Breaking High Resolution CNN Bandwidth Barriers with Enhanced Depth-First Execution

Koen Goetschalckx, Marian Verhelst
koen.goetschalckx@kuleuven.be, marian.verhelst@kuleuven.be
MICAS, Department of Electrical Engineering (ESAT),
KU Leuven, 3000 Leuven, Belgium

October 17, 2019

Abstract

Convolutional Neural Networks (CNNs) now also start to reach impressive performance on non-classification image processing tasks, such as denoising, demosaicing, super resolution and super slow motion. Consequently, CNNs are increasingly deployed on very high resolution images. However, the resulting high resolution feature maps pose unseen requirements on the memory system of neural network processing systems, as on-chip memories are too small to store high resolution feature maps, while off-chip memories are very costly in terms of I/O bandwidth and power. This paper first shows that the classical layer-by-layer inference approaches are bounded in their external I/O bandwidth vs. on-chip memory trade-off space, making it infeasible to scale up to very high resolutions at a reasonable cost. Next, we demonstrate how an alternative depth-first network computation can reduce I/O bandwidth requirements up to >200× for a fixed on-chip memory size or, alternatively, reduce on-chip memory requirements up to >10000× for a fixed I/O bandwidth limitation. We further introduce an enhanced depth-first method, exploiting both line buffers and tiling, to further improve the external I/O bandwidth vs. on-chip memory capacity trade-off, and quantify its improvements beyond the current state-of-the-art.

1 Introduction

With convolutional neural networks (CNNs) exceeding human performance on the ImageNet classification challenge, focus is shifting to alternative tasks. In many of those upcoming applications, higher resolution images are either required or preferred. Object detection, for instance, could equally well detect objects farther away or in a wider view by using higher resolutions. Moreover, CNNs running at high resolution can be used to replace traditional imaging pipeline steps such as demosaicing [1], (combined) noise-reduction [1], bokeh

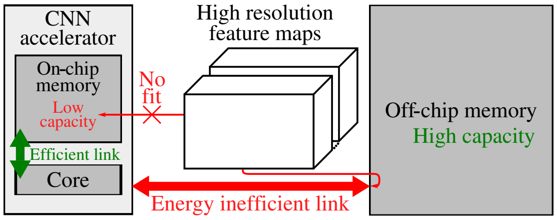


Figure 1: As on-chip memory can not hold high resolution feature maps, inefficiently linked off-chip memory is used.

effects and super resolution (upscaling) [2,3]. These exploitations of CNNs show opportunities to mitigate the need for very high quality imaging hardware and/or achieve enhanced output image quality. As a result, the future promises a strong rise in the deployment of CNNs running on high resolution images, such as 720p HD or 4k UHD.

However, using higher resolutions for input images also results in larger internal feature maps within the CNN, being the input and output tensors of each CNN layer. CNN accelerators typically have on-chip SRAM memories which are orders of magnitude smaller than such large feature maps. In realistic area constraints, the on-chip memory capacity can be pushed to one or a few MBs. Yet, a modest 720p RGB image (2.8 MB) already fills such capacity, whereas typical intermediate feature maps have ≥64 channels and even higher resolutions are preferred.

Hence, CNN processors are traditionally paired with off-chip memory, as shown in Fig. 1. Unfortunately, the I/O-communication and physically long data link to this memory results in a high energy usage and relatively low bandwidth. Several strategies have been reported to reduce the amount of required off-chip accesses by smartly deciding when to have which parts of the feature maps and the CNN model on-chip [4–10]. For instance, [4] uses tiling to optimize data locality and [6] introduced an analysis framework and a new 'Row-Stationary Plus' dataflow, which allows tiling of all dimensions. This involves a complex trade-off of several design and network mapping parameters, yet can bring savings of one to several orders of magnitude on bandwidth or on-chip memory requirements.

However, those works still rely on a layer-by-layer processing approach, wherein neural network layers are processed sequentially, i.e. the next layer only starts when the previous one is completely executed. This paper shows that even under the most optimistic, best-case assumptions and dataflow schemes, additional orders of magnitude can be gained for high resolution CNNs by switching from this layer-by-layer approach to a depth-first execution, explained in sections 3 and 4. By already using feature maps before they are fully completed, the depth-first approach removes the necessity to store and fetch complete intermediate feature maps, positively impacting on-chip memory needs and I/O bandwidth requirements. As will be discussed in section 7, few previous works [3, 11–13] have explored some form of depth-first CNN processing, but lack a broad comparison to state-of-the-art layer-by-layer processing. This work analyses these relative gains for several benchmarks. It further introduces an enhancement of the depth-first CNN processing towards additional savings, exploiting tiling in combination with line buffers. Moreover, an analysis is done of optimal cut and tiling combinations of the networks, and finally a benchmarking of this enhancement against the existing methods concludes the paper.

The main contributions of this paper are as such:

1. Section 2 describes a **lower bound in the off-chip bandwidth vs. on-chip memory capacity space for layer-by-layer CNN execution**.

2. Section 6 uses this bound to **quantify the improvement of depth-first over layer-by-layer** approaches on specific emerging benchmarks.

3. We further propose an **improvement to the depth-first approach using a combination of tiling and line buffers** in section 4, and compare it to the state-of-the-art in section 7.

4. By analyzing the optimal combinations of depth-first network stacks and tiling, section 6 **gives insights** in how to efficiently deploy the proposed technique.

2 Layer-by-layer CNN processing

2.1 On-chip memory capacity shortage

The bulk of neural network processing assumes a layer-by-layer processing approach [4–10]. This technique is common, as it is easily written in parallel programming frameworks and executed on parallel processors such as GPUs. Such layer-by-layer inference typically consists of:

1. loading the first layer's input feature map (i.e. the network's input) to on-chip memory,

2. loading the model (kernel weights) of the first layer to on-chip memory,

3. based on those two inputs, calculating the first layer's output feature map, which is the second layer's input feature map, and storing it on-chip,

4. loading the model weights of the second layer on-chip,

5. calculation the output feature map of the second layer,

6. repetition for the remaining layers,

7. and finally offloading the output of the last layer off-chip as the final network output.

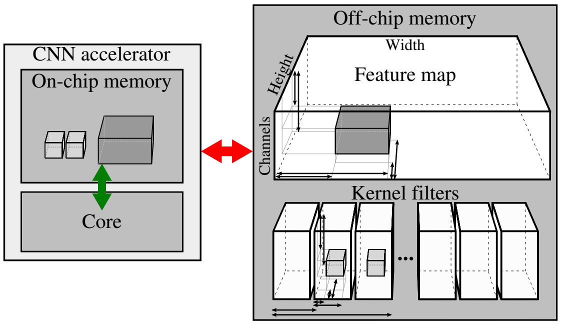


Figure 2: In traditional layer by layer tiling, a tile of the complete feature map and the kernel is copied from off-chip memory to on-chip memory, where it is then used for calculation. For each feature map (kernel), 6 (8) indices (↔) have to be set for all times in such a way that the usage of the inefficient link between the accelerator and off-chip memory (red arrow) is minimized.

This method fails, however, when the on-chip memory capacity is too small to contain the complete input feature map, complete output feature map, and model of a layer all at once. The severity of this problem increases with the input resolution. E.g.: an 8-bit, single channel, 1280×720 pixels (720p HD) large feature map is about 922 kB large, which is of the same order of magnitude as on-chip memory capacities. Much higher resolutions are however desired for the aforementioned applications. Moreover, feature maps typically contain tens to hundreds of channels. Thus, feature maps of realistic networks largely exceed on-chip memory capacities.

2.2 Reducing layer-by-layer on-chip memory usage by tiling

To accommodate for this, state-of-the-art implementations [4–9] constrain on-chip memory usage through iterative storage of contiguous parts, or ‘tiles’, of feature maps and weight kernels. Specifically, these tiles are fetched from an external memory and are temporarily stored on-chip. This allows to compute (intermediate results of) (parts of) the output feature map, which can then either be kept on-chip or be exported to external memory. Later, when they are required for completion or for the next layer, they are fetched back from off-chip memory. Of course, discarding any data that will be required again later has an external memory bandwidth cost.

To minimize this external memory bandwidth cost, the locations and dimensions of the tiles in the layer’s input feature map, output feature map, and kernel during the layer’s execution are optimized extensively. Specifically, for every single network layer, 12 tiling parameters can be tuned – 4 for the spatial location and size of the tiles in the input and output feature map, $2{\cdot}2{=}4$ for the ranges of input and output channels, and 4 more for the spatial location and size of the tiles in the kernel – as well as the storage locations of the tiles in on-chip memory. This results in a complex external bandwidth minimization problem, which is constrained to require no more on-chip memory than available. Extensive studies have been performed in this regard, e.g.: [4–7, 9, 10].

2.3 Layer-by-layer lower bound

Despite the complexity of this tiling optimization problem, we can define a lower bound in the external I/O bandwidth vs. on-chip memory trade-off space by defining some best-case assumptions:

1. **All feature data is loaded on-chip no more than once per layer**. Thus, we assume a tiling method that is able to I/O avoid multiple transfers of the same features.

2. **Weight kernel sizes are negligible** compared to feature maps as the focus is on high resolution image applications. Hence, our lower bound optimistically assumes they incur no external bandwidth nor on-chip memory cost.

3. **At the end of executing a layer, the on-chip memory is completely filled with features.** Although most feature maps are much too big to fit in on-chip memory, our lower bound assumes that the amount of features that can be retained in the on-chip memory are not send off-chip, but are again consumed for the next layer computation. The remaining part of the feature map is stored off-chip.

4. **Skip or residual connections come at no cost**. For our optimistic lower bound, we optimistically assume that skip or residual connections incur no extra external I/O memory bandwidth or on-chip memory usage, so that no trade-offs between storage of the connection's feature map vs. in-between feature maps need to be considered.

5. Batchnorm is fused in the layer's weights and activations are applied as soon as possible, so that **batchnorm and activation layers come for free** in the sense that they have no contribution to the external I/O bandwidth, nor to the used on-chip memory capacity.

Under these assumptions, the external I/O bandwidth lower bound for all tiling methods is simply the sum of the network's inputs, the network's outputs, and two times[1] all intermediate features that are not excluded by 3).

Fig. 5 plots this bound in terms of required I/O bandwidth in function of on-chip memory capacity for two resolutions, 720p HD and 4k UHD, and three CNNs [1, 2, 14]. Section 6 further discusses these CNNs and the lower bound results.

[1] off-loaded as output of a layer + fetched as input of the next layer

3 Line buffer-based depth-first CNN processing

This section contains a thorough explanation of the state-of-the-art depth-first approach with line buffers [12], which will be analyzed and enhanced further. The key idea of this computational approach is to limit the lifetime of intermediate feature map data items as much as possible.

In the following discussion, the term ‘pixel’ will be used to refer to all features from different channels but of a single spatial location in a feature map. The goal of the depth first approach is here to keep only very few pixels ‘alive’ at any given moment in time, defined as calculated and still needed as input for further calculations. This avoids the need to store large amounts of pixels at once and allows to keep all relevant pixels of intermediate feature maps in a relatively small on-chip memory. As such, data does not continuously has to be off-loaded to external memory, hence saving drastically on the external I/O bandwidth.

To this end, subsection 3.1 first explains the method for layers with kernels that operate on feature map patches of size 1×1 (with possibly C_{in} channels). Examples include activation, batchnorm, and pointwise $1 \times 1 \times C_{in}$ CNN layers. These are layers in which all features of different channels of a single pixel in the output feature map only depend on (all channels of) the pixel in same location in the input feature map. Note that existing software libraries already often integrate the former two in the preceding layer. Hence, these libraries already effectively implement a simple depth-first approach across a shallow stack of layers.

Next, subsection 3.2 describes using line buffers to support kernels operating on $k \times k$ pixel patches. Examples are regular 3D convolutional kernels, or depthwise $k \times k \times 1$ filter kernels. In this case, each pixel in the output feature map also depends on neighboring pixels in the input feature map.

Finally, 3.3 points out the shortcomings of this approach and as such motivates using tiling as explained in section 4.

3.1 1×1 patch layers

This paper first studies depth-first processing of layers that operate on feature map patches 1×1 pixels (with possibly C_{in} channels).

The goal of depth-first processing is now to achieve a minimal lifetime of all intermediate pixels. To this end, a newly available pixel $P_{new,i}$ in the input feature map FM_i of a layer L_i is immediately used to calculate a new pixel $P_{new,i+1}$ of the layer's output feature map FM_{i+1}. If $P_{new,i}$ has influence on only $P_{new,i+1}$ and no other pixels in the output feature map, as is the case in a 1×1 layer, it will not be required again. Hence, it can be discarded from on-chip memory almost immediately after calculation and no more than one pixel is ever ‘alive’ at the same time. Thus, only very little on-chip memory is needed: just enough to be able to store the largest single pixel. In Fig. 3, layer 1 shows an example of this: whenever a new input pixel ① is available, it is directly consumed to calculate a new pixel ② in FM_1 and is then immediately discarded.

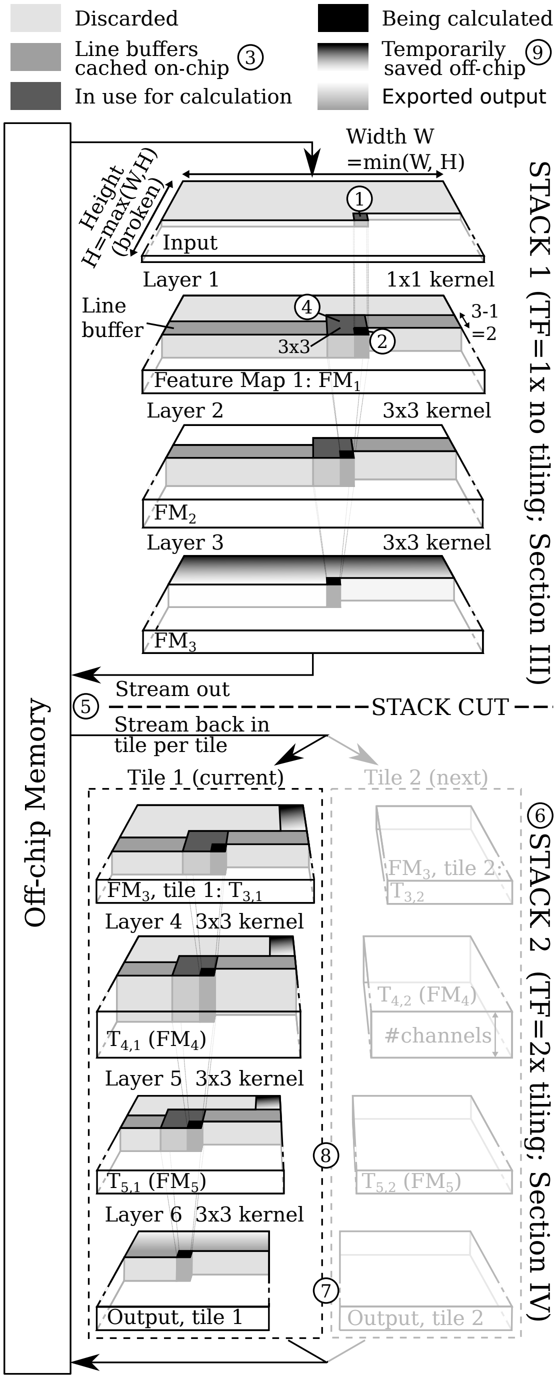


Figure 3: The depth-first approach, with (stack 2) and without (stack 1) tiling (subsection 4). Pixels within a tile are immediately propagated through a stack. At the end of a stack, they are send to an off-chip memory and then refetched when executing the next stack. Note that feature maps are not drawn to scale and typically much larger.

3.2 Exploiting line buffers for $k \times k$ patch layers

However, many neural network layers operate on larger, $k \times k$ pixel patches of their input feature maps, such as layer 2 in Fig. 3. Examples are the $3{\times}3{\times}C_{in}$ kernels now commonly used in networks such as DMCNN-VD [1] and SRGAN [2]. Consequently, $P_{new,i}$ will be needed in the computation of multiple pixels in the layer's output feature map FM_{i+1}. In this case, the pixel value $P_{new,i}$ has to be stored temporarily until its last use.

Our work stores these values in on-chip cache by using a scanning line buffer approach. To have a $k \times k$ patch available when a new pixel arrives, on-chip memory should always hold $k-1$ full lines of pixels plus $(k-1)$ pixels of every feature map in the stack of layers in memory. The lines are oriented over the shortest spatial distance of the input feature map FM_i. The length of these lines thus equals $\min(W, H)$ with W the width and H the height of FM_i. This results in an on-chip storage requirement of

$$N_P = (k-1) \times \min(W, H) + (k-1) \tag{1}$$

full pixels per feature map, indicated by the 'line buffers cached on-chip' shade ③ in Fig. 3.

With these pixels in memory, all pixels in the patch in which a newly arrived pixel is always the bottom right pixel ④ remain available. This allows to apply the $k \times k$ convolutional kernel on this patch without further I/O feature map fetching. After this evaluation, the top left pixel from the patch will not be required again and is replaced by the just arrived bottom right pixel. Consequently, the on-chip line buffer again holds the last N_P pixels.

For each single layer in the depth-first layer stack, there is thus always a patch ready for evaluation. Indeed, the just executed evaluation of layer L_i created a new pixel in the next layer L_{i+1}'s input feature map. L_{i+1} also has an input line buffer according to the size of its weight kernel, where newly computed pixels are added and computation fires as soon as all input data for the next patch is available. When the pixel computation is fully propagated through the depth-first stack, a final output pixel can be streamed to off-chip memory, and execution continues at the first layer with its next input pixel.

At the cost of storing just a few lines of pixels for each feature map in a stack, this method avoids external I/O communication of any feature internal in the stack. The only external I/O communication is for the input and output feature maps of the depth-first layer stack.

3.3 Stacking limits

The approach described above allows to stack many consecutive neural network layers and to compute their outputs in a depth-first way without any I/O communication for external loading or storing of intermediate features. Yet, the depth of the stack is limited by the size of the on-chip cache. Indeed, every stacked layer operating on $k \times k$ patches adds an on-chip memory requirement requirement of ($\approx k-1$) lines of pixels. If, by adding another layer to the stack, the total required on-chip memory exceeds the available on-chip memory capacity, two solutions can be applied:

1: **The stack can be ended ('cut').** The whole network can be split into multiple independent depth-first stacks, in between which feature data is streamed back to/from external memory. This can e.g. be seen in between layers 3 and 4 in Fig. 3, where the first stack ends by streaming features into the off-chip memory and back into the chip when executing the next stack of layers ⑤. This of course incurs an external I/O bandwidth cost. Hence, this allows an on-chip memory size vs. external I/O bandwidth trade-off.

2: **Tiling can be used** to decrease the buffer sizes within a stack. The following section elaborates further on this.

4 Tiling for decreased line buffer sizes

4.1 Motivation

The limitation on the number of layers that can be stacked for a given on-chip memory size, results in a trade-off between I/O bandwidth and on-chip memory size. The only features in this on-chip memory are those in the line buffers. Hence, to further optimize this fundamental trade-off, the length of the line buffers should be decreased. We propose a tiling approach on top of the baseline depth-first approach to achieve this.

4.2 Base depth-first tiling setup

In order to decrease the size of the line buffers, our approach cuts all feature maps of a stack in 'tiling factor' TF tiles along the dimension of the line buffers. If $\min(W,H){=}W$, there are TF vertical feature map tiles. If $\min(W,H){=}H$, there are TF horizontal feature map tiles. An example is illustrated by stack 2 ⑥ in Fig. 3, where $H > W$ and the feature maps are thus split into $TF = 2$ vertical tiles ⑦. No two tiles of the same feature map are ever needed simultaneously.

With TF the number of tiles the stack is divided into, the tiling method decreases the amount of required on-chip memory per network layer in the stack to

$$N_P = (k-1) \times \min(W,H) \div TF + (k-1) + O, \tag{2}$$

with O an overhead explained in subsection 4.4.

4.3 Impossibility of further tiling

Tiling along the other dimensions of the feature maps in an effort to further decrease the required on-chip memory, as is done in traditional tiling approaches, is impossible or useless. Specifically, tiling across the channel dimension is often impossible as all channels are needed to compute and propagate pixels through the network layers. Tiling along the remaining spatial dimension, e.g. vertically in Fig. 3 as $\min(W,H){=}W$, on the other hand has no effect on the size of the line buffers at all. It is therefore useless within the depth-first approach. In an alternative view, one could also regard the described depth-first approach itself as maximally tiling this spatial dimension.

4.4 Handling tile boundaries

Special care is required at the boundaries between tiles Specifically, with $k \times k, k > 1$ patches, the t^{th} tile in feature map FM_i, $T_{i,t}$, necessary for computing $T_{i+1,t}$ in FM_{i+1}, should be $(k-1)/2$ pixels wider than $T_{i+1,t}$ on each side.

To provide the extra pixels on the right side, the right boundaries of all tiles of FM_i are $(k-1)/2$ pixels more to the right than the corresponding boundaries in FM_{i+1}, as exemplified by ⑧ vs. ⑦ in Fig. 3. However, this also makes the line buffers in the leftmost tile $T_{i,1}$ of any FM_i longer than those in $T_{i+1,1}$ of FM_{i+1}. The O term in (2) represents this overhead and sections 6 and 7 take it into account.

FM_i's extra needed pixels on the left side of a tile $T_{i+1,t}$ were already calculated for $T_{i+1,t-1}$. Hence, there are three options to provide these pixels for the calculation of $T_{i+1,t}$: they can be recalculated for $T_{i+1,t}$, stored on-chip like in [11], or stored off-chip. However, recalculation of these pixels similarly requires recalculation of their inputs in FM_{i-1}, etc. The region of pixels that have to be recalculated widens with $(k-1)/2$ pixels for each layer in the stack, which superlinearly increases the total amount of pixels to be recalculated with the number of layers in a stack. Because each pixel can require a large amount of operations[2], we choose not to use such a recalculation method, like [11]. Caching these pixels in on-chip memory is not an option either: the purpose of the tiling method is to decrease the amount of pixels cached on-chip.

We therefore propose to use the third option: offloading these pixels at the left of $T_{i+1,t}$ to external memory upon their computation for $T_{i+1,t-1}$⑨. They are then refetched for computing $T_{i+1,t}$. This avoids their recomputation, yet at the expense of an increased external I/O bandwidth. The amount of overlapping pixels N_{OlP} to store externally (note: not at the same time) then equals

$$N_{OlP} = (TF-1) \cdot \max(W,H) \cdot \max(0, k-S), \tag{3}$$

with TF from (2), k the layer's patch size and S its stride, W the feature map's width and H its height. The max-function gives the number of pixels alongside a tile boundary.

Although (3) indicates that tiling creates an overhead on the I/O bandwidth, it can still allow a reduction of the overall I/O bandwidth. Specifically, because the necessary line buffers decrease in size due to tiling, it allows more layers to be stacked within the same on-chip memory constraint. This prevents complete feature maps to be passed to and from external memory. As will be shown in Section 6, when optimized properly, this trade-off favors tiling for high

[2] e.g. each C=64 channels large pixel calculated with a classic convolutional kernel from a 3×3 pixels large input patch with C_{in}=64 channels requires $3 \times 3 \times 64 \times 64 = 36864$ multiply-accumulates

resolutions, as its bandwidth cost grows with $\max(W, H)$ only, whereas the cost of transferring feature maps between layers grows with $W \cdot H$.

5 Optimally combining stacking and tiling

Previous sections indicated that both stacking and tiling allow to influence the trade-off between on-chip cache capacity and external I/O bandwidth inherent to the depth-first approach. This results in a multidimensional optimization problem, where on-chip cache size and required external I/O bandwidth are optimized by tuning the grouping of network layers into a sequence of depth-first stacks, together with the degree of tiling used in every stack.

In this work, this optimization is approached by first manually choosing a set of potential candidate cut positions for splitting the network into stacks, and then exhaustively testing all combinations of these candidate cut positions. Ideally, the candidate stack cut positions are set after every layer, so that the optimal solution can not be excluded by this choice. However, the number of possible combinations grows exponentially with the number of candidate positions. Thus, when dealing with very deep networks, some engineering choices should be made to keep the problems complexity manageable. E.g.: it is a good idea not to break short skip or residual connections by putting a stack cut in between, as then not only the feature map at the stack cut needs to be passed through external memory, but also the feature map from the skip or residual connection. Once the candidate cut positions chosen, the necessary cache capacities and external I/O bandwidths for all combination are first calculated assuming no tiling.

Next, for each combination of stack cuts, the tiling factor TF is increased individually for every stack individually in order to lower the required on-chip capacity at the cost of a higher external I/O bandwidth. Note that only the stack that uses the most on-chip memory should be considered for tiling, as this stack determines the required on-chip memory capacity. Applying tiling to other stacks would not decrease the necessary memory capacity, but would still result in an extra bandwidth cost. Therefore, our optimization first finds the stack that uses the most on-chip memory and then increases the amount of tiling used in it. These two steps are iterated for each combination of stack cut positions until a chosen maximum tiling factor is reached.

Aside from the feature maps, the CNN model itself should be accounted for as well. In our optimization, the memory cost of the CNN weight kernels can be accounted for in two ways: either the complete CNN model is always in on-chip memory, or only the part of the model of the stack being executed is in on-chip memory. The former option has a higher on-chip memory cost whereas the latter increases the bandwidth as the model needs to be fetched for each inference. Both options are available to the optimizer.

Finally, our optimization takes the Pareto front of all evaluated combinations of stack cut positions, tiling factors, and model storage options.

6 Analysis and results

6.1 Details of analysis setup

The layer-by-layer lower bound defined in subsection 2.3 and the results of our baseline, and tiling-enhanced depth-first approach (sections 3-5) are compared in the external I/O bandwidth vs. on-chip memory capacity trade-off space for the three different CNNs shown in Fig. 4:

- SRGAN [2], a 37-layer super resolution CNN
- The MobileNetV2-SSDLite [14], a feature extraction network for object detection
- DMCNN-VD(3×3) [1], a 20-layer CNN for jointly demosaicing and denoising images.

All networks are deployed on 720p HD (1280×720) and 4k UHD (3840×2160) images. Candidate stack cut positions (see section 5) are set as shown in Fig. 4. Activation and batchnorm layers are considered part of the preceding layer. For the depth-first method proposed in this paper, short skip/residual connections are never broken whereas long skip/residual connections are supposed to be saved in off-chip memory, as indicated in Fig. 4. The depth-first results account for the external I/O bandwidth this requires.

We chose to double the tiling factor whenever it is increased. A linear step could be applied as well, but would only give marginal gains. The maximum tested tiling factor was 64×. The results indicate that little is to be gained with higher tiling factors (see subsection 6.3). The allowed tiling factors thus were $1\times, 2\times, 4\times, ..., 64\times$.

6.2 Depth-first vs. layer-by-layer comparison

Fig. 5 shows the layer-by-layer lower bounds for all three CNNs for both input image resolutions. When more memory is available, more features are stored in on-chip memory, and the external I/O bandwidth is hence smaller. At the right end of all curves, the biggest and thus all feature maps fit, and thus no internal features are ever send to off-chip memory. At that point, the bandwidth is minimal and equal to just the network's input and output size.

For both resolutions, Fig. 5 also shows the results of the optimization from section 5 for the depth-first approach with up to 64× tiling in a stack. Results with no tiling and with a lower maximum tiling factor of 4× are also shown.

A comparison of the layer-by-layer bounds with the depth-first Pareto fronts in Fig. 5 clearly shows the depth-first approach beating the traditional layer-by-layer approaches by orders of magnitude, even under the optimistic best-case assumptions made for the latter. For instance, the minimal bandwidth depth-first solution for DMCNN-VD running at 4k UHD requires 268× less bandwidth than the best-case layer-by-layer solution with the same amount of on-chip memory,

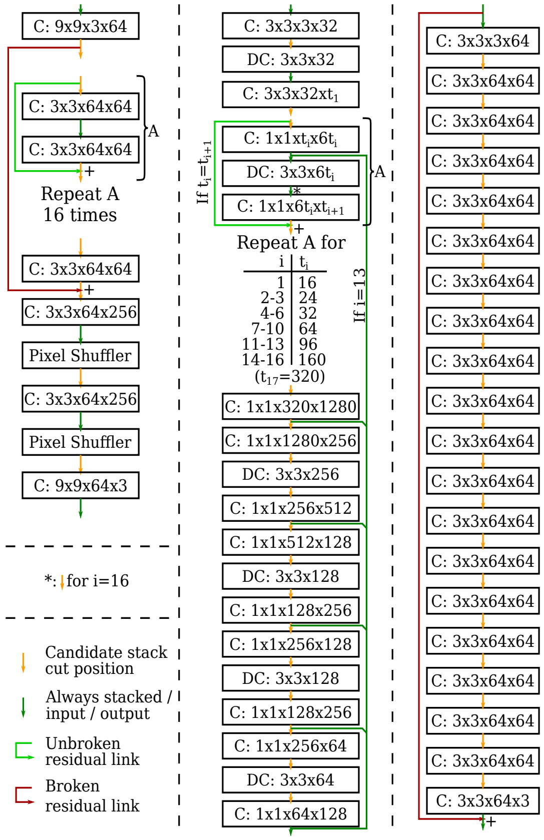
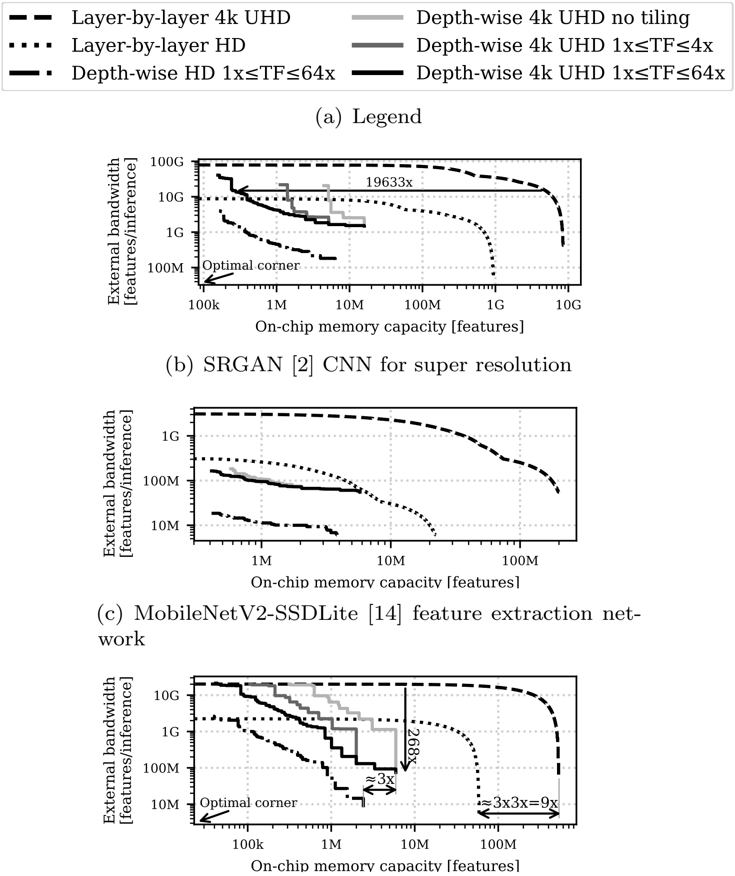


Figure 4: The three CNNs used for evaluation in this paper. From left to right: SRGAN [2], MobileNetV2-SSDLite [14] and DMCNN-VD [1]. Normalization and activation layers and activation layers are not pictured and always stacked. C: Convolutional Layer, DC: Depthwise Convolutional Layer.

as indicated in Fig. 5(d). Vice versa, for the same external I/O bandwidth, the tiled depth-first approach requires up to 19633× less on-chip memory, as indicated in the high bandwidth region of SRGAN running on 4k UHD images. This illustrates the stringent need to go to optimized depth-first processing for CNN deployment on very high resolution images.

A closer look at Fig. 5(b) and 5(d) show a larger minimum bandwidth for our depth-first approach than for the layer-by-layer one. However, this difference is only due to the long skip connections in these networks, which are accounted for in the depth-first curves but, according to the unrealistic best-case assumptions explained in subsection 2.3, not in the layer-by-layer bounds.

Next, the Pareto fronts for 4k UHD with different maximum tiling amounts show the benefit of our tiling approach (section 4): when some extra I/O bandwidth is allowed, it moves the Pareto front towards smaller on-chip memory



(a) Legend

(b) SRGAN [2] CNN for super resolution

(c) MobileNetV2-SSDLite [14] feature extraction network

(d) DMCNN-VD(3 × 3) [1] demosaicing CNN

Figure 5: Comparison between the layer-by-layer lower bound from subsection 2.3, a baseline depth-first approach without tiling, and our depth-first approach with tiling from section 3. Results for both HD and 4k UHD (=HD×3×3).

capacities. Although the effect is small for MobileNetV2-SSDLite, for which kernels contribute heavily to on-chip memory usage, it can save up to >20× of necessary on-chip memory capacity for equal external I/O bandwidths (and vice versa), as shown by Fig. 5(b) and Fig. 5(d).

Finally, Fig. 5 allows to derive some scaling laws in function of input resolution. First, the minimum bandwidth, which consists of just the input and output feature maps of the CNN, scales with the overall input size ($W{\cdot}H$). This is inherent to the application and hence unavoidable. However, the on-chip memory capacity necessary to reach the minimum bandwidth scales different for both approaches. The layer-by-layer approach caches feature maps, which scale with both spatial dimensions of the input, i.e. with $W{\cdot}H$. The depth-first approach, however, uses line buffers, which only scale with one spatial dimension ($\min(W, H)$). This is most visible in Fig. 5(b) and 5(d), where kernel sizes are more negligible than in 5(c). These different scaling laws show the importance of using depth-first approaches for high resolutions.

6.3 Insights in optimal depth-first stacking and tiling

In order to gain insights in the resulting Pareto optimal combinations of stacking and tiling, Fig. 6 shows the following information for all labeled points on the Pareto optimal depth-first curves for SRGAN in Fig. 6(a) (curves of Fig. 5(b)).

1. The **locations of the stack cuts**, as in Fig. 3 ⑧. In Fig. 6, for each shown Pareto point (horizontal axis and letters), each stack cut is represented by a marker.

2. For each of these stacks, the **amount of tiling** that is used is indicated by the symbol of the marker of 1) corresponding to the 'Tiling' legend in Fig. 6(d).

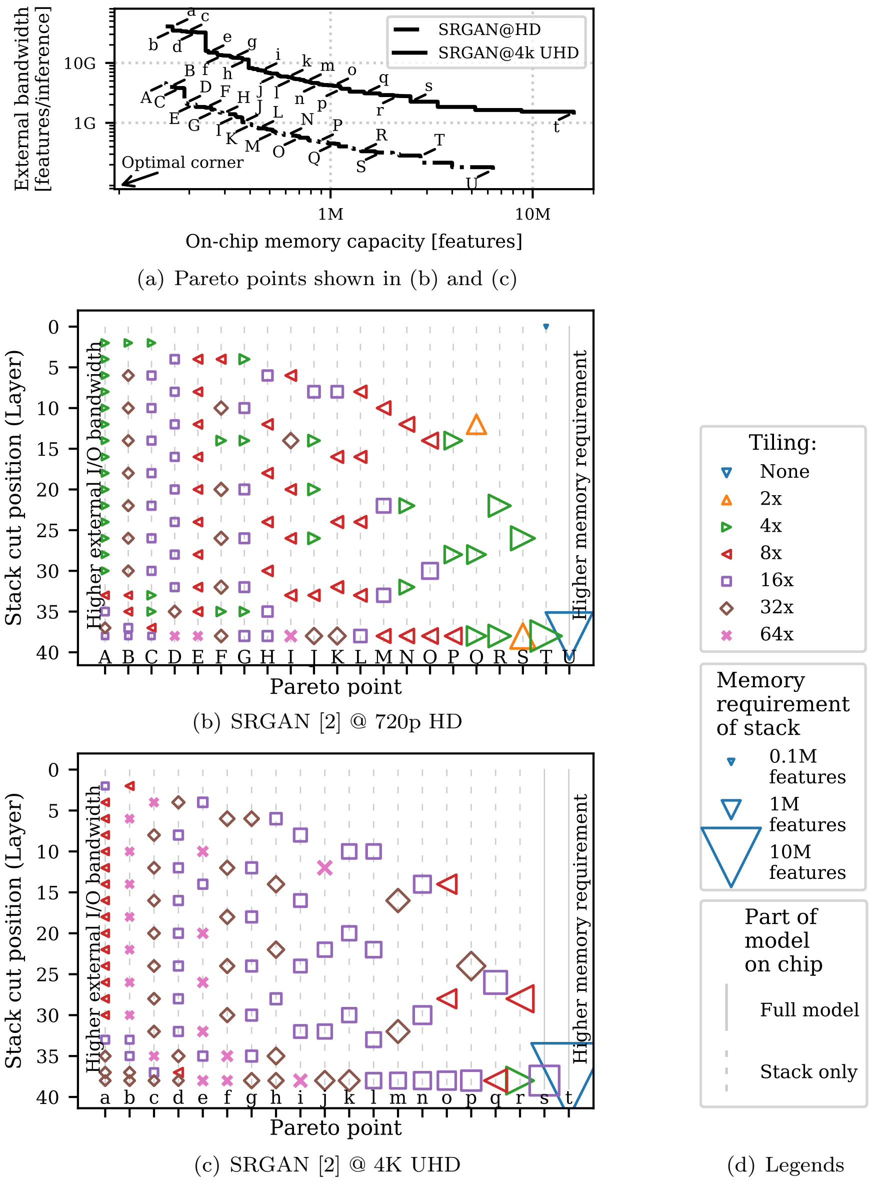
3. For each of these stacks, the **amount of required on-chip memory capacity** for its execution is represented by the size of the marker of 1). Note that, for each Pareto point, the required on-chip memory capacity for inference of the whole network hence corresponds to the size of the largest marker on the vertical gray line belonging to that Pareto point. The 'Memory requirement of stack' legend in Fig. 6(d) relates the marker size to numerical memory requirements.

4. For each shown Pareto optimal point, the absence or presence of dashes in the vertical gray lines indicate **if the CNNs whole model is always in on-chip memory**, or only the part necessary for executing the current stack. The corresponding legend in Fig. 6(d) is titled 'Part of model on-chip.'

With this information in Fig. 6(b) and 6(c), the following observations can be made.

By comparing the required on-chip memory capacities for all stacks of a single Pareto point (on a single vertical gray line) with 3), it is clear that each stack of a single Pareto optimal solution requires approximately the same on-chip memory capacity. If not, the stack with the lesser on-chip memory requirement could be made deeper by the optimizer – which is generally beneficial for the external I/O bandwidth – without increasing the maximum on-chip memory requirement for inference of the whole network. As can be seen, the optimizer adjusts the depth and tiling factor of each individual stack to achieve comparable on-chip memory requirements for each stack of a particular Pareto-point.

This balance can be achieved both by (re)placing stack cuts and by changing the tiling factor TF in the stacks. In Fig. 6(c), Pareto points 'a' and 'b' give an example: in 'a', the stacks at the end of the CNN compensate for working on larger feature maps (compared to stacks more to the front) by using more tiling.



(a) Pareto points shown in (b) and (c)

(b) SRGAN [2] @ 720p HD

(c) SRGAN [2] @ 4K UHD

(d) Legends

Figure 6: For the points labeled in (a), the positions of the stack cuts in the network (marker present; vertical axis), the amount of tiling used in each stack of each point (‘Tiling’ legend), the required on-chip memory capacity during execution of each stack of each point (‘Memory requirement of stack’ legend), and whether the CNNs whole model is always on-chip, or just the part for the currently executing stack (‘Part of model on-chip’ legend).

In ‘b’ however, the stacks more to the front are made deeper and, to restore the balance, use higher tiling factors. In both cases however, all stacks are balanced in the sense that they use approximately the same amount of memory.

1) and the ordering of the Pareto points also show that Pareto optimal solutions with smaller on-chip memories require more stack cuts, whereas low bandwidth solutions require few. This is expected from the trade-offs given in sections 3 and 4.

Using 2), a comparison of Fig. 6(b) and 6(c) shows that optimal solutions for higher resolutions generally use more tiling. Also, comparing the Pareto points with the lowest memory requirements between Fig. 6(b) and 6(c), i.e. points ‘A’ and ’a’, with 3) shows that both need approximately the same minimal capacity of on-chip memory. This can be explained by the fact that both solutions require little more memory than the largest weight kernel (about 148k parameters), which can be found in the second to last convolutional layer. In other words, the tiling is so aggressive that the feature maps no longer dominate the required memory capacity shared by the line buffers and model weight kernels. Note that both use 64× tiling in the final stacks. This also indicates that our set maximum of 64× tiling did not restrict the optimization much.

Only the Pareto points with the lowest bandwidths keep the CNNs whole model on-chip. Note that the minimal bandwidth Pareto optimal point ‘U’ for HD requires a bandwidth of 165M features/inference and 6.4M on-chip memory, of which 24% is used for keeping SRGAN’s whole model (1.5M) on-chip. As loading the model from off-chip memory once per inference causes only a relatively low bandwidth overhead, the optimization results show that this is the best option when a small increase in external I/O bandwidth is allowed.

7 State-of-the-art discussion and comparison

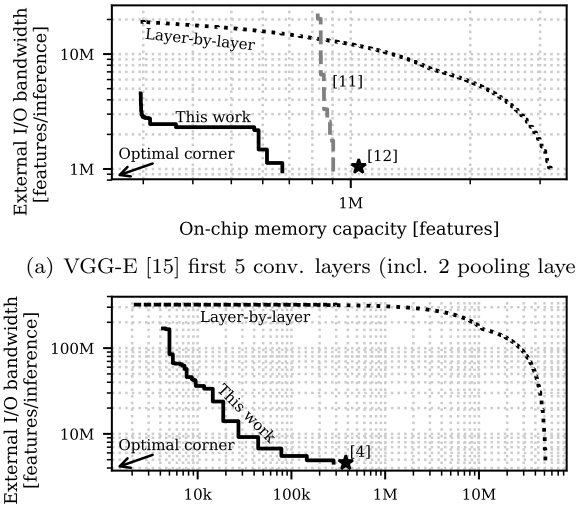
Depth-first approaches to CNN inference have recently been introduced into the state-of-the-art, sometimes calling it ‘layer-fusion’. We hereby highlight the most important differences with the work presented in this paper, summarized in Table 1. Ref. [13] presents a software framework for layer fusion on CPU/GPU. Yet, the required recalculations of intermediate features in the presented approach prevent improvements on convolutional layers. These are hard to avoid in the GPU parallelism model because fine grain synchronization between threads is needed. However, to allow deeper stack, which save more bandwidth, it is necessary to support convolutional layers.

The FPGA-focused work of [11] introduces the idea of depth-first processing for efficient FPGA mapping of deep neural networks. However, [11] relies on two-dimensional region-of-influence pyramids. Such region-of-influence grows going from the last to first layer in a stack. We argue that only the region-of-influence from one layer to the immediately preceding layer matters, as it can capture the influence of all layers before that. Our work exploits this with the scanning line buffers approach.

Similar to our work, [12] uses line buffers. However, [12] uses coarser gran-

Table 1: Related CNN state-of-the-art overview

	[4–9]	[13]	[3] & [11]	[12]	This work
HW/SW	HW	SW	HW	HW	HW
Depth-first		✓	✓	✓	✓
Line buffers				✓	✓
Tiling	✓				✓



(a) VGG-E [15] first 5 conv. layers (incl. 2 pooling layers)

(b) QFSRCNN [3]

Figure 7: This work compared to [11] and [12] in 7(a) and [3] in 7(b)

ularity computations, processing layers line-by-line instead of pixel-by-pixel. The former needs larger buffers, the latter could suffer from low utilization of multiplier units due to limited parallelism. This is a trade-off which should be optimized further for practical implementations. However note that aside from one spatial dimension, parallelism for higher throughput can still be achieved over the kernel dimensions (2 spatials k, input channels C_{in}, output channels (filters) C).

Ref. [12] further presents a dynamic programming algorithm to decide the stack cut positions. This is useful for deeper networks, but for the CNNs in this paper, our exhaustive optimization was still feasible. Most importantly, our work improves upon [12] with the tiling approach from subsection 4. As shown in section 6, this approach can improve upon line buffer approaches without tiling by another order of magnitude. Like [11], [12] also misses a broader comparison between depth-first and layer-by-layer approaches.

As [11] and [12] benchmark their results on the first 5 convolutional (and 2 pooling) layers from VGG-E [15] (operating on relatively small 224×224 pixels large inputs), Fig. 7(a) compares the work in this paper to those on the same benchmark[3]. Fig. 7(a) shows that our method requires up to ≈3× less on-chip memory for the same external I/O bandwidth compared to [11].

Fig. 7(b) compares our work with [3], a super resolution solution without off-chip memory, streaming input directly from a camera and output directly to a monitor. As this is still external I/O bandwidth, a comparison remains valid. For fairness, we also used line buffers along the (longer) horizontal dimension of the input in this case, because this is how the imager data is streamed in in [3].

In the non-CNN image processing field, depth-first-like processing making use of line buffers has been more widespread [16–21]. Ref. [22] is a state-of-the-art work based on 'Halide' [23], a software language that separates an algorithm and its scheduling. Both [23] and [22] do not deploy this towards CNN processing, yet [10] introduces Halide-based CNN scheduling and hardware optimization. However, [10] did not include the possibility for depth-first execution of CNNs, as made clear in for instance Algorithm 1 in their work, which misses an outer loop iterating over all layers. We consider extending this work with such an outer loop, which effectively allows a depth-first approach as presented here, is interesting future research.

8 Conclusion

Traditional layer-by-layer approaches are fundamentally bounded in the external I/O bandwidth vs. on-chip memory capacity trade-off space, preventing deployment on high resolution images. However, depth-first approaches gain up to two orders of magnitude of bandwidth in this space over the most optimistic layer-by-layer lower bound at equal on-chip memory capacity. Reversely, up to four orders of magnitude of on-chip memory capacity can be saved for the same external I/O bandwidth limitation. Depth-first approaches thus greatly reduce the external I/O bandwidth of CNN accelerators and the energy or throughput costs that come with it. The approaches diverge further for increased resolution images. Specifically, for layer-by-layer processing both bandwidth as well as on-chip memory capacity scale with both width and height of the image, rendering high resolution processing infeasible in embedded platforms. The line buffer-based depth-first processing introduced in this paper, in contrary, benefits from an only linear scaling of required on-chip memory capacity with only the image's width (or height). The here introduced enhanced depth-first with tiling approach can further gain up to an order of magnitude of on-chip memory capacity for a given bandwidth (or vice versa) over an untiled depth-first approach.

[3] Results extracted using Fig. 7 from [11] and Table 1 from [12]

9 Acknowledgment

This work has been supported by the FWO SBO project OmniDrone under agreement S003817N, and the EU ERC project Re-SENSE under agreement ERC-2016-STG-715037.

References

[1] N.-S. Syu, Y.-S. Chen, and Y.-Y. Chuang, "Learning deep convolutional networks for demosaicing," *arXiv preprint arXiv:1802.03769*, 2018.

[2] C. Ledig, L. Theis *et al.*, "Photo-realistic single image super-resolution using a generative adversarial network." in *CVPR*, vol. 2, no. 3, 2017, p. 4.

[3] J. Chang, , K.-W. Kang, and S. Kang, "An energy-efficient fpga-based deconvolutional neural networks accelerator for single image super-resolution." [Online]. Available: http://arxiv.org/abs/1801.05997

[4] M. Peemen, A. A. Setio, B. Mesman, H. Corporaal *et al.*, "Memory-centric accelerator design for convolutional neural networks." in *ICCD*, vol. 2013, 2013, pp. 13–19.

[5] Y. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan 2017.

[6] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss v2: A flexible and high-performance accelerator for emerging deep neural networks," *arXiv preprint arXiv:1807.07928*, 2018.

[7] Y. Shen, M. Ferdman, and P. Milder, "Escher: A cnn accelerator with flexible buffering to minimize off-chip transfer," in *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on.* IEEE, 2017, pp. 93–100.

[8] A. Parashar, M. Rhu *et al.*, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2. ACM, 2017, pp. 27–40.

[9] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on.* IEEE, 2017, pp. 535–547.

[10] X. Yang, M. Gao *et al.*, "Dnn dataflow choice is overrated," *arXiv preprint arXiv:1809.04070*, 2018.

[11] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE Press, 2016, p. 22.

[12] Q. Xiao, Y. Liang, L. Lu, S. Yan, and Y.-W. Tai, "Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on fpgas," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 62.

[13] N. Weber, F. Schmidt, M. Niepert, and F. Huici, "Brainslug: Transparent acceleration of deep learning through depth-first parallelism," *arXiv preprint arXiv:1804.08378*, 2018.

[14] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[15] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: http://arxiv.org/abs/1409.1556

[16] J. Hegarty, J. Brunhaver *et al.*, "Darkroom: compiling high-level image processing code into hardware pipelines." *ACM Trans. Graph.*, vol. 33, no. 4, pp. 144–1, 2014.

[17] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "Polymage: Automatic optimization for image processing pipelines," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1. ACM, 2015, pp. 429–443.

[18] M. A. Özkan, O. Reiche, F. Hannig, and J. Teich, "Fpga-based accelerator design from a domain-specific language," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016, pp. 1–9.

[19] D. Koeplinger, M. Feldman *et al.*, "Spatial: a language and compiler for application accelerators," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2018, pp. 296–311.

[20] J. Hegarty, R. Daly, Z. DeVito, J. Ragan-Kelley, M. Horowitz, and P. Hanrahan, "Rigel: Flexible multi-rate image processing hardware," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 4, p. 85, 2016.

[21] S. Smets, T. Goedemé, A. Mitaal, and M. Verhelst, "978gops/w flexible streaming processor for real-time image processing applications in 22nm fdsoi," in *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, 2019.

[22] J. Pu, S. Bell *et al.*, "Programming heterogeneous systems from an image processing dsl," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 3, p. 26, 2017.

[23] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines,” *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 519–530, 2013.