On the Efficiency of Sparse-Tiled Tensor Graph Processing for Low Memory Usage

On The Efficiency of Sparse-Tiled Tensor Graph Processing For Low Memory Usage

Antonio Cipolletta, Andrea Calimera
Politecnico di Torino, 10129 Torino, Italy

***Abstract*—The memory space taken to host and process large tensor graphs is a limiting factor for embedded ConvNets. Even though many data-driven compression pipelines have proven their efficacy, this work shows there is still room for optimization at the intersection with compute-oriented optimizations. We demonstrate that tensor pruning via weight sparsification can cooperate with a model-agnostic tiling strategy, leading ConvNets towards a new feasible region of the solution space. The collected results show for the first time fast versions of MobileNets deployed at full scale on an ARM M7 core with 512KB of RAM and 2MB of FLASH memory.**

I. Introduction

The deployment of Deep Convolutional Neural Networks (ConvNets) on tiny devices, such as Micro-Controller Units (MCUs), encompasses several design issues. The most critical is the lack of memory resources, both for storing the weights of the model (i.e., the off-chip FLASH) and the temporary results produced during the forward pass of the model (i.e., the on-chip SRAM). Even the most powerful off-the-shelf MCUs for the IoT (such as the ARM M7 core [1] targeted in this work) come with very few MBs of FLASH (2 MB) and hundreds of KBs of SRAM (512KB), which is orders of magnitude smaller than that of mobile CPUs [2]. Moreover, the available memory is further reduced by other routines stored and processed in the background to ensure device operability.

Several optimization techniques have been proposed in the last few years. Most of them leverage the statistical nature of Deep Learning (DL) and the inherent algorithmic resilience of ConvNets to reduce the feature maps hosted by the SRAM (also called activations) or the weights stored in the FLASH. Seminal works include arithmetic precision scaling via quantization [3], structured and unstructured pruning [4], [5], resolution scaling and topology scaling via channel pruning (a.k.a. width multiplier) [6], [7]. Those techniques are data-driven and tend to degrade the model accuracy. Quantization and pruning are fine-grain knobs that allow accuracy to be recovered with iterative re-training stages, whereas resolution and topology scaling are coarser knobs for fast and aggressive memory compression that induce non-recoverable loss. A discriminant factor is the kind of memory where savings materialize: weights pruning generally works for the FLASH memory only, input resolution scaling is for SRAM, quantization and topology scaling affect both FLASH and SRAM. Therefore, which knobs to deploy should be weighted by the characteristics of the ConvNet and the hardware specifications.

Another class of techniques tackles the problem from a different angle, leveraging compilation strategies that are data-independent, training-free, and contribute to iso-accuracy optimizations. Examples are the scheduling of the tensor graph nodes to guarantee minimum activation footprint [8], or the acceleration of the arithmetic operators with local transformations [9], [10]. Since the weights set of the ConvNet model remains untouched, the savings brought are mostly on the activation maps, and hence the active SRAM consumed during inference.

The differences between the two kinds of strategies suggest that optimality can be achieved through effective co-operation, something partially shown in state-of-the-art works that proposed training and optimization pipelines with different knobs applied together [11], [12]. How to integrate data-independent techniques has received little attention and remains the weak ring of the chain, indeed. Compilation strategies aimed at optimizing the resource allocations often come too late in the pipeline, only after quantization, pruning, and topology scaling already set the trade-off between accuracy and memory consumption. As a side effect, some regions of the optimization space may remain unexplored.

The objective of this work is to fill this gap by showing a memory optimization pipeline that combines two techniques belonging to the two different classes of methods: data-driven and data-independent. We deploy sparsification to reduce the model size with minimal accuracy loss and tensor graph tiling to reduce the activation size without any accuracy loss. Our proposal is enabled by a functional-preserving rewriting procedure that works at the graph-level, handling tensors as abstract data objects, regardless of how they are actually packed, stored, and processed. This procedure opens to memory-efficient global graph transformations that can be applied with more degrees of freedom in the earlier stages of the optimization flow. A thorough assessment of MobileNets (v1 and v2) deployed on the NUCLEO-F767ZI board [1] powered by an ARM Cortex-M7 with 512KB/2MB of SRAM/FLASH shows sparse-tiled graphs are dominant in the memory-accuracy space. Compared to other compression pipelines that combine either data-driven techniques, i.e., sparsity, resolution, and topology scaling, or data-independent techniques, i.e., tensor tiling, the proposed strategy enables the deployment of new optimal configurations.

II. Background and Motivations

A. Static Graph Memory Allocation

ConvNets, and any tensor graph in general, are modeled by dataflow graphs (DFGs) where nodes represent the algebraic operators and edges the tensors. Fig. 1 shows a DFG portion of MobileNetV2 [7]. Tensors containing the weights are mapped sequentially in the non-volatile memory, the FLASH memory. Many MCUs have a direct access path to the FLASH that can be exploited at run time for fast data retrieving [1]. The volatile memory, usually an on-chip SRAM, hosts the activations and the buffer used as a scratchpad for auxiliary procedures, e.g., the im2col buffer for GEMM-based convolutions. To notice that only active maps, namely, those produced but not yet consumed by all dependent operators, need to be kept alive. The computational nodes are commonly processed sequentially following a topological order of the DFG. Therefore, a liveness analysis on the scheduled DFG estimates the amount of memory to be allocated. 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. Non-overlapping tensors, i.e., those with disjoint lifetimes, can share the same memory region, enabling reuse. As shown in Fig. 1, the sum of overlapping tensors in a given cycle sets the active memory, while the cycle with peak memory utilization defines the total SRAM footprint.

B. Data-Driven Memory Optimization Techniques

Arithmetic Precision. Precision scaling leverages an arithmetic relaxation of the operations to reduce the computational effort and to alleviate the memory bandwidth requirements. Quantizing the model from 32-bit floating-point to 8-bit fixed-point has become a standard for tiny devices, leading to 4x global memory reduction and up to 4x speed-up [3].

Pruning. Over-parametrized ConvNets can be simplified by dropping less important parameters at different levels of granularity. With *weight-pruning* [4], close-to-zero weights are zeroed until a pre-defined level of sparsity is reached. This sparsification enables the use of compression methods based on sparse data representations, like Huffman coding, leading to substantial FLASH savings. The *group-pruning* works at a higher level of granularity. Weights are pruned in blocks of a size such that the utilization of the parallel arithmetic units of the hosting hardware is maximized [13]. The main benefit is to reduce the model footprint with savings on the FLASH memory, whereas the speed-up may depend on the adoption of sparse computational kernels. At the coarsest granularity, *kernel-pruning* [5] drops entire convolutional kernels, with proportional savings for the FLASH memory, the activation footprint, and the number of operations.

Width Modulation. These methods play with the topology of the ConvNets. Indeed, entire convolutional filters are cut, reducing the cardinality of the inner feature maps. Specifically, the number of filters are scaled across all the layers according to a predefined ratio α, called the width multiplier [6]. Thin layers take less FLASH (as fewer channels imply fewer weights to store) and less SRAM (fewer features to be processed). Obviously, the information stored in the ConvNet is removed at a fast pace, leading to a substantial accuracy drop.

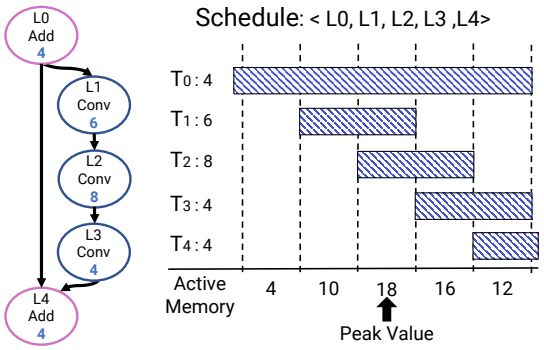


Fig. 1: Dataflow graph of a MobileNetV2 block [7] (left) and the conflict graph of its tensors T_i (right). Each node is labeled with an ID, its operation, the size of the output tensor.

Resolution Scaling. It reduces the spatial resolution of the intermediate features by scaling the resolution of the input patterns fed to the network [6]. High-resolution features may contain more fine-grain details that help to improve the prediction accuracy, which comes at the cost of larger SRAM usage and more operations processed. On the other hand, lower resolution features lighten the SRAM pressure but might affect accuracy severely. Moreover, the model footprint does not scale, that is, the same amount of FLASH is consumed.

Since the trade-off between accuracy degradation and memory savings is strictly related to the complexity of the task and data, all the above methods call for re-training. Moreover, one should consider that SRAM savings can be mainly obtained by width modulation and resolution scaling, and when the SRAM size is the dominant constraint, a sudden accuracy drop is a cost to pay for getting the model deployable. The main intention of this work is to show there exists a more effective option.

C. Data-Independent Memory Optimization Techniques

Operator-Level Transformations. Optimize the computing scheme of each neural operator by means of alternative multi-loop implementations that minimize memory usage. They can be operated automatically, using a code synthesis process [9], or manually, through a hardware conscious code restyling [14], [15]. The resulting code is custom-tailored to a specific platform, e.g., CPUs, GPUs, or spatial accelerators.

Graph-Level Transformations. Aim at orchestrating the flow of macro-operations rather than the inner code organization. They manipulate the DFG by removing, modifying, or adding nodes while preserving the overall functionality. Specifically, they make use of hand-crafted graph rewriting rules to reduce the peak memory consumption leveraging the algebraic properties of the operators [8]. Many prior works elaborated on the concept of operator fusion [16], by which the processing of chained operators is rearranged in a depth-first manner: the outputs of an operator are consumed by the next operator in the chain as soon as they get ready. The

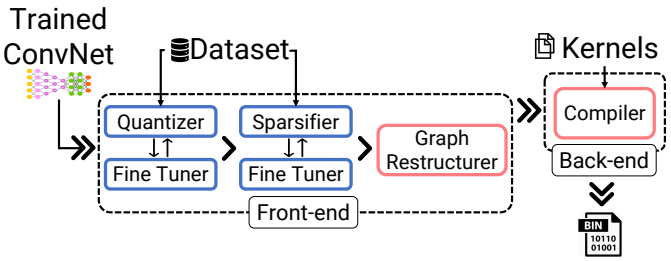


Fig. 2: The optimization pipeline implemented to translate the high-level description of a neural network into an optimized executable binary. The blue boxes indicate data-driven passes, while the red boxes data-independent passes.

resulting graph has a different topology due to the presence of fused operators. The authors of [17] introduced a framework that first identifies the operators that can be aggregated and then generates a fused code tailored for CPUs or GPUs. The layer fusion applies for convolutions followed by element-wise and pooling operators; therefore, it just covers specific patterns and cannot be generalized to any tensor graph. In [16], [18], more aggressive strategies were introduced, leveraging a custom memory architecture for fusing multiple convolutional operators.

Unlike the optimization methods discussed in the previous sub-section, these data-independent techniques are functional preserving and hence do not affect the ConvNet accuracy. This property motivates the idea behind our proposal: to achieve a better memory-accuracy trade-off backing up lossy statistical methods with lossless graph transformations. In doing this, and to make it applicable to general-purpose MCUs, we observed the need for a graph-level restructuring procedure able to (i) achieve fast memory reduction, greater or equal to that attainable with width modulation and resolution scaling, (ii) ensure generality, breaking any dependence from the ConvNet architecture, the kind of neural operators adopted, or the hardware specifications.

III. Optimization Pipeline

An overview of the optimization pipeline is shown in Fig. 2. The input of the pipeline is a pre-trained ConvNet, and the output is a binary file ready for the target device. The pipeline consists of two main stages: (i) the *front-end*, where the memory optimization happens; (ii) the *back-end* with the compiler. The *front-end* consists of two data-driven passes, quantization and sparsification, and the data-independent graph restructuring procedure. The *back-end* uses low-level routines extracted from a library of neural kernels optimized for the target device. The following subsections provide details on the memory optimization stages.

A. Data-Driven Passes

The quantizer reduces the arithmetic precision of weights and activations to 8-bits, following a linear scheme with power-of-two scaling to efficiently exploit the integer SIMD extension of the Cortex-M architecture. While the bitwidth is uniform for the entire network, the radix-points of the feature maps, the weights, and the biases are assigned layer-by-layer. Such values are calculated by minimizing the mean squared error between the original floating-point distribution and the quantized one. A fine-tuner is used to recover a possible accuracy loss [19]. The effect of the quantization step is a 4x reduction in both activation and model footprint, with no loss of accuracy.

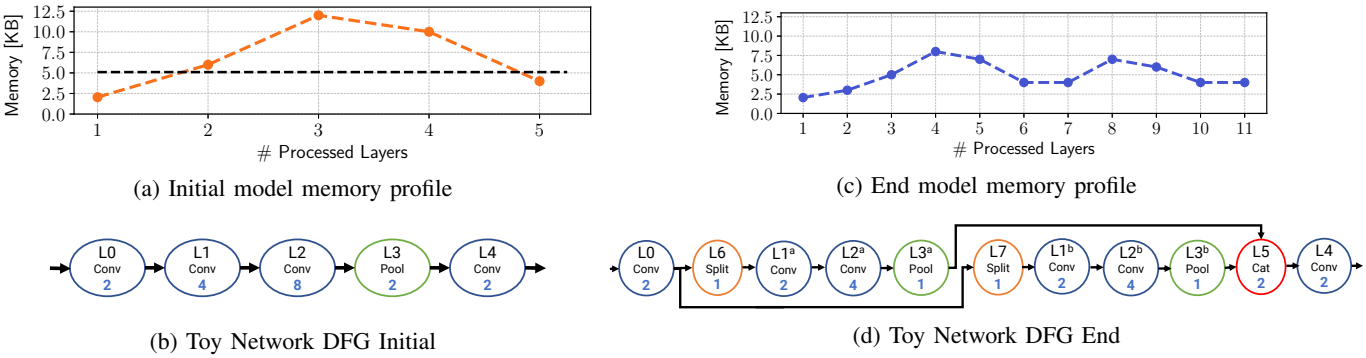
The quantized graph is then sparsified following the structured pruning strategy proposed in [13]. The weights are pruned in groups of two, such that the sparse matrix multiplication routine (spMM) can be efficiently implemented exploiting the 2-lane SIMD datapath of the Cortex-M architecture. The weight matrices are represented in a modified Compressed Sparse Row format (CSR) according to [13]. In particular, each sparse weight matrix is represented with three 1D vectors: the *nnz-values* storing the non-zero weights, the *j-idx* storing the first column index of each couple of non-zero elements, the *i-idx* storing the number of non zero elements in each row. Both *nnz-values* and *j-idx* vectors are accessed sequentially hence ensuring good cache locality. Moreover, by properly unrolling the spMM kernel's hot-loop, it is possible to fully exploit the vector load instructions and so the available bandwidth to the L1 data cache. As a result of both sparsification and compression, the memory footprint of the model reduces to approximately one and a half times the number of non-zero elements (some overhead due to the storage of the non-zero elements along with the metadata of the CSR format). Note that this pruning strategy does not reduce the activation footprint; 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 recover accuracy with a fine-tuning procedure [13].

B. Data-Independent Pass

The last step of the front-end is the graph-restructuring, which applies data-independent transformations on the tensor graph. Specifically, it minimizes the volume of concurrent tensors, and hence the overall allocated SRAM, by means of functional preserving topology rewritings that flatten the memory profile through tensor tiling.

The graph-restructurer first identifies the regions of the models contributing to the peak memory value, then extends these regions to include surrounding layers with fewer memory demands, and, finally, rewrites the selected layers as smaller independent branches that are less memory and computational dense. Each of these branches computes a tile of the output tensor by processing a tile of the original input tensor(s); the input tiles are obtained through the insertion of proper split operators, whereas the output tiles are concatenated (cat) to reconnect the rewritten sub-graph with the other nodes of the graph. The split and cat operators can be seen as the special nodes that create lightweight, independent processing paths between graph regions with a lower memory pressure.

The rewriting is functional preserving and it does not alter the weights of the operators; therefore, it does not require



(a) Initial model memory profile

(c) End model memory profile

(b) Toy Network DFG Initial

(d) Toy Network DFG End

Fig. 3: Example of graph restructuring on a sequential DFG.

access to the input dataset. This topology transformation could be interpreted as a generalization of *depth-first processing*, but, differently from previous works [16]–[18], it plays at a higher level of abstraction, managing tensors as abstract data objects independently from the on-device representation and the computing strategies adopted. Such characteristic breaks external dependencies imposed by software and hardware implementations, shifting the optimization in the front-end of the pipeline closer to data-driven optimizations.

To achieve memory saving, the graph-restructurer plays with the number of created branches and the boundaries of the restructuring region, i.e., the placement of the split and cat operators. The number of branches affects the activation footprint of the created paths; hence, it determines the savings achieved by processing sequentially each lightweight branch instead of the original monolithic path. If the restructuring region contains stencil operators, such as convolution and pooling, then some redundant computations and duplicated elements are added to make the processing paths independent. To select the restructuring region, we adopted a simple greedy strategy, which we illustrate through a toy example reported in Fig. 3. The DFG in Fig. 3b comprises a linear chain of operators. As depicted in Fig. 3a, the peak of the activation footprint (12 KB) is dictated by operator L2. We set a fraction of the peak memory value ($\gamma \cdot peak\_value$) as a threshold value, and we include as part of the restructuring region those layers whose active memory is above the threshold; the restructured region covers L1, L2, and L3 as shown in Fig. 3a. After the rewriting, the resulting DFG is shown in Fig. 3d; it has two independent branches, p_1=$\{L1^{(a)}, L2^{(a)}, L3^{(a)}\}$ and p_2=$\{L1^{(b)}, L2^{(b)}, L3^{(b)}\}$, each containing copies of the original layers playing with tensors halved in size. L6 and L7 are the two newly inserted operators in charge of halving the tensor produced by L0, whereas L5 concatenates the output slices produced by the created branches. Intuitively, the parallel branches are two times faster than the original monolithic path, and the overall latency is the same (except for a low overhead introduced by redundant operations due to split). Moreover, they are independent, namely, they consume and produce disjoint tensors that can be therefore scheduled in sequence, as shown in Fig. 3d. The memory profile of the tiled graph, which is reported in Fig. 3c, has a lower peak value, hence a smaller memory footprint (33% less the original DFG). Extensive analysis of state-of-the-art ConvNets (e.g., ResNets, VGG, Inception) has shown SRAM savings between 40% and 75% depending on the topology and the hyperparameters of the model.

IV. Results

A. Benchmarks and Deployment

We tested the proposed optimization pipeline on MobileNetV1 [6] and MobileNetV2 [7], two state-of-the-art ConvNets designed to meet a broad range of hardware constraints. A set of public configurations is freely available, including pre-trained configurations obtained under different settings of the width multiplier α and the input resolution ρ, the two main knobs adopted for scaling down the model size. Table I 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 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 ConvNets and tasks achieving similar results, not reported in this manuscript for the sake of space.

The tests were conducted on a NUCLEO-F767ZI board [1] 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 [15] with a modified in-house version of the SIMD-aware sparse matrix multiplication kernels presented in [13][1]. 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 [14], [15] and it does not prevent other low-level automatic code optimization, e.g. [9]. We adopted the GNU Arm Embedded Toolchain (version 6.3.1) for cross-compilation.

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

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

TABLE I: Baseline Characterization. Accuracy on ImageNet taken from tensorflow repositories[2].

B. Pipeline Set-up

The sparse networks are obtained following the method proposed in [13] 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 [13], [20], which demonstrates that 80% is a safe threshold to preserve 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. Finding the optimal ratio between the extension of the restructuring region and the number of slices is out of the scope of this paper.

C. Experimental Results

We do focus on two main extra-functional metrics: memory (SRAM and FLASH) and accuracy.

Fig. 4 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[3]. 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 (Table I). 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$50% lower SRAM usage and are compliant with the constraint. However, they exceed the available FLASH in most cases, which calls for aggressive width modulation ($\alpha \leq .50$). The bar-chart in Fig. 5 shows the latency collected on-board; the negative bars are for models not fitting the memory space. Sparse-tiled networks get much faster than dense only and tiled only nets; this can be observed for the first two networks on the left side of the chart ($\{.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 III, the restructuring procedure adds overhead due to redundant computations: $\approx$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. 6 and 7 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 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.

V. Conclusion

This work introduced a memory optimization pipeline combining statistical DL techniques with a functional-preserving graph restructuring procedure. The model parameters are group-pruned by a sparsification procedure, while regions of the model contributing to the peak memory consumption are identified and rewritten through tensor splitting and independent processing. The combined effect is a reduction of both RAM and FLASH footprint. The collected results show for the first time several configurations of MobileNets deployed on an ARM M7 core with 512KB/2MB of SRAM/FLASH. We expect that the joint combination of data-driven and data-independent optimizations will open to more efficient

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

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

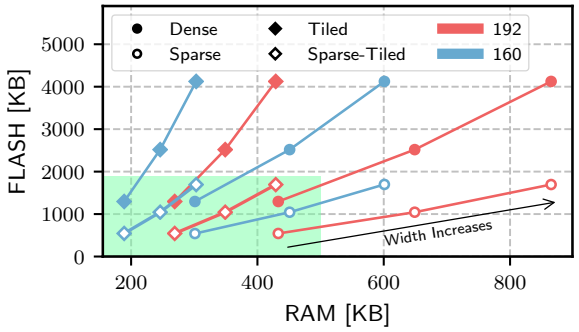


Fig. 4: FLASH and RAM requirements for different configurations of MobileNetV1. Width values are 0.50, 0.75, 1.00.

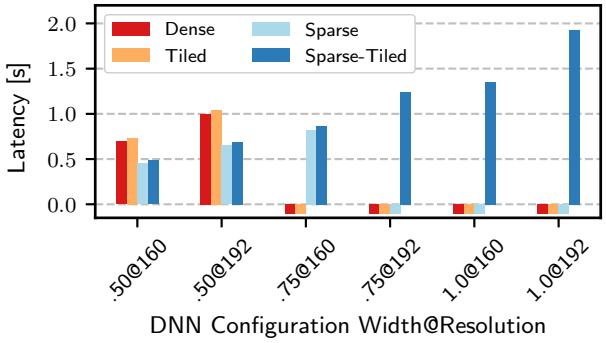


Fig. 5: Latency measurements for MobileNetV1. Bars sorted for accuracy, from least accurate (left) to most accurate (right).

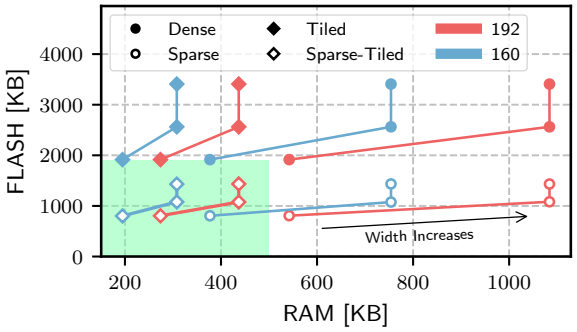


Fig. 6: FLASH and RAM requirements for different configurations of MobileNetV2. Width values are 0.50, 0.75, 1.00.

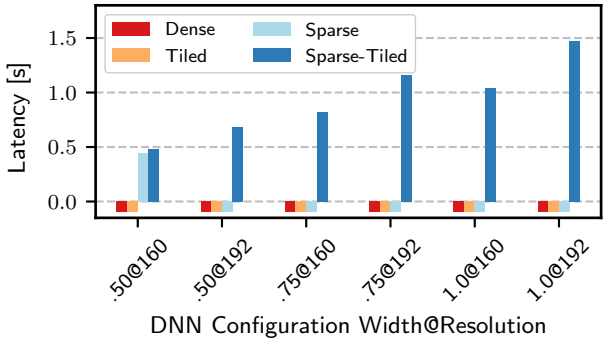


Fig. 7: Latency measurements for MobileNetV2. Bars sorted for accuracy, from least accurate (left) to most accurate (right).

tensor graph computing to meet the needs of next-generation intelligent applications deployed on tiny devices.

References

[1] Nucleo-f767zi. [Online]. Available: https://www.st.com/en/evaluation-tools/nucleo-f767zi.html

[2] Rpi3b+. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus

[3] B. Jacob *et al.*, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713.

[4] S. Han *et al.*, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *Proc. of the International Conference on Learning Representations*, 2016.

[5] H. Li *et al.*, "Pruning filters for efficient convnets," *CoRR*, vol. abs/1608.08710, 2016. [Online]. Available: http://arxiv.org/abs/1608.08710

[6] A. G. Howard *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[7] M. Sandler *et al.*, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proc. of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[8] B. H. Ahn *et al.*, "Ordering chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices," in *Proc. of Machine Learning and Systems*, 2020, pp. 44–57.

[9] T. Chen *et al.*, "Learning to optimize tensor programs," in *Proc. of the International Conference on Neural Information Processing Systems*, 2018, p. 3393–3404.

[10] Y. Wen *et al.*, "Taso: Time and space optimization for memory-constrained dnn inference," in *Proc. of the IEEE International Symposium on Computer Architecture and High Performance Computing*, 2020, pp. 199–208.

[11] H. Yang *et al.*, "Automatic neural network compression by sparsity-quantization joint learning: A constrained optimization-based approach," in *Proc. of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 2175–2185.

[12] F. Tung *et al.*, "Clip-q: Deep network compression learning by in-parallel pruning-quantization," in *Proc. of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 7873–7882.

[13] J. Yu *et al.*, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *Proc. of the International Symposium on Computer Architecture*, 2017, p. 548–560.

[14] S. Chetlur *et al.*, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[15] L. Lai *et al.*, "Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus," *arXiv preprint arXiv:1801.06601*, 2018.

[16] M. Alwani *et al.*, "Fused-layer cnn accelerators," in *Proc. of the IEEE/ACM International Symposium on Microarchitecture*, 2016, pp. 1–12.

[17] N. Weber *et al.*, "Brainslug: Transparent acceleration of deep learning through depth-first parallelism," *arXiv preprint arXiv:1804.08378*, 2018.

[18] K. Goetschalckx *et al.*, "Breaking high-resolution cnn bandwidth barriers with enhanced depth-first execution," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 323–331, 2019.

[19] A. Mishra *et al.*, "Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy," *arXiv preprint arXiv:1711.05852*, 2017.

[20] E. Elsen *et al.*, "Fast sparse convnets," in *Proc. of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 14 617–14 626.