Dataflow Restructuring for Active Memory Reduction in Deep Neural Networks

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

***Abstract*—The volume reduction of the activation maps produced by the hidden layers of a Deep Neural Network (DNN) is a critical aspect in modern applications as it affects the on-chip memory utilization, the most limited and costly hardware resource. Despite the availability of many compression methods that leverage the statistical nature of deep learning to approximate and simplify the inference model, e.g., quantization and pruning, there is room for deterministic optimizations that instead tackle the problem from a computational view. This work belongs to this latter category as it introduces a novel method for minimizing the active memory footprint. The proposed technique, which is data-, model-, compiler-, and hardware-agnostic, does implement a functional-preserving, automated graph restructuring where the memory peaks are suppressed and distributed over time, leading to flatter profiles with less memory pressure. Results collected on a representative class of Convolutional DNNs with different topologies, from Vgg16 and SqueezeNetV1.1 to the recent MobileNetV2, ResNet18, and InceptionV3, provide clear evidence of applicability, showing remarkable memory savings (62.9% on average) with low computational overhead (8.6% on average).**

I. Introduction

The astonishing achievements made by Deep Learning (DL) in the last decade is driving the growth of intelligent applications where Deep Neural Nets (DNNs), mainly Convolutional DNNs (CNNs), serve as the backbone of complex data reasoning tasks. With the intention to get more improved characteristics and better performance of the whole data-analytics stack, the quest for neural models that can be ported and distributed across different hardware architectures, such as high-end CPUs for office applications, low-power CPUs for the mobile segment, and tiny MCUs deployed onto the end-nodes of the IoT, accelerated the development of multi-stage pipelines where not just accuracy, but also non-functional metrics, memory utilization in particular, play as concurrent variables to optimize.

The compression of the model size, i.e., number and size of the learned weights, had always been considered the key aspect of the memory optimization problem. However, looking at how DNNs evolved, the reduction of memory taken by the intermediate results produced during inference, i.e., the activation maps of the hidden layers, has become even more concerning. That is due to two main factors. First, the resolution of multi-dimensional input data grew at a fast pace, with much more information to process; computer vision tasks are examples here. Second, the search for more efficient DNNs, often built with hardware-aware auto-ML tools, is leading to more irregular neural architectures [1] where the weight-to-activation ratio reduces substantially; for instance, SwiftNet [2], one of the winning submissions of the Visual Wake Words competition [3], takes 250K memory words for the weights and 200K for the activation maps[1].

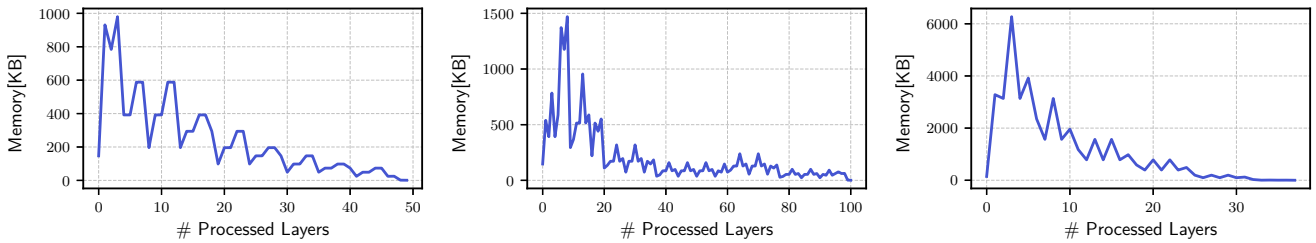
A way to attack the problem is to leverage the statistical nature of DL and the intrinsic redundancy of DNNs. The lowering of the arithmetic precision [4], [5], or the pruning of weak portions of the model that contribute less to the prediction accuracy [6], [7], are common options today. These techniques are data-driven and come with a high degree of uncertainty. They must be embedded as part of the training stage, increasing the cardinality of the optimization space with a negative impact on convergence. Moreover, they tend to wear away model accuracy as the information learned by the model is gradually removed until the design specs are met. Other methods belonging to a different class tackle the memory reduction problem from a computational perspective, namely, applying data-independent transformations during the compilation pipeline. For instance, by optimizing the execution flow of the tensor graph [8], or by locally accelerating the arithmetic operators [9]. The techniques are training-free, and hence model- and task-agnostic, they are fast and predictable, and can be superimposed to data-driven optimizations without further loss of information.

This work belongs to the second class as it introduces a restructuring methodology aimed at minimizing the volume of concurrent tensors processed within memory-critical regions of the dataflow graph. The idea was born from the simple observation that, for many DL models, the total active memory reaches its peak value for a limited amount of time during the processing of few layers[2], whereas it is remarkably lower before and after. The plots reported in Fig. 1 confirm the trend for three popular CNNs. This translates into a source of waste. The fixed-size blocks allocation strategies adopted in modern neural compilers reserve a memory pool big enough to host the activation maps of the highest demanding concurrent layers, but such space remains underutilized for most of the inference time. Intuitively, one could redistribute the memory peaks over less critical layers to alleviate the memory pressure and get more balanced profiles. That is precisely the purpose of the proposed strategy, implemented through (i) a tunable algorithm that seeks for critical, i.e., memory demanding, sub-graphs to be restructured and (ii) graph-rewriting procedures that implement functional preserving topology transformations based on the concept of tensor splitting and independent processing.

As key features, the proposed approach can work on any DNN, without the need for specialized code or components,

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

[2] In this work, the terms *layer and* operator are used interchangeably.



(a) ResNet18. Peak Memory 980KB. (b) MobileNetV2. Peak Memory 1470KB. (c) Vgg16. Peak Memory 6272KB.

Fig. 1: Memory profile of three different CNNs during the forward pass of a 3x224x224 image.

and is orthogonal to other data-/hardware-driven optimizations. In fact, differently from similar previous works, e.g., [10], our solution plays at a higher level of abstraction and with a modular granularity, managing tensors as abstract data-objects regardless how they are actually packed, stored, and processed. In this sense, it breaks free of those external dependencies imposed by software and hardware implementations, meeting flexibility and portability. A thorough assessment of the proposed technique on five state-of-the-art ConvNet architectures — VGG-16, ResNet18, InceptionV3, MobileNetV2, and SqueezeNetV1.1 — quantifies the memory reduction achieved (62.9% on average) as well as the computational overhead introduced (8.6% on average).

II. Background

DNNs, and in general tensor graphs, are modeled as dataflow graphs (DFGs) with nodes representing the computational operators while edges the data-dependencies. Each node is fed with at least one tensor and produces one output tensor. Fig. 2 graphically depicts an example for a residual block used in ResNet [11]. The DFG is static if the inner tensors have time-invariant sizes, and the execution flow is known at compilation-time; it is dynamic if the execution-flow, and so the size of the tensors, can change at run-time. This work deals with static graphs, leaving its dynamic extension for future works.

Both resource scheduling and binding run at compile time. Specifically, the compiler does the schedule through a topological sort operation, then it runs a liveness analysis for the tensors and estimates the amount of memory 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. Non-overlapping tensors can share the same portion of memory, enabling reuse. As shown in Fig. 2, the sum of overlapping tensors in a given cycle sets the active working memory, and the cycle with the highest memory requirement, i.e., the peak memory of the execution flow, defines the total memory footprint. Intuitively, both the graph topology and the size of tensors affect the amount of memory usage.

III. Related Works

This section gives an overview of existing data-independent optimization strategies. Table I shows a taxonomy based on the level of abstraction (i.e., *graph*-, *operator*-level), the main optimization objective (i.e., *Memory*, *Latency*), and the target hardware (i.e., *CPU*, *GPU*, *Custom* ASIC or FPGA). The

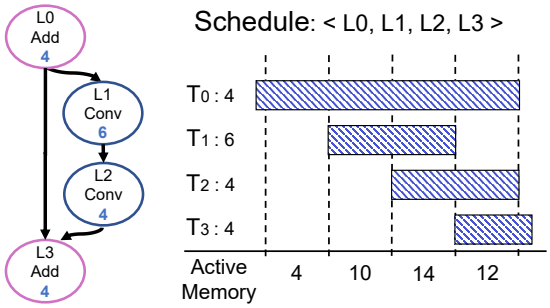


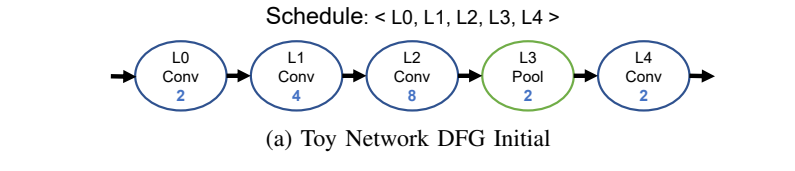
Fig. 2: Dataflow graph of a residual block [11] (left) and the conflict graph of its tensors T_i (right). Each node is labeled with an ID, its operation, the size of the output tensor.

level of abstraction reflects the hierarchical strategy adopted by modern DL frameworks and compilers [12], [13] to translate high-level dataflow graph into executable code. At the graph-level, the transformations are hardware-independent and aim at orchestrating the flow of macro-operations; at the operator-level, the transformations optimize the computing scheme of each operator, obtaining a code tailored to a specific platform, e.g., CPUs with shared memories and Single Instruction Multiple Data (SIMD) units, or GPUs with private memories and parallel cores, or accelerators with a distributed memory hierarchy and tightly coupled processing elements.

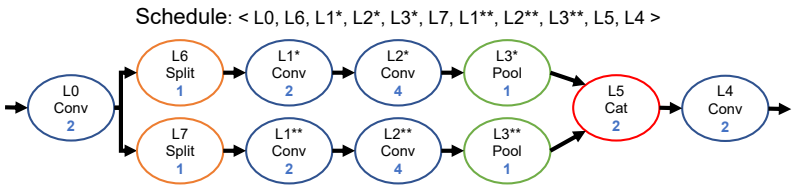
Graph-level transformations manipulate the dataflow by removing, modifying, or adding nodes while preserving the overall functionality. Specifically, they make use of hand-crafted graph rewriting rules to lower the latency of the scheduled dataflow [14] or to reduce the peak memory consumption [8] leveraging the algebraic properties of the operators. For instance, a graph pattern with a concatenation followed by a convolution can be rewritten as a sum of partial convolutions. Differently from these rule-based approaches, we exploit the spatial property of an operator rather than its algebraic behavior, thus obtaining solutions that work for any DL model.

	Graph-Level	Op-Level	Objective	Platform
[8]	x		M	Any
[14]	x		L	Any
[9], [15], [16]		x	L	CPU/GPU
[17]	x	x	L	CPU/GPU
[10], [18]	x	x	M	Custom
Ours	x		M	Any

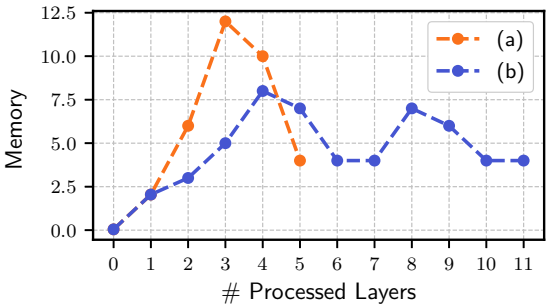
TABLE I: Taxonomy of related works.



(a) Toy Network DFG Initial



(b) Toy Network DFG End



(c) Initial and end model memory profiles

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

Operator-level transformations optimize the multi-loop implementation of a given operator in order to improve its performance, e.g., improve caches utilization. They can be operated automatically, using a code synthesis process [9], or manually, through a hardware conscious code restyling [15], [16].
Operator- and graph-level transformations complement each other and should not be seen as competitors. There exist hybrids that play as cross-level optimizations indeed. They are mainly based on the concept of operator fusion, by which the processing of subsequent chained operators is rearranged in a depth-first manner. For instance, given an operator, its output values (i.e., the pixels of the output map) are consumed by the next operator as soon as they are ready, and not after the whole output (i.e., the entire output map) is completed. The result can be conceived as an optimized graph with a restructured topology enabled by the availability of a custom intra-layer operator. In [17], the authors proposed a framework that first identifies the operators that can be aggregated, then it generates a fused code tailored for CPUs or GPUs. Due to the memory architecture of these platforms, the layer fusion brings savings only when applied to a convolutional with the subsequent element-wise and pooling operators. Therefore, the degree of optimization is strictly bounded by the types of operators in the dataflow. The authors of [10] and [18] introduced more aggressive strategies enabled by custom hardware architectures with special local buffers. In these cases, multiple convolutional operators can be fused together, empowering the reuse of the on-chip memory ([10]) and improving performance ([18]). The resulting graph restructuring is coupled with internal loop organization of the operator. By contrast, our proposal does formulate the restructuring process at the graph-level only, breaking the dependence from other cross-layer strategies, which, however, can be applied later, depending on the specific target platform and other constraints.

IV. Peak Memory Reduction

The objective of the proposed optimization is to seek, suppress, and distribute the memory peaks starting from a valid schedule of the DFG. This goal is accomplished through a methodology that first identifies those critical sub-graphs with the highest memory requirement, and then applies a localized, memory-driven topology restructuring based on tensor splitting and independent processing. The new topology preserves the same functionality of the original model, without further updates on the inner weights of the operators, but it shows many smaller independent branches that are less memory and computational dense. Each of these branches is built to work on a spatial subset of the original input tensor(s) and to contribute to the computation of a slice of the output tensor. The resulting output slices are concatenated to build the main outcome.

Before detailing the algorithmic implementation, we give an abstract view of the problem. For such purpose, Fig. 3a depicts the optimization process showing the DFG of a sequential model before (a) and after (b) the restructuring process, together with the resulting memory profiles (c). The original memory footprint (12 units in the example) is dictated by operator L2, and to lower it down encompasses the reduction of the active tensors handled by L2 itself. This reduction can be achieved by splitting L2 and propagating the transformation backward to L1 and forward to L3. The resulting DFG has two smaller (in terms of activations) independent branches: from L6 and L7 (the newly inserted operators in charge of splitting the tensor produced by L0), to L5 (which concatenates the output slices produced by L3* and L3**). Within the two branches, there are copies of the original operators that work on tensors halved in size. To be noted that the two branches are two times faster than the initial monolithic path (except for a small overhead discussed later in this section), hence latency gets almost the same. Even more important, they are independent; namely, they consume and produce disjoint tensors, and can be therefore scheduled in sequence to break lifetime conflicts and to maximize the memory reuse. As shown in the plot, the obtained memory profile shows a lower peak value and, thus, a smaller memory footprint (33% savings).

There are essential aspects to be considered in order to achieve the desired savings, mainly originating from a key step of the restructuring phase: the optimal placement of the fork-points, implemented by the split operators (L6 and L7), and the join-point(s), implemented by the concatenation operator (L5). They define the critical region to be restructured and their function is to create independent processing paths between surrounding points in the graph with a lower memory pressure (L0 and L4 in the example). Do not find the appropriate anchor points

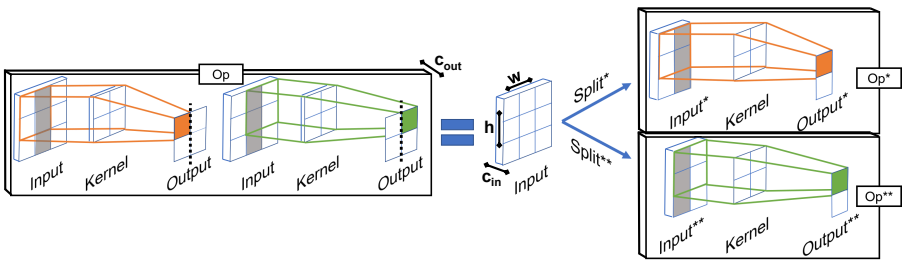


Fig. 4: Splitting a stencil operator into two smaller independent operators.

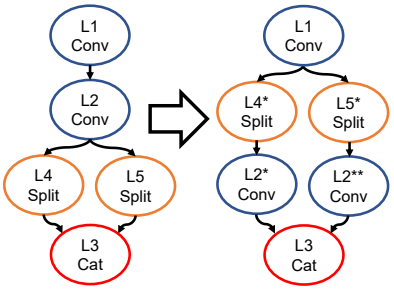


Fig. 5: Graph-rewriting that propagates the split operators in the DFG.

would make the restructuring failing. 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, the presence of fork-points and join-points creates further dependencies with neighboring tensors. In fact, the input slices (provided by the split operators) and the output slices (collected by the concatenation operator) must be kept alive when processing each independent branch. In addition, one should not forget that several operators, such as convolution and pooling, are stencils with overlapping windows that lead to redundant computations and duplicated elements when creating the independent branches. Making the produced branches independent of each other requires the replica of a few data from the original tensor. An example is shown in Fig. 4 for a stencil operation with a kernel window of size 2×2. The output tensor is split along the vertical direction, generating two smaller operators; each of them operates on a private portion of the original input tensor, but such two portions must have duplicated values (grey-shaded elements in the picture) to preserve the arithmetic equivalence. When the split operator is propagated across a path, those duplicated elements are computed several times by different operators, resulting in redundant computations. Where the fork- and join-points are placed affects the length of the independent branches, hence the number of redundant operations. Last but not least, the number of independent branches, i.e., the number of splits, plays an important role in determining the potential savings and the cost of the fork- and join-points.

Based on these considerations, it is evident that a multi-objective optimization approach is needed. However, formulating the problem in a closed-form might result impractical due to the many hyper-parameters of the graph and the huge cardinality of the search space. Therefore, we provide the restructuring algorithm that can serve as an engine for different greedy optimization strategies (one of which will be shown as part of the experimental section).

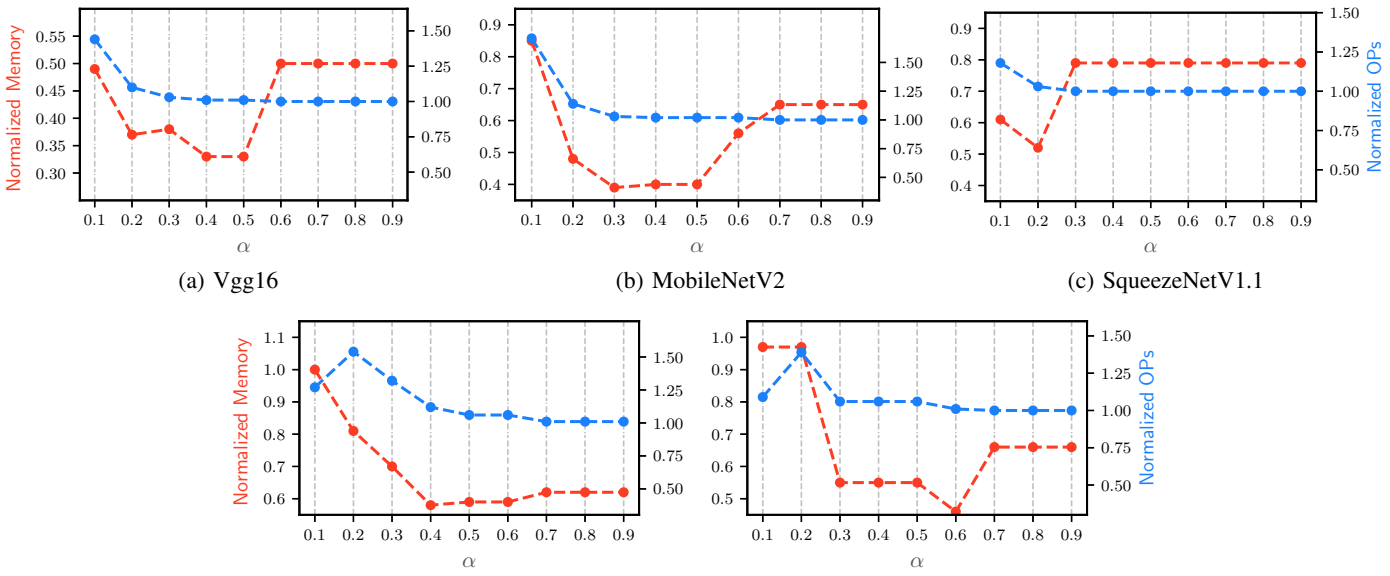
A. Restructuring Algorithm

Algorithm 1 reports the pseudocode of the proposed restructuring algorithm, named RESTRUCTURE. The input parameter α is a scalar controlling the extension 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

Algorithm 1 Restructuring Algorithm

```
1:  procedure RESTRUCTURE(DFG, α, n_slices)
2:      critical_set ← explore(DFG, α)
3:      subDFG ← subgraph(DFG, critical_set)
4:      reverseVisit(subDFG, n_slices)
5:  end procedure
6:
7:  procedure EXPLORE(DFG, α)
8:      schedule ← sort(DFG)
9:      lifetimes ← getLifetimes(DFG, schedule)
10:     memory_profile ← getMemProfile(schedule)
11:     peak_memory ← max(memory_profile)
12:     for all node ∈ DFG.nodes do
13:         criticality[node] ← computeCriticality(
14:                                 lifetimes, node, memory_profile)
15:     end for
16:     critical_set ← {node ∈ DFG.nodes,
17:                         s.t. criticality[node] == peak_memory}
18:     while ∃ node ∈ fanin(frontier(DFG, critical_set))
19:             s.t. criticality[node] ≥ α · peak_memory  do:
20:         critical_set ← critical_set ∪ {node}
21:     end while
22:     while ∃ node ∈ fanout(frontier(DFG, critical_set))
23:             s.t. criticality[node] ≥ α · peak_memory  do:
24:         critical_set ← critical_set ∪ {node}
25:     end while
26:
27:     return critical_set
28: end procedure
29:
```

the tensors. At first, the EXPLORE procedure is invoked to seek the set of nodes included in the critical sub-graph (line 2), i.e., those to be restructured. Here is where the fork- and join-points discussed in the previous sub-section get defined. Then, the same nodes are projected on the DFG, and the critical sub-graph is isolated (line 3). Finally, the sub-graph is visited in reverse topological order operating the graph-rewriting (lines 4). It is during this stage that nodes are split, and the independent branches are created. 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. 5. To compute automatically the parameters of the split operators, we developed a symbolic evaluation engine on top of a domain-specific language (DSL) that associates a functional specification to each DL operator. In the prologue of the EXPLORE procedure (lines 8-9), the



(a) Vgg16 (b) MobileNetV2 (c) SqueezeNetV1.1

(d) ResNet18 (e) InceptionV3

Fig. 6: Normalized memory and number of operations for $\alpha \in [0.1, 0.9]$ and n_slices=$\{2h, 2w\}$.

DFG is scheduled and the lifetime of each tensor extracted. These data are used to calculate the total working memory and the peak value (lines 10-11). Each node is labeled with a criticality index equal to the total amount of memory active during its processing (lines 12-15). All the nodes with a criticality index equal to the peak memory take part to the initial critical set (lines 16-17). The frontier of the critical set is made up of those critical nodes with in-/out-degree edges from/to nodes outside the critical set. The extension of the critical set continues using α as the driving parameter. Specifically, nodes at the boundary of the frontier 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, namely, when the remaining nodes have a criticality lower than $\alpha \cdot peak\_value$. Therefore, α can be used to control the extension of the critical region, i.e., the placement of the fork- and join-points, and hence the efficiency of the restructuring.

V. 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. Within the selected benchmarks, VGG-16 is taken as representative of large and computationally intensive networks with conventional single-layer connectivity; ResNet18, with residual connections, and InceptionV3, with inception blocks, can represent models with irregular connectivity; MobileNetV2 and SqueezeNetV1.1 are more compact networks designed for mobile platforms. We provide a parametric analysis taking into consideration the two main parameters of the restructuring algorithm (Sec. IV-A): 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.

A first set of the collected results is reported in Fig. 6. The plots show the normalized peak memory (in red) and the normalized number of operations (in blue) as functions of α under a default value for n_slices=$\{2h, 2w\}$. As a general trend, the memory usage gets smaller with α, until it reaches a global minimum, indicated by α_{opt}. Tab. II collects a summary of such optimal points reporting the memory savings achieved and the corresponding computing overhead (normalized with respect to the original models). Averaging over the five CNNs, memory savings reach 54.3%, with an 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, while it gets slightly worse on ResNet18 (41.6% of savings, 11.9% overhead) due to residual blocks that complicate the topology with more overlapping tensors and more constraints to the restructuring process. To notice that in four networks (i.e., all but SqueezeNetV1.1), configurations with α=0.9 already achieve a significant improvement (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.

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

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

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

TABLE III: Computational overhead and memory saving for the optimal setting of n_slice when $\alpha = \alpha_{opt}$.

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

Table III 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 Tables III and II demonstrates that freeing the values in n_slices leads to achieving larger memory savings (62.9% vs. 54.3% on average) at the cost of some computational penalty (8.6% vs. 4.1% on average). As 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 the exploration of various settings is fast and efficient. According to the measurements of our single core implementation on a machine powered by an Intel i7-8700K, the sweep of the restructuring algorithm over nine α values completes in $82s$ for SqueezeNetV1.1 (fastest) and $375s$ for InceptionV3 (slowest). 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 DNN structures are still as 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 [9] or vendor-specific libraries [15], [16], freeing the user from the considerable burden of developing additional operator-level optimizations for all possible target platforms.

VI. Conclusion

This work introduced a functional-preserving graph restructuring technique to reduce the memory footprint of the hidden activation maps of a Deep Neural Network. Regions of the model contributing to the peak memory consumption are identified and rewritten by means of tensor splitting and independent processing. A thorough analysis conducted on a representative set of DNNs demonstrates broad applicability, reporting remarkable memory savings (62.9% on average) with low computational overhead (8.6% on average). We expect that the joint combination of the proposed restructuring process with other graph- and operator-level transformations, together with alternative search strategies, will open to further optimizations and more efficient tensor graph computing.

References

[1] H.-P. Cheng *et al.*, "Msnet: Structural wired neural architecture search for internet of things," in *Proc. of the IEEE Conference on Computer Vision Workshops*, 2019, pp. 2033–2036.

[2] H.-P. Cheng *et al.*, "Swiftnet: Using graph propagation as meta-knowledge to search highly representative neural architectures," *arXiv:1906.08305*, 2019.

[3] A. Chowdhery *et al.*, "Visual wake words dataset," *arXiv:1906.05721*, 2019.

[4] B. Jacob *et al.*, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," *arXiv:1712.05877*, 2017.

[5] V. Peluso *et al.*, "Integer ConvNets on Embedded CPUs: Tools and Performance Assessment on the Cortex-A Cores," in *Proc. of the IEEE Conference on Electronics, Circuits and Systems*, 2019, pp. 598–601.

[6] J. Yu *et al.*, "Scalpel: Customizing DNN pruning to the underlying hardware parallelism," in *Proc. of the ACM/IEEE International Symposium on Computer Architecture*, 2017, pp. 548–560.

[7] M. Grimaldi *et al.*, "Optimality Assessment of Memory-Bounded ConvNets Deployed on Resource-Constrained RISC Cores," *IEEE Access*, vol. 7, pp. 152 599–152 611, 2019.

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

[9] T. Chen *et al.*, "Learning to optimize tensor programs," in *Advances in Neural Information Processing Systems*, 2018, pp. 3389–3400.

[10] 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.

[11] K. He *et al.*, "Deep Residual Learning for Image Recognition," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[12] T. Chen *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. of the 13th USENIX Symposium on Operating Systems Design and Implementation*, 2018, pp. 578–594.

[13] N. Rotem *et al.*, "Glow: Graph lowering compiler techniques for neural networks," *arXiv:1805.00907*, 2018.

[14] Z. Jia *et al.*, "Optimizing DNN Computation with Relaxed Graph Substitutions," in *Proc. of the Conference on Systems and Machine Learning*, 2019, pp. 27–39.

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

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

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

[18] Q. Xiao *et al.*, "Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs," in *Proc. of the ACM/EDAC/IEEE Design Automation Conference*, 2017, pp. 1–6.