Streaming Dilated Convolution Engine

Dionysios Filippas, Chrysostomos Nicopoulos and Giorgos Dimitrakopoulos

***Abstract*—Convolution is one of the most critical operations in various application domains and its computation should combine high performance with energy efficiency. This requirement is critical both for standard convolution and for its other spatial variants, such as dilated, strided, or transposed convolutions. In this work, we focus on the design of a streaming convolution engine, called LazyDCstream, that is tuned for dilated convolution. LazyDCstream utilizes a sliding-window architecture for input data reuse and leverages the already-known decomposition of dilated convolution to: (a) maximize window buffer sharing, and (b) enable "lazy" data movement that keeps data transfers per clock cycle as few as possible, and, most importantly, *independent of the dilation rate*. These two architectural features reduce the power consumption relative to efficient streaming convolution engines without introducing any throughput or area penalty.**

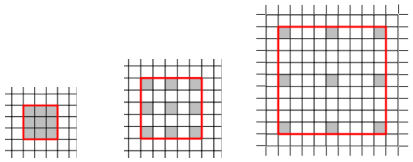
***Index Terms*—spatial filtering, dilated convolution, convolutional neural network, low power design**

I. Introduction

The quality of deep learning has increased significantly with Convolutional Neural Networks (CNNs) [1]. CNNs consist of multiple convolutional layers that convolve the input feature maps with multiple filters to produce a new set of output feature maps [2], [3].

Standard 2D convolution assumes that each filter slides across the pixels of an input to produce a filtered output. Several other spatial convolution variants are also possible [3]. Strided convolution assumes that the filter is applied at a stride of k elements, which reduces the output resolution by the same factor k. Transposed convolution is the inverse of strided convolution and increases output resolution. In an effort to increase the receptive field [4], [5], the filter's coefficients are spaced out R elements apart in dilated convolution. Two examples of an inflated kernel are shown in Fig. 1. For $R = 1$, dilated convolution is the same as standard convolution.

The widespread proliferation and adoption of CNNs has triggered the need to accelerate them directly in hardware. CNN accelerators are built using mainly two architectural templates [6]. In the first approach, shown in Fig. 2(a), CNN layers are executed on systolic array accelerators [7], after transforming convolutions to matrix multiplications [2]. The alternative setup, shown in Fig. 2(b), involves spatial pipeline dataflow architectures, such as those in [8], [9], [10], which consist of multiple streaming engines with each one implementing a specific CNN layer using locally-stored weights. Data transfers between the engines occur in a pipelined manner through convolution-specific buffer architectures that employ line and window buffers to facilitate data and memory reuse [11], [10], [12]. The streaming nature of data transfers allows for the use of much simpler Direct Memory Access (DMA) engines that are only tasked with rudimentary access patterns. The addition of support for unconventional convolutions would require the handling of irregular memory accesses [13], or rearrangement of the flow of data inside the systolic array [14], [15].



(a) $R = 1$ (b) $R = 2$ (c) $R = 4$

Fig. 1. The kernel in a dilated convolution for varying dilation rates.

In this work, we focus on the design of a power-efficient dilated convolution engine that is part of a spatial dataflow pipeline accelerator (see Fig. 2(b)). Our goal is to optimize the data movement to reduce power consumption without obstructing the uninterrupted pipelined flow of data across consecutive engines, and without relying on any data re-organization, or any irregular data fetching from a DMA engine. The proposed design, called LazyDCstream, does not alter the architecture of the streaming engines themselves. Instead, it optimizes their buffer usage when performing dilated convolutions, without restricting any other orthogonal optimizations that may be employed (e.g., datapath unrolling).

LazyDCstream utilizes the decomposition of dilated convolution to multiple non-dilated convolutions [16] to:

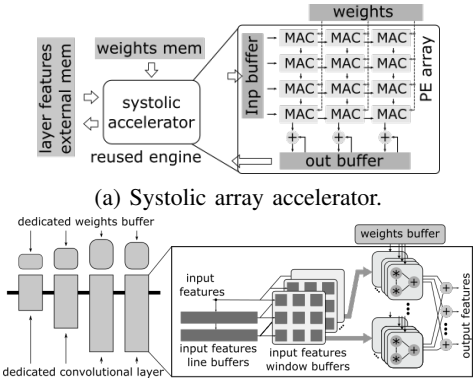
(a) Minimize the amount of buffering needed to support dilated convolutions by appropriate time-sharing of buffers;
(b) Improve clock gating efficiency by limiting data switching activity during the engine's operation to the absolute minimum needed for the algorithm's correctness. In fact, data movements per cycle are equal to the number of kernel coefficients and are *independent of the dilation rate.*

To highlight the effectiveness of LazyDCstream in reducing power consumption, we designed a spatial dataflow accelerator that embeds LazyDCstream in each engine and executes inference on a variant of VGG-16 [17], [18] that includes dilation in all CNN layers. The experimental results demonstrate that LazyDCStream can reduce the power consumption of various CNN layers by 15–39% for ASIC implementations and 2–15% for FPGA implementations, without incurring any throughput or area penalty.

This work was supported by Siemens EDA research grant to Democritus University of Thrace, Greece on "High-Level Synthesis Research for System on Chip".

Dionysios Filippas, and Giorgos Dimitrakopoulos are with the Department of Electrical and Computer Engineering, Democritus University of Thrace, Xanthi, Greece, (e-mail: dfilippa@ee.duth.gr, dimitrak@ee.duth.gr)

Chrysostomos Nicopoulos is with the Department of Electrical and Computer Engineering at the University of Cyprus, Nicosia, Cyprus (e-mail: nicopoulos@ucy.ac.cy).



(a) Systolic array accelerator.

(b) Spatial pipeline dataflow accelerator.

Fig. 2. The two main architectural templates used for CNN accelerators.

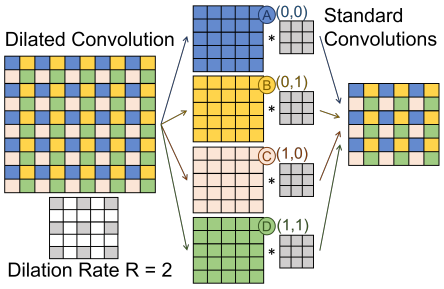


Fig. 3. The decomposition of a dilated convolution with $R = 2$ to multiple independent non-dilated convolutions as proposed in [16]. Each channel works on the non-dilated version of the kernel applied on parts of the original input.

II. Streaming convolution engines for standard and dilated convolutions

The spatial pipelined dataflow accelerators of Fig. 2(b) are composed of a series of streaming convolution engines, with each engine adapted to the characteristics of the implemented CNN layer [6], [8]. To handle incoming data streams, each engine utilizes a sliding-window memory architecture [10], [12] that consists of: 1) the line buffers that store the elements of the active rows of the input feature map, following the sliding-window pattern of the kernel; 2) a window buffer able to store all the currently active features of an input feature map where the corresponding filter will be applied. The size of the kernel determines the number of line buffers and the size of the window buffer. For instance, in standard convolution, a $W \times W$ kernel needs a $W \times W$ window buffer (registers) and $W - 1$ line buffers, which are mapped to SRAM blocks.

Incoming input features are pushed in the corresponding window buffer and in the upper row of line buffers. In the meantime, the window buffer is filled with data coming from the line buffers. This data is also moved downwards in the line buffers, thereby simulating the downward sliding of the filter. To simulate the rightward sliding of the filter over the input, the window buffer shifts its content to the right. Following this data movement, the engine manages to keep the input features aligned with the corresponding filter coefficients needed in each cycle to compute a new output. A new output feature is produced by accumulating the individual results across the multiple input features maps.

The reference dataflow architecture can be customized to various performance requirements [19], [9]. Unrolled architectures using wide enough window and line buffers and multiple MAC operators can store data from multiple input feature maps and compute multiple output features per clock cycle. Other area-efficient architectures allow buffer sharing by producing only a part of the output features.

Computing dilated convolution for a reference streaming convolution engine would require more line buffers and a larger window buffer. In dilated convolution, the filter's size is artificially increased according to the dilation rate. For a $W \times W$ kernel and a dilation rate of R, the streaming convolution engine would need $(W - 1)R$ line buffers and larger window buffers for all input features. Input data would be continuously shifted in and out of the window buffers following the same access pattern. However, in some clock cycles, those pixels would be multiplied with zero coefficients without affecting the output. The rows of the window buffers that contain just the holes of the inflated kernel can be completely removed, thus simplifying the reference architecture.

III. The architecture of LazyDCstream for dilated convolutions

Without loss of generality, to easily elucidate the functionality of LazyDCstream, we will henceforth describe the architecture of a streaming dilated convolution engine that convolves a single input feature map with a single filter.

Dilated convolution can be decomposed to multiple standard convolutions applied on different parts of the input [16]. This decomposition is highlighted in Fig. 3 for a 3×3 kernel applied using a dilation rate of $R = 2$. Each one of the four smaller convolutions (channels) receives a subset of the input and the same non-dilated 3×3 kernel. For the example shown in Fig. 3 the input is broken down into four channels: A, B, C and D. A and B consist of data that are found in the even rows of the input, while C and D consist of data from the odd rows. Equivalently, A and C contain data from the even columns of the input, while B and D contain data from the odd columns.

In the general case, a dilated convolution of rate R is split to R^2 channels. Input (i, j) of the input belongs to channel (k, l) where $k = i \mod R$ and $l = j \mod R$. In the example of Fig. 3, channels A, B, C, and D represent channels $(0, 0)$, $(0, 1)$, $(1, 0)$, and $(1, 1)$, respectively.

A. Decomposed dilated computation on time-shared hardware

Adopting "as is" the decomposition of a dilated convolution [16] means that it can be computed using multiple standard streaming convolution engines each one operating on the original "small" kernel and a subset of the input.

Although such an approach leads to a functionally-correct design, it has a lot of redundancy. The multipliers and adders that compute the convolution operation can be shared across channels. Each arriving data item belongs to one of the four channels of Fig. 3. Therefore, in only one of the channels data will need to move (get shifted in the window buffer and move across line buffers). As a result, in this cycle, only

one channel needs to compute a new output. To share the multipliers and adders, we insert multiplexing logic at the input of the multipliers to pick the correct data depending on which channel is active. This organization is shown in Fig. 4 for a 3×3 kernel and a dilation rate of $R = 2$. Each channel consists of a 3×3 window buffer and two smaller line buffers, since each channel performs a convolution on a subset of the input.

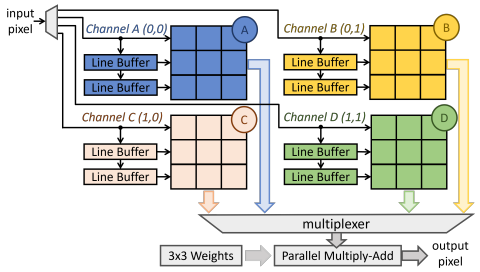


Fig. 4. The decomposition of dilated convolution allows its computation using multiple smaller standard convolution engines that can share a common parallel multiply-add unit.

In our example, the line buffers of channel A contain data that come from the even rows and even columns of the input. In the case of channel B, line buffers store data from the even rows and odd columns. Since data arrive in a row-wise manner, the line buffers of channel A and B will contain data from the same rows but from different columns. This observation allows us to merge the line buffers from the two channels into a bigger double-size line buffer (the size of the merged line buffer is no more than the one used for reference non-dilated convolution). Data are stored in the line buffers in an interleaved manner; data from even columns are stored in even addresses and those from the odd columns are stored in the odd addresses. The same merging can be done for channels C and D as shown in Fig. 5. The demultiplexers are used to align the data transfer from the line buffers to the window buffers based on the channel that the incoming data item belongs.

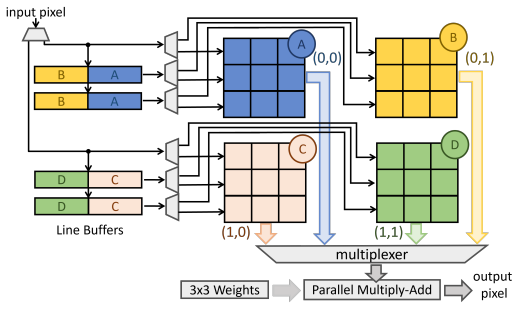


Fig. 5. Line buffers that correspond to channels of the same group of rows can be merged to avoid memory fragmentation. The size of the merged line buffers is equal to the size of a standard convolution engine.

The streaming mode of data arrival allows our architecture to be further optimized. In the running example for $R = 2$, when reading a specific input row, only two of the four channels will be active. For the even-numbered rows channels A and B will be active and C and D will be inactive. The opposite holds for the odd-numbered rows. This means that when a data item arrives from a new row, the channels that correspond to that row will start pushing data into their window and line buffers, while the rest will stay inactive. This means that *only two window buffers in case of* $R = 2$ are necessary to compute dilated convolution. Those two window buffers will be used either by channels A and B or channels C and D depending on the index of the active row.

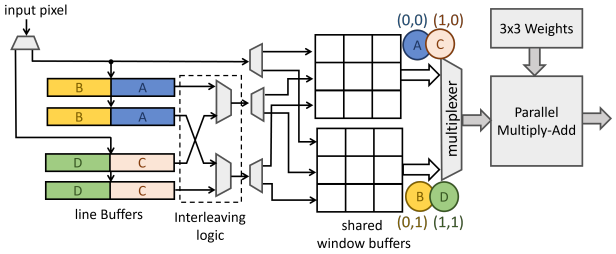


Fig. 6. The organization of LazyDCstream for a 3×3 kernel and a dilation rate $R = 2$ that consists of merged line buffers, time-shared window buffers and a single multiply-add datapath.

To achieve the sharing of the window buffers among all channels we add extra multiplexing logic at the output of line buffers that enables the sharing of the window buffers between the different channels. This is shown in Fig. 6 where the four line buffers are interleaved depending on the row index of the current input. Line buffer to window buffer connectivity has not changed when moving from Fig. 5 to 6. In both cases, the upper line buffers of all channels feed the middle row of the corresponding window buffers, while the lower line buffers feed the last row of the window buffers. Multiplexing logic just selects which line buffers would feed the the corresponding rows of the shared window buffers.

In the general case, we need $(W - 1)R$ line buffers that are active in groups depending on the row index of the input. We need, as well, R time-shared window buffers, each one holding the original small kernel of $W \times W$ input data.

B. Lazy data movement

The channels of the decomposed dilated convolution operate in groups in a mutually exclusive way based on the the row and column indexes of the incoming input. R window buffers are time shared across the R active channels. In each cycle, however, only one window buffer will be active shifting data and computing output data. The other window buffers can be clock gated. At most $W \times W$ data movements will occur in each cycle that correspond to the size of the original (non-inflated) kernel. This amount of data transfers per clock cycle is constant and, most importantly, independent of dilation rate.

Fig. 7 illustrates an example on how data move inside the two time-shared window buffers in the case of 2-dilated convolution for a 3×3 kernel. In cycle t_n, channel A is activated as the new inputs A_7, A_4 and A_1 arrive from the input channel and the line buffers. The window buffer shifts its content to the right. During this time, the third column of the upper window buffer as well as the second and third columns of the lower window buffer have not yet received any input. In fact, the lower window buffer does not receive

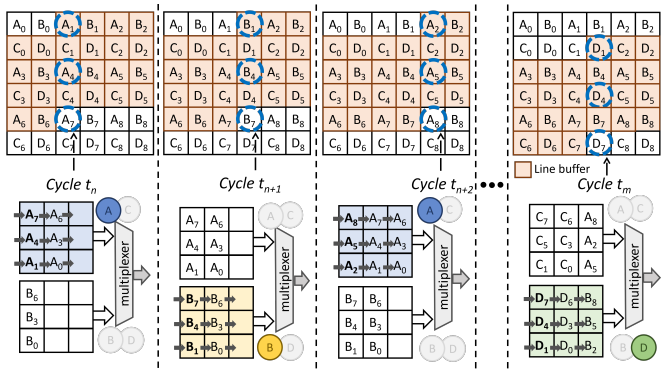


Fig. 7. A snapshot of four cycles (three consecutive cycles and one at a later time) of the operation of LazyDCstream that highlights the reduced data movement and the clock gating of the registers of the inactive windows.

any inputs in this cycle and thus remains clock gated. On the contrary, in cycle t_{n+1}, channel B is activated and receives input $\{B_7, B_4, B_1\}$ from the corresponding line buffers and the input to its first column. The rest of its columns are shifted right, following the operation of standard convolution. In this cycle, the upper window buffer is clock gated. In the next cycle t_{n+2}, channel A is activated again while channel B gets deactivated. The same operation is repeated until the end: the two window buffers are activated in turns leading to nine data elements moving per cycle.

When a new row is being read from the input, as in cycle t_m, the two window buffers are used to accommodate the input from channels C and D. During this time, the outputs have no dependency to input data from channels A and B and therefore the data inside the window buffers are overwritten.

Apart from the window buffer, the amount of data transfers per clock cycle between the line buffers is also constant. On each new input, only one channel is activated by writing also data across line buffers (like shifting data downwards). As shown also Fig. 6, each channel consists of $W-1$ line buffers. Thus, $W-1$ writes will occur in each clock cycle again independent of the dilation rate.

Although LazyDCstream is optimized for dilated convolutions, it can also compute standard convolutions in parallel. For instance, a design that supports a maximum dilation rate of R can utilize the R window buffers available for computing R standard convolutions in parallel.

IV. Experimental results

To highlight the benefits of LazyDCStream, we designed two spatial dataflow accelerators that both execute inference on a variant of VGG-16 [17] that employs dilation in all CNN layers. The first accelerator utilizes LazyDCStream as its streaming convolution engine, while the second one is a state-of-the-art architecture, as employed in [10], [19]. As suggested in [19], even if each CNN layer can employ a separate datapath unroll factor, it is safe to use a uniform unroll factor for all layers. For both designs under comparison, we employed a (4,16) unroll factor for the input and output features, respectively. Therefore, both designs exhibit the same amount of parallelism per layer and, thus, have the same total execution time. They only differ in their *buffering architecture* for dilated convolution layers.

All designs have been implemented in C++ and synthesized to Verilog RTL using Catapult HLS. The buffers in the reference state-of-the-art design – as used in [10], [11] – operate directly on the inflated kernel and utilize the same number of line buffers as LazyDCstream. Using a carefully-designed C++ model for the reference state-of-the-art, hereafter referred to as "Reference," allowed Catapult HLS to keep the same multiply-add units needed for a non-inflated kernel. In this way, the area complexity of both the Reference and LazyDCstream designs is almost the same.

A. ASIC implementation results

Fig. 8 reports the average power consumption of LazyDCStream and the Reference design for each CNN layer of the modified VGG-16 [17]. The modified VGG, similar to the original one, is split in blocks of two or three CNN layers of the same dilation rate. The dilation rate increases exponentially across blocks. Using dilation in all CNN layers (except the first two) enables us to better quantify the expected savings when using LazyDCStream. The numbers for each CNN layer also include the power consumed in padding and in the activation stages that accompany each convolution layer. Pooling and fully-connected layers are not shown, since they consume only a small part of the total power.

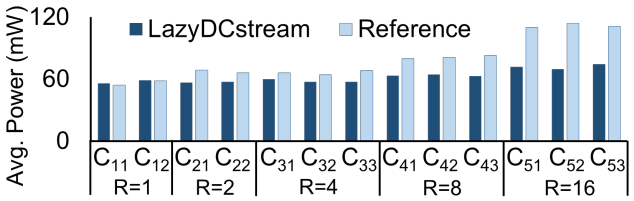


Fig. 8. The average power consumption of the Reference and LazyDCstream architectures for the CNN layers of a modified VGG-16 [17].

Power consumption – both dynamic and leakage – was estimated after logic synthesis using the PowerPro power analysis and optimization tool for a 45 nm standard-cell library. Both designs operate at 500 MHz with 16-bit fixed-point inputs and weights. Switching activity information was gathered after feeding the modified VGG-16 with sample images from ImageNet. Verilog RTL code was derived from C++ using Catapult HLS and synthesized using the Oasys logic synthesis tool. The line buffers are mapped to SRAM macro blocks to minimize the area of the convolution engines.

As can be seen in Fig. 8, LazyDCstream is more power efficient in all layers that involve dilation, mainly due to its reduced data movement and its time-shared window buffers. Power savings range between 15%, for layers with R=2, up to 39%, for the last block with R=16. Since all CNN layers utilize the same datapath unroll factor, they exhibit almost the same power for the same dilation rate. The differences are only due to data switching activity. For the Reference design, the power consumption increases with increasing dilation rate. On the contrary, as expected, LazyDCStream is only marginally affected by the increase in dilation rate.

Even though LazyDCstream requires additional multiplexing logic to implement the time sharing of the window buffers, the area overhead is negligible. The area is dominated by the line buffers and the parallel multiply-and-add units that are the same for both the LazyDCstream and Reference designs.

B. FPGA implementation results

Similar conclusions are drawn when considering the FPGA implementations of the same Reference and LazyDCstream architectures. We implemented each CNN layer separately on the Virtex Ultrascale VCU108 Evaluation Board, targeting a clock frequency of 150 MHz. The results were obtained after mapping the Verilog RTL code produced by Catapult HLS to the FPGA using Xilinx Vivado 2021.1.

The average power consumption numbers – for each layer of the modified VGG-16 – in the two compared designs are shown in Fig. 9. The obtained results clearly indicate that LazyDCstream computes dilated convolution with less power. Depending on the dilation rate of each layer, the power savings range between 2% (for R=2) to 15% (for R=16), while, in both cases, static power consumption contributes around 900 mW to the overall consumption. The key takeaway point is that, unlike the Reference design, LazyDCstream's power consumption is independent of the dilation rate.

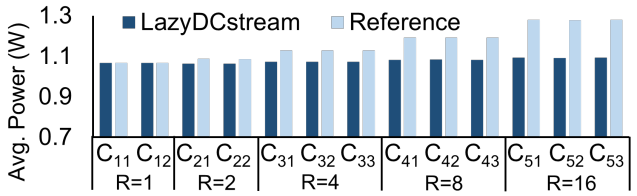


Fig. 9. The power consumption of the Reference and LazyDCstream architectures when implementing the modified VGG-16 layers on the Virtex Ultrascale VCU108 Evaluation board at 150 MHz.

Regarding the area consumption, Table I reports the utilization of FPGA resources for two layers of the modified VGG-16 [17] for the two architectures. Both designs utilize the same number of DSP and BRAM blocks, since they employ the same number of multipliers and adders and the same number of line buffers. In each case, 576 DSP blocks are used to implement the parallel MAC operations of 4×16 unrolled 3×3 filters, while the number of BRAM blocks differs per layer, as the dilation affects the number of line buffers in the design. The time-shared operation in LazyDCstream has two contradictory effects with respect to FPGA utilization: it increases the number of LUTs required to implement the multiplexing logic, while it reduces the amount of registers needed to implement the window buffers.

TABLE I
THE UTILIZATION OF FPGA RESOURCES FOR REFERENCE AND LAZYDCSTREAM FOR TWO LAYERS OF THE MODIFIED VGG-16.

Layer	R	Architecture	LUTs	FFs	DSP	BRAM
C21	2	LazyDCstream	14688	5862	576	96
		Reference	14085	6505	576	96
C41	8	LazyDCstream	15384	7033	576	144
		Reference	14256	8160	576	144

V. CONCLUSIONS

Dilated convolution spreads the kernel's coefficient to a larger window that slides on the input, similar to traditional convolution. In this way, the receptive field of the applied filter is enlarged in a computationally efficient manner. In this work, we take advantage of the "holes" inside the inflated filter of a dilated convolution to perform computation in time-shared streams. These streams operate in groups in a mutually exclusive way, thus requiring only one set of multiply-add units and R window buffers of size equal to the original non-inflated kernel. Most importantly, the data switching activity remains constant per clock cycle and independent of the dilation rate.

REFERENCES

[1] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "A ConvNet for the 2020s," *arXiv:2201.03545*, 2022.

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

[3] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *arXiv:1603.07285*, 2016.

[4] N. Chaudhary *et al.*, "Efficient and Generic 1D Dilated Convolution Layer for Deep Learning," *arXiv:2104.08002*, 2021.

[5] L.-C. Chen *et al.*, "Deeplab: Semantic image segmentation with deep convolutional nets, à trous convolution, and fully connected crfs," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 40, no. 4, pp. 834–848, 2017.

[6] A. M. Deiana *et al.*, "Applications and techniques for fast machine learning in science," *Frontiers in big Data*, vol. 5, 2022.

[7] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Int. Symp. on Comp. Arch. (ISCA)*, 2017, p. 1–12.

[8] M. Blott *et al.*, "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Trans. on Reconfigurable Technology and Systems*, vol. 11, no. 3, pp. 1–23, 2018.

[9] Y. Ma *et al.*, "Optimizing the convolution operation to accelerate deep neural networks on FPGA," *IEEE Trans. on VLSI Systems*, vol. 26, no. 7, pp. 1354–1367, 2018.

[10] P. Whatmough *et al.*, "FixyNN: Energy-efficient real-time mobile computer vision hardware acceleration via transfer learning," *Machine Learning and Systems*, vol. 1, pp. 107–119, 2019.

[11] L. Ioannou, A. Al-Dujaili, and S. A. Fahmy, "High Throughput Spatial Convolution Filters on FPGAs," *IEEE Trans. VLSI Systems*, vol. 28, no. 6, pp. 1392–1402, 2020.

[12] L. Petrica *et al.*, "Memory-efficient dataflow inference for deep CNNs on fpga," in *IEEE Intern. Conf. on Field-Programmable Technology (ICFPT)*, 2020, pp. 48–55.

[13] J.-S. Park *et al.*, "A 6k-MAC feature-map-sparsity-aware neural processing unit in 5nm flagship mobile SoC," in *IEEE Intern. Solid-State Circ. Conf. (ISSCC)*, 2021, pp. 152–154.

[14] D. Im, D. Han, S. Choi, S. Kang, and H.-J. Yoo, "DT-CNN: Dilated and transposed convolution neural network accelerator for real-time image segmentation on mobile devices," in *IEEE Intern. Symp. on Circuits and Systems (ISCAS)*, 2019, pp. 1–5.

[15] K.-W. Chang and T.-S. Chang, "Efficient accelerator for dilated and transposed convolution with decomposition," *arXiv preprint arXiv:2205.02103*, 2022.

[16] Z. Wang and S. Ji, "Smoothed dilated convolutions for improved dense prediction," *Data Mining and Knowledge Discovery*, vol. 35, no. 4, pp. 1470–1496, 2021.

[17] M. A. R. Ratul *et al.*, "Skin lesions classification using deep learning based on dilated convolution," *BioRxiv*, p. 860700, 2020.

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

[19] C. Zhang *et al.*, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Inter. Symp. on FPGAs*, 2015, p. 161–170.