unfeasible solution space for the adopted MCU, i.e., FLASH footprint $> 2MB$.

Fig. 4.13 shows the dynamic behavior of CNNs optimized with the *Slimmable* approach [24]. Slimmable networks at maximum width w=1.00 do not satisfy the FLASH constraints ($\leq 2MB$). Only three smaller but weaker configurations can be deployed on-device indeed. Thanks to sparse encoding instead, Nested Sparse CNNs at maximum width meet the memory constraints. Except for the smallest width (w=0.25), Nested Sparse CNNs at s=70% and s=80% turn out to be more accurate and faster than the slimmable models. A Nested Sparse CNN presents a moderate scaling capacity than a slimmable model, which is intuitive as the sparsity acts as a fine-grain control knob. Nevertheless, the low storage footprint paves the way to an attractive hybrid solution, where the width multiplier serves as the static knob complementary to dynamic sparsity. The *Pareto analysis* of Fig. 4.13 reveals that the three Nested Sparse CNNs, i.e., width $\{0.75, 0.50, 0.25\}$, made scaling over the three sparsity levels, outperform the slimmable counterparts, originating eight Pareto optimal implementations that, if stored together, consume less storage than a slimmable model. Precisely, $904kB$ for ResNet9 and $1334kB$ for MobileNetV1, that is, 28% and 25% less than the deployable configurations of the *slimmable* models ($width \leq 0.75$).

As final remark, it worth noticing that the newer scalable training methods, e.g., *EfficientNet* [159] and *OFA* [160], play smartly at *design time* not only with the width but with all network dimensions, namely, width, depth, kernel sizes, and input resolution, to translate a higher computational/memory budget into higher accuracy. Such scaling methods are of utter importance to the design of efficient CNNs, but their purpose is different from that of our work. We demonstrated that tweaking at *run time* the accuracy-latency trade-off via

Table 4.6 SSD-MobileNetV2. Best results for each sparsity level are highlighted in bold.

Training	Sparsity	w=0.50			w=0.35		
		mAP	**Storage**	**Latency**	**mAP**	**Storage**	**Latency**
	[%]	[%]	[**kB**]	[**ms**]	[%]	[**kB**]	[**ms**]
Dense	0	68.32	869	1549	63.42	523	998
Single Sparse	70	66.01	508	1080	60.58	329	752
	80	62.72	407	972	55.20	274	689
	90	29.40	306	862	23.06	219	625
Ours	70	**68.30**	514	1225	**63.12**	334	883
	80	**66.37**		1103	**61.03**		807
	90	**60.33**		951	**55.84**		712

sparsity is feasible even with a reduced storage footprint: only one compressed weight-set must be stored on-device for a Nested Sparse CNN. Thus, our solution can be used in conjunction with existing scaling mechanisms implemented at *design time*. The proposed pipeline, in fact, enables optimally designed networks to cover a broader region of the accuracy-latency space while consuming the same amount of storage space.

Object Detection

To prove the generalization capability of our approach, we evaluated a Nested MobileNetV2 on a bounding-box detection task. The results reported in Tab. 4.6 are for w=$\{0.50, 0.35\}$, which are the only configurations feasible given the FLASH constraint of our target MCU. The Nested Sparse object-detector gets more accurate than the sparse models trained as separate instances. More in detail, for the most sparse configurations (i.e., s=90%), it is 31.85% more accurate (average over the two widths), confirming the stability of the proposed training. As for the image classification task, a hybrid solution is highly effective also here: combining width scaling with nested sparsity enables scalability across a wide latency-accuracy range (ΔTop-1/ΔL = 12.46(%)/368(ms)) while cumulatively occupying 848kB, which is still less than the single dense model at w=0.50.

4.4.4 Discussion

The training and compression pipeline proposed in this section enables the use of model sparsity as a dynamic knob on tiny off-the-shelf devices. The training procedure ensures high accuracy, while a custom compressed storage format and custom compute kernels enable the deployment on tiny off-the-shelf devices. Although the experimental assessment revealed that Nested Sparse CNNs outperform other dynamic strategies while occupying a smaller storage footprint, some issues have not been addressed in the current version of the work. First, the sparsity levels have to be fixed manually before the training procedure. However, as the trade-off accuracy vs. latency depends on the model architecture and the task, designing the optimal set of sparsity values is not trivial and should be automated. Second, although using the same sparsity ratio for all layers of the network was proven effective in previous works [34], exploiting the effects that different layers have on both accuracy [161, 162] and latency [163] may lead to new Pareto solutions. Thus, future works could investigate the integration of an automatic search engine (e.g., [164, 165]) in the proposed pipeline such that multiple sparse configurations are sampled and tested at training time to optimize storage, latency, and accuracy simultaneously.

4.5 Conclusions

This chapter focused on increasing the *energy efficiency* of CNNs. Specifically, the works presented in this chapter extended the achievable accuracy-energy trade-off proposing dynamic knobs capable of tuning the working point of a CNN at run time. The first part of the chapter described and assessed the use of two dynamic knobs, namely, scalable neural topology and multiple arithmetic precision, to build an energy-quality scalable system for monocular depth estimation named EQPyD-Net. This work demonstrated that arithmetic precision serves as a fine-grain knob to extend the energy-quality scalability offered by a dynamic neural topology. Tested on an off-the-shelf ARM Cortex A53, EQPyD-Net can work in a wide range of energy-quality space, comprising five operating points. The second part of the chapter introduced a new training and compression pipeline aimed at using sparsity as a dynamic knob for deploying energy-quality scalable CNNs on MCU devices. Sparsity is an

excellent candidate as it offers fine-grain control over the energy-quality trade-off and reduces the model footprint as sparse models can be compressed via sparse encoding formats. When tested on image classification and object detection tasks on an off-the-shelf ARM-M7 Micro Controller Unit (MCU), the dynamic sparse CNNs obtained with the proposed pipeline outperform other state-of-the-art dynamic strategies.

Chapter 5

Conclusions and Future Works

This dissertation investigated the challenges of deploying modern CNNs on embedded systems. We strongly believe that the on-device processing of CNNs can pave the way for new disruptive applications with low latency and high privacy standards. For this reason, the research questions addressed in this dissertation were inspired by the practical constraints that are currently limiting the deployment of CNNs on an embedded system: the lack of memory resources, the limited amount of computing power, and the need for extreme energy efficiency. Overcoming such strict limitations relies on the availability of *small*, *fast*, and *energy-efficient* CNNs, and achieving it with vertical and automated optimizations is precisely the main contribution of this work.

Chapter 2 presented our effort towards building *small* CNNs. This chapter first reviewed the main algorithms adopted to tackle the problem of memory allocation in DL compilers. Specifically, the extensive quantitative assessment demonstrated the need to revisit the dominant trends in memory allocations, showing that a MILP-based method can achieve better results (up to 33% of memory savings) than commonly adopted heuristics in a reasonable run time (28.31 s in the worst case). The second part of the chapter introduced a novel functionality-preserving graph restructuring procedure aimed at reducing the activation memory footprint of CNNs. The proposed technique exploits the spatial locality of the tensor operators adopted in virtually all CNN architectures, achieving remarkable activation memory savings (62.9% on average) at the cost of small computational overhead (8.6% on average). Finally, the chapter

presented a new compression pipeline, which combines weight pruning with graph restructuring to deploy high-resolution and wide CNNs, like MobileNetV1 and MobileNetV2, on MCU devices with 2MB of FLASH and 512KB of RAM.

Chapter 3 focused on how to build *fast* CNNs on *low-power* devices. This chapter introduced a comprehensive design and optimization framework to accelerate the inference of quantized, lightweight CNNs on ARM Cortex-A CPUs and ARM Cortex-M MCUs. Specifically, the first part of the chapter discussed the acceleration of PyD-Net, a CNN for monocular depth estimation, on a multi-core ARM Cortex-A53 mobile CPU. The experimental evaluation showed marginal accuracy loss on the KITTI dataset with 16-bit (8-bit) integers, latency reduction up to 1.16× (1.64×), and memory footprint reduction up to 2× (4×) compared to single-precision floating-point. The second part of the chapter introduced a CNN named μPyD-Net designed through a mix of optimizations working at the neural architectural level, algorithmic level, and operator level to perform monocular depth estimation on MCUs. The experimental evaluation proved that μPyD-Net is capable of breaking the 1FPS barrier with a power envelope of 0.4 W. Specifically, μPyD-Net on an MCU powered by an ARM Cortex-M7 takes about $600ms$ and $300ms$ to perform depth estimation on 48×48 and 32×32 RGB images while occupying less than $512KB$ of memory.

Chapter 4 focused on increasing the *energy efficiency* of CNNs. Specifically, the works presented in this chapter proposed dynamic knobs capable of tuning the accuracy-energy trade-off of a CNN at run time. The first part of the chapter described and assessed the use of two dynamic knobs, namely, scalable neural topology and multiple arithmetic precisions, to build an energy-quality scalable system for monocular depth estimation. This work demonstrated that arithmetic precision serves as a fine-grain knob to extend the energy-quality scalability offered by a dynamic neural topology. Tested on an off-the-shelf ARM Cortex A53, EQPyD-Net can be shifted across five operating points, ranging from a maximum accuracy of 82.2% on the KITTI dataset with 0.4 Frame/J up to 92.6% of energy savings with 6.1% of accuracy loss. Nevertheless, EQPyD-Net still has a minimal memory footprint of 5.2 MB for the weights and 38.3 MB (in the worst-case) for the run-time processing. The second part of the chapter introduced a new training and compression pipeline using sparsity as a dynamic knob for deploying energy-quality scalable CNNs

on MCU devices. The experimental evaluation carried out on an off-the-shelf ARM-M7 MCU showed that the proposed pipeline produces dynamic sparse CNNs that outperform other state-of-the-art dynamic strategies, like *slimmable* networks. Specifically, the *Pareto analysis* reveals that Nested Sparse CNNs are Pareto dominant in the accuracy-latency space, still occupying less storage than a single slimmable model. Precisely, Nested Sparse ResNet9 occupies $904kB$ while Nested Sparse MobileNetV1 $1334kB$, that is, 28% and 25% less than the deployable configurations of the *slimmable* models.

In summary, this dissertation presented novel optimization techniques, which act vertically across the different layers of the computing stack to reduce the memory footprint, lower the inference processing time, and increase the energy efficiency of state-of-the-art CNNs with minimal to no accuracy loss. Finally, this dissertation further extended the achievable accuracy-complexity trade-off of a CNN by introducing novel dynamic knobs to modulate the energy quality working point at run time. Overall, the works presented in this thesis contribute to the state-of-the-art by pushing further the boundary of accurate CNNs that can be deployed on tiny embedded devices. However, at the moment, the optimizations introduced in this thesis have not been integrated into existing CNN compiler technologies yet. As reported in Section 1.2, all current CNN compilers are structured as multi-level compilers, comprising a graph-level IR and one or multiple operator-level IRs. Thus, the proposed methods do not add further engineering constraints to existing frameworks, which may fundamentally prevent their near-future adoption in a production environment. At the same time, specific design choices of the adopted IRs may make the integration challenging to do in practice. For this reason, as will be discussed in the next section, the modularity of a CNN compiler infrastructure represents a promising direction for future investigations.

5.1 Future Perspectives

This thesis addressed some of the most immediate questions and challenges that arise from the proliferation of DNNs on tiny devices. However, many optimization opportunities are still left on the table, and we believe that they could be explored by building upon the insights revealed by the works

presented in this dissertation. Specifically, we showed that a cross-layer, vertical optimization approach could bring more remarkable gains and allow the quality-of-result to be co-optimized with all non-functional figures of merit of the system. Such specialization of algorithms, compilers, and hardware platforms is crucial to let the performance of DNNs continue to scale, as programmers cannot rely anymore on Moore's Law to get more computational power every few years. In our opinion, bringing forward such a vertical optimization methodology is the most exciting and promising research direction. Thus, in the following paragraphs, we summarise some of the most important steps to address.

Generalization

The optimization techniques presented in this dissertation can be applied to virtually all modern DNNs and to a broad set of general-purpose MCUs and CPUs. However, targeting other DL architectures and different hardware devices may require a considerable engineering effort, as many of the characteristics of the algorithms and the hardware platforms must be manually recognized, represented, and encoded in the optimization process. As domain-specific hardware accelerators for DNNs [166] and novel neural architectures, like Transformers [167] and Sparsely activated multi-task models [168], have started to become more broadly adopted, having a vertical optimization stack ready from *day-zero* is of uttermost importance. To this end, Google has recently introduced the Multi-Level Intermediate Representation (MLIR) framework [169] as a toolkit for building domain-specific compilers. The key feature of MLIR is the ability to define custom abstract intermediate representations, called dialects, which can coexist concurrently during the optimization process. This approach provides a significant level of modularity and composability by letting the optimization framework use the right abstraction level for solving a particular problem. For example, a hardware dialect can capture the semantic of a hardware accelerator that natively supports some tensor operations, providing primitive building blocks. Then, such building blocks can be gradually composed with generic optimization algorithms into larger blocks and exposed to the algorithmic level. At the same time, high-level information can be captured by an algorithmic dialect and kept as meta-data during the whole lowering process to guide more low-level, hardware-specific optimization passes. We believe that new

methodologies to create dialects and retargetable optimizations algorithms that can work on many different dialects represent an exciting step forward.

Cross-device offloading

With the advent of the Internet-of-Things era and the promises of future network infrastructures, users and intelligent environments will become increasingly equipped with more connected embedded devices. In this dissertation, we have investigated how to optimize the deployment of a DNN when targeting a *single device*. However, for example, a smartwatch might be able to offload some more heavy workload to a smartphone, thus taking advantage of a nearby device with more computational power. The smartphone could then use the powerful infrastructure available in the cloud to run some re-training due to the availability of new data or to fine-tune the model for some new tasks. In this space, future research efforts may be directed to explore this ecosystem of interconnected devices as a new axis in the vertical optimization space.

List of Publications

This appendix contains a complete list of the publications co-authored by Antonio Cipolletta during the years of his PhD. This list also includes publications which were not presented in this thesis, but still releated to the optimization of Deep Neural Networks for embedded systems.

Overall, Antonio Cipolletta has co-authored:

- 2 accepted papers in international peer-reviewed journals;
- 6 accepted papers in international peer-reviewed conferences;
- 1 national patent.

International Peer-Reviewed Journals

- V. Peluso, **A. Cipolletta**, A. Calimera, M. Poggi, F. Tosi, F. Aleotti, and S. Mattoccia, "Monocular Depth Perception on Microcontrollers for Edge Applications," in IEEE Transactions on Circuits and Systems for Video Technology.
- **A. Cipolletta**, V. Peluso, A. Calimera, M. Poggi, F. Tosi, F. Aleotti, and S. Mattoccia, "Energy-Quality Scalable Monocular Depth Estimation on Low-Power CPUs," in IEEE Internet of Things Journal, vol. 9, no. 1, pp. 25-36, 1 Jan.1, 2022.

International Peer-Reviewed Conferences

- V. Peluso, **A. Cipolletta**, A. Calimera, M. Poggi, F. Tosi and S. Mattoccia, "Enabling Energy-Efficient Unsupervised Monocular Depth Esti-

mation on ARMv7-Based Platforms," 2019 Design, Automation and Test in Europe Conference and Exhibition (DATE), 2019, pp. 1703-1708.

- V. Peluso, R. G. Rizzo, **A. Cipolletta** and A. Calimera, "Inference on the Edge: Performance Analysis of an Image Classification Task Using Off-The-Shelf CPUs and Open-Source ConvNets," 2019 Sixth International Conference on Social Networks Analysis, Management and Security (SNAMS), 2019, pp. 454-459.

- V. Peluso, **A. Cipolletta**, F. Vaiana and A. Calimera, "Integer ConvNets on Embedded CPUs: Tools and Performance Assessment on the Cortex-A Cores," 2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS), 2019, pp. 598-601.

- V. Peluso, **A. Cipolletta**, A. Calimera, M. Poggi, F. Tosi, F. Aleotti, and S. Mattoccia, "Enabling monocular depth perception at the very edge," 2020 Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CPVR), 2020, pp. 392-393.

- **A. Cipolletta** and A. Calimera, "Dataflow Restructuring for Active Memory Reduction in Deep Neural Networks," 2021 Design, Automation and Test in Europe Conference and Exhibition (DATE), 2021, pp. 114-119.

- **A. Cipolletta** and A. Calimera, "On The Efficiency of Sparse-Tiled Tensor Graph Processing For Low Memory Usage," 2021 58th ACM/IEEE Design Automation Conference (DAC), 2021, pp. 643-648.

Patents

- Stima di profondità a ridotto consumo di potenza, e altri segnali da singola immagine. Brevetto nazionale

References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[2] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[3] Hany Hassan, Anthony Aue, Chang Chen, Vishal Chowdhary, Jonathan Clark, Christian Federmann, Xuedong Huang, Marcin Junczys-Dowmunt, William Lewis, Mu Li, Shujie Liu, Tie-Yan Liu, Renqian Luo, Arul Menezes, Tao Qin, Frank Seide, Xu Tan, Fei Tian, Lijun Wu, Shuangzhi Wu, Yingce Xia, Dongdong Zhang, Zhirui Zhang, and Ming Zhou. Achieving human parity on automatic chinese to english news translation, 2018.

[4] Zhong-Qiu Zhao, Peng Zheng, Shou-Tao Xu, and Xindong Wu. Object detection with deep learning: A review. *IEEE Transactions on Neural Networks and Learning Systems*, 30(11):3212–3232, 2019.

[5] William Chan, Navdeep Jaitly, Quoc Le, and Oriol Vinyals. Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4960–4964, 2016.

[6] Ronald Mutegeki and Dong Seog Han. A cnn-lstm approach to human activity recognition. In *2020 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, pages 362–366, 2020.

[7] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. *Ten Lessons from Three Generations Shaped Google's TPUv4i*, pages 1–14. IEEE Press, 2021.

[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE*

Conference on Computer Vision and Pattern Recognition, pages 248–255, 2009.

[9] Simone Bianco, Remi Cadene, Luigi Celona, and Paolo Napoletano. Benchmark analysis of representative deep neural network architectures. *IEEE Access*, 6:64270–64277, 2018.

[10] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[11] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for Large-Scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[12] Yanjun Ma, Dianhai Yu, Tian Wu, and Haifeng Wang. Paddlepaddle: An open-source deep learning platform from industrial practice. *Frontiers of Data and Domputing*, 1(1):105, 2019.

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association.

[14] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, Jack Montgomery, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, Misha Smelyanskiy, and Man Wang. Glow: Graph lowering compiler techniques for neural networks, 2018.

[15] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[16] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016.

[17] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[18] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[20] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[21] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[22] Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E. Gonzalez. Skipnet: Learning dynamic routing in convolutional networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[23] Xinqi Zhu and Michael Bain. B-cnn: Branch convolutional neural network for hierarchical classification, 2017.

[24] Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas S. Huang. Slimmable neural networks. *CoRR*, abs/1812.08928, 2018.

[25] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[26] Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. NAS-bench-101: Towards reproducible neural architecture search. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 7105–7114. PMLR, 09–15 Jun 2019.

[27] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks, 2019.

[28] Torsten Hoefler, Dan Alistarh, Tan Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research*, 22(241):1–124, Sep. 2021.

[29] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[30] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[31] Shail Dave, Riyadh Baghdadi, Tony Nowatzki, Sasikanth Avancha, Aviral Shrivastava, and Baoxin Li. Hardware acceleration of sparse and irregular tensor computations of ml models: A survey and insights. *Proceedings of the IEEE*, 109(10):1706–1752, 2021.

[32] Nal Kalchbrenner, Erich Elsen, Karen Simonyan, Seb Noury, Norman Casagrande, Edward Lockhart, Florian Stimberg, Aaron van den Oord, Sander Dieleman, and Koray Kavukcuoglu. Efficient neural audio synthesis. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2410–2419. PMLR, 10–15 Jul 2018.

[33] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 548–560, New York, NY, USA, 2017. Association for Computing Machinery.

[34] Erich Elsen, Marat Dukhan, Trevor Gale, and Karen Simonyan. Fast sparse convnets. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[35] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, Oct 2017.

[36] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 26–35, New York, NY, USA, 2016. Association for Computing Machinery.

[37] Hande Alemdar, Vincent Leroy, Adrien Prost-Boucle, and Frédéric Pétrot. Ternary neural networks for resource-efficient ai applications. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2547–2554, 2017.

[38] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – ECCV 2016*, pages 525–542, Cham, 2016. Springer International Publishing.

[39] Liangzhen Lai and Naveen Suda. Enabling deep learning at the lot edge. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6, 2018.

[40] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[41] Sebastian Vogel, Mengyu Liang, Andre Guntoro, Walter Stechele, and Gerd Ascheid. Efficient hardware acceleration of cnns using logarithmic data representation with arbitrary log-base. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE Press, 2018.

[42] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1737–1746, Lille, France, 07–09 Jul 2015. PMLR.

[43] Taesik Na and Saibal Mukhopadhyay. Speeding up convolutional neural network training with dynamic precision scaling and flexible multiplier-accumulator. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, ISLPED '16, pages 58–63, New York, NY, USA, 2016. Association for Computing Machinery.

[44] Lei Shan, Minxuan Zhang, Lin Deng, and Guohui Gong. A dynamic multi-precision fixed-point data quantization strategy for convolutional neural network. In Weixia Xu, Liquan Xiao, Jinwen Li, Chengyi Zhang, and Zhenzhen Zhu, editors, *Computer Engineering and Technology*, pages 102–111, Singapore, 2016. Springer Singapore.

[45] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. Association for Computing Machinery.

[46] Chris Leary and Todd Wang. Xla: Tensorflow, compiled. *TensorFlow Dev Summit*, 2017.

[47] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):708–727, 2021.

[48] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 47–62, New York, NY, USA, 2019. Association for Computing Machinery.

[49] Byung Hoon Ahn, Jinwon Lee, Jamie Menjay Lin, Hsin-Pai Cheng, Jilei Hou, and Hadi Esmaeilzadeh. Ordering chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 44–57, 2020.

[50] Phitchaya Mangpo Phothilimthana, Amit Sabne, Nikhil Sarda, Karthik Srinivasa Murthy, Yanqi Zhou, Christof Angermueller, Mike Burrows, Sudip Roy, Ketan Mandke, Rezsa Farahani, Yu Emma Wang, Berkin Ilbeyi, Blake Hechtman, Bjarke Roune, Shen Wang, Yuanzhong Xu, and Samuel J. Kaufman. A flexible approach to autotuning multi-pass machine learning compilers. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 1–16, 2021.

[51] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions, 2018.

[52] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to

optimize tensor programs. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[53] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning, 2014.

[54] Arm compute library. https://github.com/ARM-software/ComputeLibrary (last visited on 09/11/2019).

[55] Nicolas Weber, Florian Schmidt, Mathias Niepert, and Felipe Huici. Brainslug: Transparent acceleration of deep learning through depth-first parallelism, 2018.

[56] A Geiger, P Lenz, C Stiller, and R Urtasun. Vision meets robotics: The kitti dataset. *The International Journal of Robotics Research*, 32(11):1231–1237, 2013.

[57] Nai-Sheng Syu, Yu-Sheng Chen, and Yung-Yu Chuang. Learning deep convolutional networks for demosaicing, 2018.

[58] Clement Godard, Oisin Mac Aodha, Michael Firman, and Gabriel J. Brostow. Digging into self-supervised monocular depth estimation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[59] Hsin-Pai Cheng, Tunhou Zhang, Yukun Yang, Feng Yan, Harris Teague, Yiran Chen, and Hai Li. Msnet: Structural wired neural architecture search for internet of things. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*, Oct 2019.

[60] Hsin-Pai Cheng, Tunhou Zhang, Yukun Yang, Feng Yan, Shiyu Li, Harris Teague, Hai Li, and Yiran Chen. Swiftnet: Using graph propagation as meta-knowledge to search highly representative neural architectures, 2019.

[61] Aakanksha Chowdhery, Pete Warden, Jonathon Shlens, Andrew Howard, and Rocky Rhodes. Visual wake words dataset, 2019.

[62] Nucleo-f767zi. https://www.st.com/en/evaluation-tools/nucleo-f767zi.html (last visited on 09/20/2019).

[63] Valentino Peluso, Antonio Cipolletta, Francesco Vaiana, and Andrea Calimera. Integer convnets on embedded cpus: Tools and performance assessment on the cortex-a cores. In *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 598–601, 2019.

[64] Matteo Grimaldi, Valentino Peluso, and Andrea Calimera. Optimality assessment of memory-bounded convnets deployed on resource-constrained risc cores. *IEEE Access*, 7:152599–152611, 2019.

[65] Miguel de Prado, Nuria Pazos, and Luca Benini. Learning to infer: Rl-based search for dnn primitive selection on heterogeneous embedded systems. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1409–1414, 2019.

[66] Antonio Cipolletta and Andrea Calimera. Dataflow restructuring for active memory reduction in deep neural networks. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 114–119, 2021.

[67] Antonio Cipolletta and Andrea Calimera. On the efficiency of sparse-tiled tensor graph processing for low memory usage. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 643–648, 2021.

[68] Narasinga Rao Miniskar, Sirish Kumar Pasupuleti, Vasanthakumar Rajagopal, Ashok Vishnoi, Chandra Kumar Ramasamy, and Raj Narayana Gadde. Optimal sdram buffer allocator for efficient reuse of layer io in cnns inference framework. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2018.

[69] Taro Sekiyama, Takashi Imamichi, Haruki Imai, and Rudy Raymond. Profile-guided memory optimization for deep neural networks. *arXiv preprint arXiv:1804.10001*, 2018.

[70] Arun Abraham, Manas Sahni, and Akshay Parashar. Efficient memory pool allocation algorithm for cnn inference. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 345–352, 2019.

[71] Junzhe Zhang, Sai Ho Yeung, Yao Shu, Bingsheng He, and Wei Wang. Efficient memory management for gpu-based deep learning systems. *arXiv preprint arXiv:1903.06631*, 2019.

[72] Yury Pisarchyk and Juhyun Lee. Efficient memory management for deep neural net inference. *arXiv preprint arXiv:2001.03288*, 2020.

[73] tensorflow lite. https://www.tensorflow.org/lite (last visited on 10/20/2020).

[74] pytorch mobile. https://pytorch.org/mobile/home/ (last visited on 10/20/2020).

[75] arm nn sdk. https://www.arm.com/products/silicon-ip-cpu/ethos/arm-nn(last visited on 10/20/2020).

[76] Wei-Fen Lin, Der-Yu Tsai, Luba Tang, Cheng-Tao Hsieh, Cheng-Yi Chou, Ping-Hao Chang, and Luis Hsu. Onnc: A compilation framework connecting onnx to proprietary deep learning accelerators. In *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 214–218, 2019.

[77] An-Chieh Cheng, Jin-Dong Dong, Chi-Hung Hsu, Shu-Huan Chang, Min Sun, Shih-Chieh Chang, Jia-Yu Pan, Yu-Ting Chen, Wei Wei, and Da-Cheng Juan. Searching toward pareto-optimal device-aware neural architectures. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '18, New York, NY, USA, 2018. Association for Computing Machinery.

[78] Yohei Arahori, Takashi Imamichi, and Hiroshi Nagamochi. An exact strip packing algorithm based on canonical forms. *Computers and Operations Research*, 39(12):2991–3011, 2012.

[79] Gerald Gamrath, Daniel Anderson, Ksenia Bestuzheva, Wei-Kun Chen, Leon Eifler, Maxime Gasse, Patrick Gemander, Ambros Gleixner, Leona Gottwald, Katrin Halbig, Gregor Hendel, Christopher Hojny, Thorsten Koch, Pierre Le Bodic, Stephen J. Maher, Frederic Matter, Matthias Miltenberger, Erik Mühmer, Benjamin Müller, Marc Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Christine Tawfik, Stefan Vigerske, Fabian Wegscheider, Dieter Weninger, and Jakob Witzig. The SCIP Optimization Suite 7.0. Technical report, Optimization Online, March 2020.

[80] Ed Klotz and Alexandra M. Newman. Practical guidelines for solving difficult mixed integer linear programs. *Surveys in Operations Research and Management Science*, 18(1):18–32, 2013.

[81] Marco Cococcioni and Lorenzo Fiaschi. The big-m method with the numerical infinite m. *Optimization Letters*, pages 1–14, 2020.

[82] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.

[83] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[84] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[85] Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. Nimble: Efficiently compiling dynamic neural networks for model inference. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 208–222, 2021.

[86] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344, 2019.

[87] Xuechao Wei, Yun Liang, and Jason Cong. Overcoming data transfer bottlenecks in fpga-based dnn accelerators via layer conscious memory management. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[88] Alessio Burrello, Angelo Garofalo, Nazareno Bruschi, Giuseppe Tagliavini, Davide Rossi, and Francesco Conti. Dory: Automatic end-to-end deployment of real-world dnns on low-cost iot mcus. *IEEE Transactions on Computers*, 70(8):1253–1268, 2021.

[89] Koen Goetschalckx and Marian Verhelst. Breaking high-resolution cnn bandwidth barriers with enhanced depth-first execution. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):323–331, 2019.

[90] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing dnn computation with relaxed graph substitutions. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 27–39, 2019.

[91] Qingcheng Xiao, Yun Liang, Liqiang Lu, Shengen Yan, and Yu-Wing Tai. Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on fpgas. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[92] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, Pete Warden, and Rocky Rhodes. Tensorflow lite micro: Embedded machine learning for tinyml systems. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 800–811, 2021.

[93] Asit Mishra and Debbie Marr. Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy. In *International Conference on Learning Representations (ICLR)*, 2018.

[94] Matteo Poggi, Fabio Tosi, Konstantinos Batsos, Philippos Mordohai, and Stefano Mattoccia. On the synergies between machine learning and binocular stereo for depth estimation from images: a survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–1, 2021.

[95] Clement Godard, Oisin Mac Aodha, and Gabriel J. Brostow. Unsupervised monocular depth estimation with left-right consistency. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[96] Valentino Peluso, Antonio Cipolletta, Andrea Calimera, Matteo Poggi, Fabio Tosi, and Stefano Mattoccia. Enabling energy-efficient unsupervised monocular depth estimation on armv7-based platforms. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1703–1708, 2019.

[97] Valentino Peluso, Antonio Cipolletta, Andrea Calimera, Matteo Poggi, Fabio Tosi, Filippo Aleotti, and Stefano Mattoccia. Enabling monocular depth perception at the very edge. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2020.

[98] Antonio Cipolletta, Valentino Peluso, Andrea Calimera, Matteo Poggi, Fabio Tosi, Filippo Aleotti, and Stefano Mattoccia. Energy-quality scalable monocular depth estimation on low-power cpus. *IEEE Internet of Things Journal*, 9(1):25–36, 2022.

[99] Valentino Peluso, Antonio Cipolletta, Andrea Calimera, Matteo Poggi, Fabio Tosi, Filippo Aleotti, and Stefano Mattoccia. Monocular depth perception on microcontrollers for edge applications. *IEEE Transactions on Circuits and Systems for Video Technology*, pages 1–1, 2021.

[100] H. Hirschmuller. Accurate and efficient stereo processing by semi-global matching and mutual information. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, pages 807–814 vol. 2, 2005.

[101] Luigi Di Stefano, Massimiliano Marchionni, and Stefano Mattoccia. A fast area-based stereo matching algorithm. *Image and Vision Computing*, 22(12):983–1005, 2004. Proceedings from the 15th International Conference on Vision Interface.

[102] T. Kanade and M. Okutomi. A stereo matching algorithm with an adaptive window: theory and experiment. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(9):920–932, 1994.

[103] Nikolaus Mayer, Eddy Ilg, Philip Hausser, Philipp Fischer, Daniel Cremers, Alexey Dosovitskiy, and Thomas Brox. A large dataset to train convolutional networks for disparity, optical flow, and scene flow estimation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[104] Feihu Zhang, Victor Prisacariu, Ruigang Yang, and Philip H.S. Torr. Ga-net: Guided aggregation net for end-to-end stereo matching. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[105] David Eigen, Christian Puhrsch, and Rob Fergus. Depth map prediction from a single image using a multi-scale deep network. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

[106] Iro Laina, Christian Rupprecht, Vasileios Belagiannis, Federico Tombari, and Nassir Navab. Deeper depth prediction with fully convolutional residual networks. In *2016 Fourth International Conference on 3D Vision (3DV)*, pages 239–248, 2016.

[107] Ashutosh Saxena, Min Sun, and Andrew Y. Ng. Make3d: Learning 3d scene structure from a single still image. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(5):824–840, 2009.

[108] Sean Ryan Fanello, Cem Keskin, Shahram Izadi, Pushmeet Kohli, David Kim, David Sweeney, Antonio Criminisi, Jamie Shotton, Sing Bing Kang, and Tim Paek. Learning to be a depth camera for close-range human capture and interaction. *ACM Trans. Graph.*, 33(4), jul 2014.

[109] Tinghui Zhou, Matthew Brown, Noah Snavely, and David G. Lowe. Unsupervised learning of depth and ego-motion from video. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[110] R. Mahjourian, M. Wicke, and A. Angelova. Unsupervised learning of depth and ego-motion from monocular video using 3d geometric constraints. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5667–5675, Los Alamitos, CA, USA, jun 2018. IEEE Computer Society.

[111] Zhichao Yin and Jianping Shi. Geonet: Unsupervised learning of dense depth, optical flow and camera pose. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[112] Ravi Garg, Vijay Kumar B.G., Gustavo Carneiro, and Ian Reid. Unsupervised cnn for single view depth estimation: Geometry to the rescue. In

Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – ECCV 2016*, pages 740–756, Cham, 2016. Springer International Publishing.

[113] Matteo Poggi, Fabio Tosi, and Stefano Mattoccia. Learning monocular depth estimation with unsupervised trinocular assumptions. In *2018 International Conference on 3D Vision (3DV)*, pages 324–333, 2018.

[114] Fabio Tosi, Filippo Aleotti, Matteo Poggi, and Stefano Mattoccia. Learning monocular depth estimation infusing traditional stereo knowledge. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[115] Max Jaderberg, Karen Simonyan, Andrew Zisserman, and koray kavukcuoglu. Spatial transformer networks. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[116] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[117] Filippo Aleotti, Fabio Tosi, Matteo Poggi, and Stefano Mattoccia. Generative adversarial networks for unsupervised monocular depth prediction. In *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, September 2018.

[118] Nan Yang, Rui Wang, Jorg Stuckler, and Daniel Cremers. Deep virtual stereo odometry: Leveraging deep depth prediction for monocular direct sparse odometry. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[119] Huangying Zhan, Ravi Garg, Chamara Saroj Weerasekera, Kejie Li, Harsh Agarwal, and Ian Reid. Unsupervised learning of monocular depth estimation and visual odometry with deep feature reconstruction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[120] Diana Wofk, Fangchang Ma, Tien-Ju Yang, Sertac Karaman, and Vivienne Sze. Fastdepth: Fast monocular depth estimation on embedded systems. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 6101–6108, 2019.

[121] Xiaohan Tu, Cheng Xu, Siping Liu, Guoqi Xie, and Renfa Li. Real-time depth estimation with an optimized encoder-decoder architecture on embedded devices. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 2141–2149, 2019.

[122] Matteo Poggi, Filippo Aleotti, Fabio Tosi, and Stefano Mattoccia. Towards real-time unsupervised monocular depth estimation on cpu. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5848–5854, 2018.

[123] Dawei Sun, Shaoshan Liu, and Jean-Luc Gaudiot. Enabling embedded inference engine with ARM compute library: A case study. *CoRR*, abs/1704.03751, 2017.

[124] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation, 2013.

[125] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3), may 2008.

[126] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[127] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[128] Fayao Liu, Chunhua Shen, Guosheng Lin, and Ian Reid. Learning depth from single monocular images using deep convolutional neural fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(10):2024–2039, 2016.

[129] Nan Yang, Rui Wang, Jorg Stuckler, and Daniel Cremers. Deep virtual stereo odometry: Leveraging deep depth prediction for monocular direct sparse odometry. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[130] Jamie Watson, Michael Firman, Gabriel J. Brostow, and Daniyar Turmukhambetov. Self-supervised monocular depth hints. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[131] Timur Bagautdinov, Francois Fleuret, and Pascal Fua. Probability occupancy maps for occluded depth images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[132] Shijie Sun, Naveed Akhtar, Huansheng Song, Chaoyang Zhang, Jianxin Li, and Ajmal Mian. Benchmark data and method for real-time people counting in cluttered scenes using depth sensors. *IEEE Transactions on Intelligent Transportation Systems*, 20(10):3599–3612, 2019.

[133] Vinkle Srivastav, Afshin Gangi, and Nicolas Padoy. Human pose estimation on privacy-preserving low-resolution depth images. In Dinggang Shen, Tianming Liu, Terry M. Peters, Lawrence H. Staib, Caroline Essert, Sean Zhou, Pew-Thian Yap, and Ali Khan, editors, *Medical Image Computing and Computer Assisted Intervention – MICCAI 2019*, pages 583–591, Cham, 2019. Springer International Publishing.

[134] Edward Chou, Matthew Tan, Cherry Zou, Michelle Guo, Albert Haque, Arnold Milstein, and Li Fei-Fei. Privacy-preserving action recognition for smart hospitals using low-resolution depth images. *arXiv preprint arXiv:1811.09950*, 2018.

[135] Meng-Rong Lee and Daw-Tung Lin. Vehicle counting based on a stereo vision depth maps for parking management. *Multimedia Tools and Applications*, 78(6):6827–6846, 2019.

[136] R.I. Hg, P. Jasek, C. Rofidal, K. Nasrollahi, T.B. Moeslund, and G. Tranchet. An rgb-d database using microsoft's kinect for windows for face detection. In *2012 Eighth International Conference on Signal Image Technology and Internet Based Systems*, pages 42–46, 2012.

[137] Alessio Tonioni, Matteo Poggi, Stefano Mattoccia, and Luigi Di Stefano. Unsupervised adaptation for deep stereo. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[138] Art B Owen. A robust hybrid of lasso and ridge regression. *Contemporary Mathematics*, 443(7):59–72, 2007.

[139] Xiaoyang Guo, Hongsheng Li, Shuai Yi, Jimmy Ren, and Xiaogang Wang. Learning monocular depth by distilling cross-domain stereo networks. In Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss, editors, *Computer Vision – ECCV 2018*, pages 506–523, Cham, 2018. Springer International Publishing.

[140] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3213–3223, 2016.

[141] Stm32f767zit6-datasheet. https://www.st.com/resource/en/datasheet/stm32f767zi.pdf (last visited on 09/20/2019).

[142] Yang Wang, Peng Wang, Zhenheng Yang, Chenxu Luo, Yi Yang, and Wei Xu. Unos: Unified unsupervised optical-flow and stereo-depth estimation by watching videos. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[143] Eddy Ilg, Tonmoy Saikia, Margret Keuper, and Thomas Brox. Occlusions, motion and depth boundaries with a generic network for disparity, optical flow or scene flow estimation. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[144] Antonio Torralba, Rob Fergus, and William T. Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(11):1958–1970, 2008.

[145] Alessandro Capotondi, Manuele Rusci, Marco Fariselli, and Luca Benini. Cmix-nn: Mixed low-precision cnn library for memory-constrained edge devices. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 67(5):871–875, 2020.

[146] Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens van der Maaten, and Kilian Q. Weinberger. Multi-scale dense networks for resource efficient image classification, 2017.

[147] Valentino Peluso and Andrea Calimera. Energy-accuracy scalable deep convolutional neural networks: A pareto analysis. In Nicola Bombieri, Graziano Pravadelli, Masahiro Fujita, Todd Austin, and Ricardo Reis, editors, *VLSI-SoC: Design and Engineering of Electronics Systems Based on New Computing Paradigms*, pages 107–127, Cham, 2019. Springer International Publishing.

[148] Bert Moons, Bert De Brabandere, Luc Van Gool, and Marian Verhelst. Energy-efficient convnets through approximate computing. In *2016 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1–8, 2016.

[149] Muhammad Shafique, Rehan Hafiz, Muhammad Usama Javed, Sarmad Abbas, Lukas Sekanina, Zdenek Vasicek, and Vojtech Mrazek. Adaptive and energy-efficient architectures for machine learning: Challenges, opportunities, and research roadmap. In *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 627–632, 2017.

[150] A microbenchmark support library. https://github.com/google/benchmark (last visited on 01/18/2022).

[151] Zhaofeng Wu, Ding Zhao, Qiao Liang, Jiahui Yu, Anmol Gulati, and Ruoming Pang. Dynamic sparsity neural networks for automatic speech recognition. In *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6014–6018, 2021.

[152] Udo W. Pooch and Al Nieder. A survey of indexing techniques for sparse matrices. *ACM Comput. Surv.*, 5(2):109–133, jun 1973.

[153] Sharan Narang, Erich Elsen, Gregory Diamos, and Shubho Sengupta. Exploring sparsity in recurrent neural networks. *arXiv preprint arXiv:1704.05119*, 2017.

[154] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression, 2017.

[155] Jiahui Yu and Thomas S. Huang. Universally slimmable networks and improved training techniques. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[156] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[157] Mark Everingham, Luc Gool, Christopher K. Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *Int. J. Comput. Vision*, 88(2):303–338, jun 2010.

[158] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. Ssd: Single shot multibox detector. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – ECCV 2016*, pages 21–37, Cham, 2016. Springer International Publishing.

[159] Mingxing Tan and Quoc Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114. PMLR, 09–15 Jun 2019.

[160] Han Cai, Chuang Gan, and Song Han. Once for all: Train one network and specialize it for efficient deployment. *CoRR*, abs/1908.09791, 2019.

[161] Chiyuan Zhang, Samy Bengio, and Yoram Singer. Are all layers created equal? *CoRR*, abs/1902.01996, 2019.

[162] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2498–2507. PMLR, 06–11 Aug 2017.

[163] Ariel Gordon, Elad Eban, Ofir Nachum, Bo Chen, Hao Wu, Tien-Ju Yang, and Edward Choi. Morphnet: Fast and simple resource-constrained structure learning of deep networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[164] Jiahui Yu and Thomas S. Huang. Network slimming by slimmable networks: Towards one-shot architecture search for channel numbers. *CoRR*, abs/1903.11728, 2019.

[165] Ting-Wu Chin, Ari S. Morcos, and Diana Marculescu. Joslim: Joint widths and weights optimization for slimmable neural networks. In Nuria Oliver, Fernando Pérez-Cruz, Stefan Kramer, Jesse Read, and Jose A. Lozano, editors, *Machine Learning and Knowledge Discovery in Databases. Research Track*, pages 119–134, Cham, 2021. Springer International Publishing.

[166] Allan Skillman and Tomas Edsö. A technical overview of cortex-m55 and ethos-u55: Arm's most capable processors for endpoint ai. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–20, 2020.

[167] Mark Chen, Alec Radford, Rewon Child, Jeffrey Wu, Heewoo Jun, David Luan, and Ilya Sutskever. Generative pretraining from pixels. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 1691–1703. PMLR, 13–18 Jul 2020.

[168] Carlos Riquelme Ruiz, Joan Puigcerver, Basil Mustafa, Maxim Neumann, Rodolphe Jenatton, André Susano Pinto, Daniel Keysers, and Neil Houlsby. Scaling vision with sparse mixture of experts. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

[169] Chris Lattner, Jacques Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A compiler infrastructure for the end of moore's law. *arXiv preprint arXiv:2002.11054*, 2020.