2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP) | 978-1-6654-8308-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/ASAP54787.2022.00027

IMEC: A Memory-Efficient Convolution Algorithm For Quantised Neural Network Accelerators

Eashan Wadhwa, Shashwat Khandelwal & Shanker Shreejith
Department of Electronic and Electrical Engineering, Trinity College Dublin
Dublin, Ireland
Email: {wadhwae, khandels, shankers}@tcd.ie

***Abstract*—Quantised convolution neural networks (QCNNs) on FPGAs have shown tremendous potential for deploying deep learning on resource constrained devices closer to the data source or in embedded applications. An essential building block of (Q)CNNs are the convolutional layers. FPGA implementations use modified versions of convolution kernels to reduce the resource overheads using variations of the sliding kernel algorithm. While these alleviate resource consumption to a certain degree, they still incur considerable (distributed) memory resources, requiring the use of larger FPGA devices with sufficient on-chip memory elements to implement deep QCNNs. In this paper, we present the Inverse Memory Efficient Convolution (IMEC) algorithm, a novel strategy to lower the memory consumption of convolutional layers in QCNNs. IMEC lowers the footprint of intermediate matrix buffers incurred within the convolutional layers and the multiply-accumulate (MAC) operators required at each layer through a series of data organisation and computational optimisations. We evaluate IMEC by integrating it into the BNN-PYNQ framework that can compile high-level QCNN representations to the FPGA bitstream. Our results show that IMEC can optimise memory footprint and the overall resource overhead of the convolutional layers by ∼33% and ∼20% (LUT and FF count) respectively, across multiple quantisation levels (1-bit to 8-bit), while maintaining identical inference accuracy as the state-of-the-art QCNN implementations.**

***Index Terms*—Inference Algorithms, Field Programmable Gate Arrays, Convolution Neural Networks**

I. Introduction

Among the many flavours of neural networks (NNs)[1], Convolutional neural network (CNNs) have proven to be effective in solving computer vision tasks [2]. Recent research demonstrates that CNNs can achieve very high accuracy even for challenging vision problems [3]. Deep CNNs have also shown promising results in non-vision based embedded applications like wireless networks [4], intrusion detection system for automotive CAN networks [5] among many others. Integrating such compute capabilities at or closer to the data source(s) is a key enabler for near-real-time intelligent distributed applications like health monitoring or autonomous transportation systems. However, most state-of-the-art CNNs are compute intensive and thus limit their application at the edge and/or sensing nodes that have limited computing capabilities and energy budgets. While computing requirements can be catered to by using custom architectures on platforms like Field Programmable Gate Arrays (FPGAs), memory requirement to store the intermediate results and the weights of individual layers is a critical factor in the design decision. In many cases, the memory requirement is much higher than the on-chip resources available on low-cost FPGA devices, requiring the use of larger devices that consume more static and dynamic power or offloading these parameters to external storage at the expense of performance cost and energy consumption.

Researchers have attempted to address these challenges by reducing the bit-width of the parameters through quantisation techniques [6] and/or compression schemes [7]. Quantisation techniques rely on the use of a deterministic [6] or a probabilistic formula [8] on the various parameters to reduce computational complexity and memory requirements. Compression schemes like Deep Compression [7] attempt to reduce memory bottlenecks through the use of multi-stage compression pipelines. At the cost of slight reduction in accuracy, these strategies achieve a dramatic reduction in memory footprint and power consumption when deployed on reconfigurable hardware [9][10][11]. Among many others, FINN [11] is a widely used open-source framework for deploying data-driven quantised neural network (QNN) accelerators on FPGAs. The FINN framework can achieve high inference accuracy and throughput at a fraction of the resources required for mapping high-level precision implementations. For instance, a fully binarised CNN model compiled through the FINN framework achieves a throughput of 341 GOPs/s and an inference accuracy of 80.1% for the CIFAR-10 dataset, while only utilising 48% of the logic resources on a Zynq FPGA on the PYNQ-Z1 board [11]. However, over 90% of this resource usage is attributed to the convolutional layers, and 34% of the logic resources are used as memory elements (LUTRAMs). Optimising the memory overhead can consequently have a large impact on the resource requirements and energy consumption of such implementations.

In this paper, we present Inverse Memory Efficient Convolution (IMEC), a scheme to reduce the memory elements incurred by the convolution layer(s) within a QNN architecture through a series of computational and data flow optimisations. Most dataflow neural-net implementations that target FPGAs replicate the convolution operator using a *sliding window* followed by a *matrix-vector* engine to perform element-wise multiply-accumulate (MAC) operations. Compared to traditional generalised matrix-matrix multiplication operator (GEMM) used in CPUs and GPUs to implement convolution operation, this approach reduces the memory required to store in-line computations and thus resource requirements in FPGA implementations. IMEC attempts to reduce the memory incurred further by reordering the dataflow into and out of the matrix-vector engine, resulting in more efficient use of the small buffers incurred in this datapath without affecting the computation itself and thus the accuracy. IMEC can easily be integrated into end-to-end flows like FINN as a convolution header file, allowing designs to be further optimised for memory resources without sacrificing performance, accuracy or throughput. We compare IMEC against the inference designs generated by deep-learning frameworks such as vanilla FINN [11] and LUTNET [9] and show that IMEC can reduce the memory resources incurred by the convolution layers by up to 33% at multiple quantisation levels (1-bit to 8-bit) without any loss in accuracy or throughput.

II. Background

Like all NNs, a CNN is a series of hidden (convolutional) layers between the input and output layers, with inputs, outputs and weights at each layer represented as matrices. A pure convolution layer applies a convolution operation between the inputs and the weight matrices, while pooling layers reduces the spatial size of the input (to this layer), by combining outputs of clusters of neurons into one single value. A non-linear activation function (f_{actv}) is used at the output of convolution layers. By denoting the activation output of layer $l-1$ for layer l with input $\boldsymbol{x}$, we can formulate the general equation for the convolution layer as:

$$\boldsymbol{x}_{i_N,o_w,o_w,o_c} = f_{actv}\left(\sum_{n=0}^{i_N}\sum_{w_1=0}^{k_w}\sum_{w_2=0}^{k_w}\sum_{w_3=0}^{i_c} K_{w_1,w_2}\boldsymbol{x}_{n,w_1,w_2,w_3}\right) \tag{1}$$

The individual notations are described in table I.

Implementing CNNs on FPGAs at full (floating point) precision is not efficient since the resources and primitives are not optimised to deal with 32-bit floating-point computations, while the on-chip memory is insufficient to store the entire weight vectors [12]. Most approaches to FPGA implementation of CNNs utilise some form of hardware tiling [13] and/or software encoding [7] to operate at a reduced precision [6], [14]. We look into these different approaches in the following subsections.

A. Quantised Neural Networks

Early experiments exploring quantisation resulted in a significant reduction in accuracy of CNN inference, with performance issues largely related to the inefficient training techniques [15]. Integrating Batch Normalisation within the training framework addressed this issue, largely improving the accuracy of quantised networks [16]. Mapping high-level CNN models directly to FPGAs will incur floating-point weights/activations and multiplication operations to implement the convolution layers, which are expensive both in terms of resource and energy consumption. QNNs quantise their parameters using a transform while maintaining comparable prediction accuracy [6]. This representation allows the multiplication operations to be replaced by simpler operations such as bitcount, significantly reducing the computational complexity. Recent research explores low-level optimisations like pruning [17] and sparsity [18] to reduce the resource footprint further.

TABLE I: Notations used in this paper

Notation	Expression	Definition
$i_N \times i_w \times i_w \times i_c$	Input Image, I	number of images $\times$ width of image $\times$ width of image $\times$ channels of image
$o_N \times o_w \times o_w \times o_c$	Output Image, O	number of images $\times$ width of image $\times$ width of image $\times$ channels of image
$k_w \times k_w$	Weight Kernel, K	width of kernel $\times$ width of kernel

Once trained, QNNs can be deployed in hardware on FPGAs or ASIC to perform inference or other tasks. In case of FPGAs, frameworks like LUTNET [9], FINN [11] and Dnnweaver [19] can compile high-level (Q)NN representations to the target hardware bitstream. These frameworks apply algorithmic (such as compressing [20] or pruning [21] parameters) and/or low-level optimisations (such as pop-count compressors [10] and tiling [11]) during the implementation phase to utilise the hardware more efficiently [22]. Furthermore, the user can control the unroll factor to achieve high-throughput (multiple PE) or low-resource (single PE) implementations of the QNN architecture. A data-streaming structure uses a pipeline of hardware blocks, each representing a layer of the CNN [9], [11], [23] while a single computation engine approach uses systolic arrays to operate on CNN layers as one monolithic matrix [19], [24], [25]. In this work, we focus on the data-streaming flavour of IMEC for high-performance CNN accelerators; however, IMEC can also be utilised in a monolithic flow to achieve similar benefits.

B. Notations used in this work

Table I describes the notation used in this work to denote three-dimensional matrices. We use small and capital letters for representing line buffers and matrices respectively. Row major order is used for all matrix calculations. Two-dimensional matrices utilise a similar pattern eliminating the number of images and channels information. Thus, a two-dimensional output matrix O will be of size $o_w \times o_w$ instead of $o_N \times o_w \times o_w \times o_c$. The output O of a convolution layer with input I and kernel K will have dimensions given by:

$$o_w = \frac{i_w - k_w}{s_w} + 1 \tag{2}$$

Here, s_w represents the stride used by the convolution layer. In the interest of simplification, we consider the case of a single channel (i.e., $o_c = 1$) when explaining the operation of IMEC and *im2col* in the subsequent sections. Similarly, we use an 8-bit notation for all operands while describing the algorithm in the figures to represent integer values (within 0–255), making it easier to comprehend the operations; the hardware design can cater to multiple (uniform or mixed) quantisation levels. Also, since the number of images before and after convolution does not change, we have inferred $o_N = i_N$ for all cases. We evaluate the impact of the IMEC approach at different quantisation levels in section V.

C. Convolution Techniques

As discussed above, the convolution operation within a layer can be represented by the equation (1). Solving this equation using conventional General Matrix Multiplication (GEMM) operations incurs a computational complexity in the order of $O(o_w o_w k_w k_w)$. Directly implementing such multi-kernel and multi-channel convolutions on an FPGA through GEMM operations will lead to high memory overheads to store intermediate matrices and reduced computational performance [26], unless explicitly parallelised through unrolling/vectorising the multiple nested loops. While simple unrolling of parts of this equation is possible, it incurs significant memory (and computational) resource overheads and causes the performance to be memory bound.

The *im2col* algorithm addresses this challenge by converting the multi-kernel multi-channel convolutions into a collection of GEMM operations [27]. The standard im2col method has memory overheads of $o_w o_w k_w k_w i_c$, which is larger than the dimension of the input matrix ($i_w \times i_w \times i_c$) [28]. As the kernel weight matrices grows, (i.e., larger K and smaller s_w), the memory requirements scale non-linearly, making it impossible to fit into on-chip resources [29], [30]. Memory Efficient Convolution (MEC) algorithm [29] addresses this challenge (size of im2col) by utilising wide buffers within Graphics Processing Units (GPUs) to reduce the memory size from $o_w o_w k_w k_w i_c$ to $o_w i_w k_w i_c$. This allows the multiple processing cores on the GPUs to perform parallel GEMM operations on overlapping portions of these smaller matrices. However, deploying such long buffers (of size $o_w i_w k_w i_c$) on FPGA implementations is inefficient and incurs large resource overhead with limited performance benefits. State-of-the-art FPGA QNN frameworks (including LUTNET, FINN and Dnnweaver) utilise a dataflow based approach through the use of sliding windows on the inputs, reducing the intermediate buffer requirements when compiling high-level QNN representations onto FPGAs. We refer to this approach as *vanilla dataflow convolver* in this paper. The algorithm first interleaves each channel of the input matrix into the memory system, shown in fig. 1. It is then streamed as hardware lanes to a sliding window unit (matrix $L_{vanilla}$) which then banks them into buffers of optimised lengths and are then fed into a matrix-vector unit to perform MAC operations. The resulting outputs create each element of the output matrix O. While the sliding kernel implementation allows computational performance to be optimised, the memory overheads could be further reduced through dataflow optimisations. The IMEC algorithm presented in this paper aims to reduce the size of intermediate buffers incurred by the *vanilla dataflow convolver* approach by decomposing the input matrix into smaller segments allowing multiple compute units to operate on these buffers more effectively with minimal impact on latency.

III. IMEC Architecture

A. Algorithm Overview

Alg. 1 is a pseudocode representation of the proposed IMEC algorithm with Fig. 2 describing the steps involved through a numerical example of two matrices with dimensions $i_w = 7$, $k_w = 3$ and $i_c = 1$. The algorithm operates in two stages: for each iteration, the first stage populates the matrix buffer L_{IMEC}, represented by line 5 of the pseudocode, while the subsequent stage performs the MAC operation, represented by line 6 of the pseudocode. The operation is illustrated in the Fig. 2 with $o_w = 5$ for the given input dimensions (computed using (2)). Once the first line of the buffer is filled, the window shifts by s_w (= 1 in this case) and proceeds to fill up the subsequent line of L_{IMEC} (shown as blue in Fig. 2). The operations are repeated until the first line of the matrix I is fully loaded completing the first stage of IMEC. The second stage performs in-place column-wise element multiplication between the loaded lines of the input matrix and the corresponding weights, which are accumulated at L_{IMEC} to compute the first row of matrix O. The two stages is further repeated across the height of I matrix o_w times ($i_w - k_w + 1 = o_w$ times) to complete the convolution.

B. Extending IMEC for channels

The pseudocode described in alg. 1 can be extended to enable a multi-input (i_N) and a multi-channel (i_c) convolution operation respectively. We utilise a similar dataflow model used by the vanilla dataflow convolution engine in FINN, with modifications to enable reduction in memory elements inferred. We refer to this as *IMEC dataflow convolver*.

The input matrix I is segmented into smaller data stream buffers of length $o_w k_w$ that are packed together as interleaved channels of matrix I. The operation is then similar to an unrolled channel-wise application of the standard IMEC algorithm. The pseudocode in alg. 2 details the steps involved in the streaming structure. Since each image matrix is processed sequentially through this pipelined architecture, a large i_N does not create any additional constraint in this structure.

IV. Implementing in hardware

We integrated the IMEC dataflow algorithm as a convolution layer header file similar to the vanilla dataflow implementation

Algorithm 1 Standard IMEC Algorithm with $s_w = 1$ and $\boldsymbol{L} \equiv L_{IMEC}$

```
1: for $m \in 0 : o_w$ do
2:     for $n \in 1 : k_w$ do
3:         for $p \in 0 : k_w$ do
4:             for $q \in 0 : o_w$ do
5:                 $\boldsymbol{L_{IMEC}}[(n*k_w)+p,q] = \mathbf{I}[m+n,p+q]$
6:                 $\mathbf{O}[m,n] \mathrel{+}= \boldsymbol{L}[(n*k_w)+p,q] \odot K[(n*k_w)+p,1]$
```

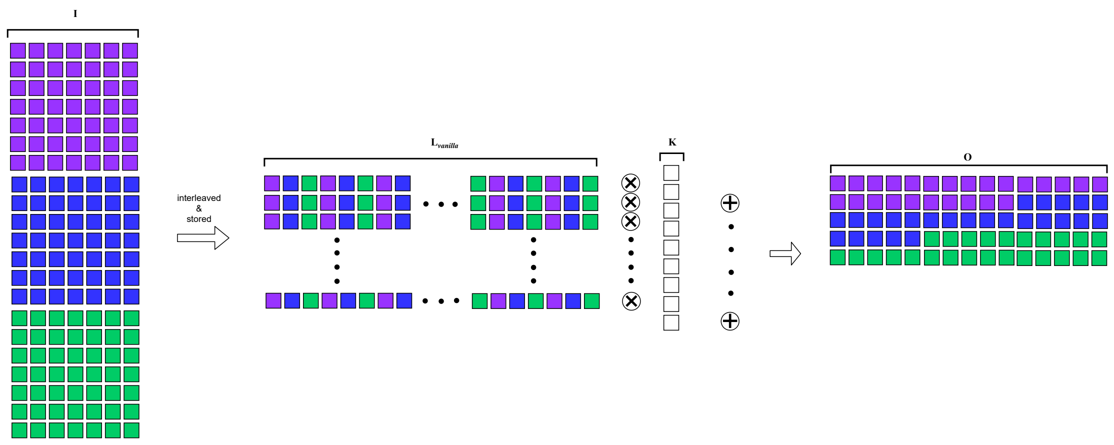

Fig. 1: Vanilla dataflow convolver used by state-of-the-art neural-network accelerators for a three-channelled input matrix

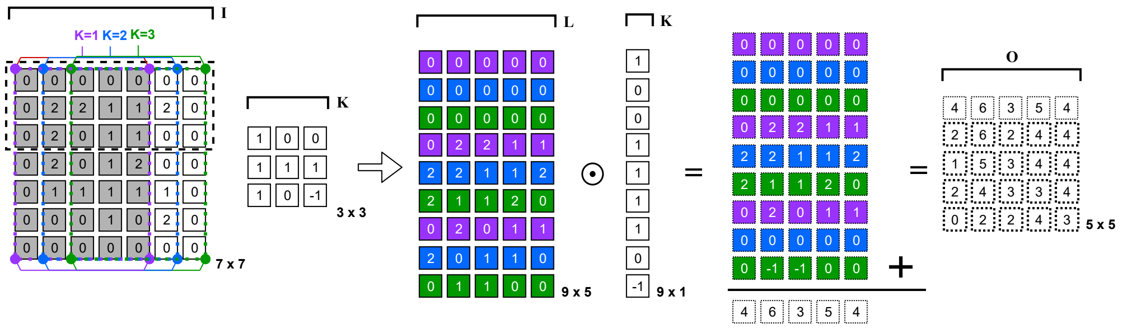

Fig. 2: A standard IMEC convolution example. Note that that values in input vector I and kernel K are chosen for illustration purposes only.

Algorithm 2 Algorithm for IMEC dataflow convolvers. The **for** loop sets are evaluated concurrently as k_w SIMD lanes with i_c PE units. After each $iteration$ there is no need to flush the buffer matrices as it is streaming in nature.

```
1: for $iteration \in 0 : (i_N)$ do
2:     for $h_{out} \in 0 : o_w$ do
3:         for $h_{in} \in 0 : k_w$ do
4:             for $w_{out} \in 0 : k_w$ do                ▷ Partitioned in separate SIMD lanes
5:                 for $w_{in} \in 0 : o_w$ do
6:                     for $i_{ctr} \in 0 : i_c$ do         ▷ Unrolled as PE units
7:                         $\boldsymbol{L_{IMEC}}[w_{out}, (h_{in} * k_w) + w_{in}, i_{ctr}] = \boldsymbol{I}[h_{out} + h_{in}, w_{out} + w_{in}, i_{ctr}]$
8:                         $\boldsymbol{O}[h_{out}, w_{in}] \mathrel{+}= \boldsymbol{L_{IMEC}}[(w_{out}, h_{in} * k_w) + w_{in}, i_{ctr}] \odot \boldsymbol{K}[(h_{in} * k_w) + w_{out}, 1]$
```

in FINN. This allows the IMEC to be seamlessly compiled to hardware through the BNN-PYNQ workflow. We use the framework to compile 1- and 2-bit QNNs targeting the PYNQ-Z2 hardware and utilise the drivers included by the tool flow. Vivado HLS 2019.2 is invoked at the back-end by the BNN-PYNQ workflow to generate the bitstream. The dataflow organisations and compute operations in the existing BNN-PYNQ framework are also updated for the IMEC dataflow convolver, as discussed below.

A. Organising the convolution window

As mentioned earlier, the first step in implementing the convolution window is the initialisation of the matrix L_{IMEC}. We use line buffers to process each row of the matrix and utilise a dataflow pipeline for lines 4 and 5 in alg. 2 to enable concurrent execution of the unrolled *for* loops. Further concurrency is achieved by splitting the matrix L_{IMEC} into k_w SIMD lanes (shown in line 4) each with i_c separate PE units handling the interleaved buffers. The *for* loop in lines 2 and 3 of the alg. 2 are sequentially laid out before streaming, replacing a conventional shifting logic in case of the outer loop. This process is repeated for every new image, indicated by the *for* loop in line 1.

B. Matrix-Accumulate

Once the dataflow architecture is established, the element-wise computation is adapted based on the IMEC algorithm. In case of multi-precision inputs, element-wise multiplication is performed between rows of L_{IMEC} and K, with a counter keeping track of the row index (line 8 in alg. 2). Accumulation occurs at the end of each row cycle. For quantised versions, pragmas specified in the design are used by the synthesis

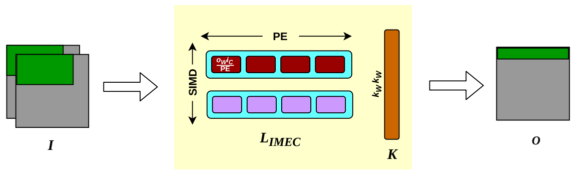

Fig. 3: Operation of the matrix-accumulate unit with 2 SIMD lanes and 4 PE units in case of IMEC in the BNN-PYNQ framework.

tools to target the resources (LUTs or DSPs) for the matrix-accumulate operations.

In case of single-bit convolution, the multiply operation is replaced by XOR operation while the accumulate operation is replaced by the *popcount* operation (i.e., counts the number of ones in each column buffer). The compute organisation is shown in Fig. 3 where each row coming from L_{IMEC} is fed to SIMD lanes, each with multiple PEs. The PEs' perform element-wise XOR with a single kernel matrix K of dimensions k^2. This operation with the kernel weight matrix is repeated across the multiple SIMD lanes and PE units. Parallel instances of this function fills the rows of the output matrix O.

V. Results

In this section, we evaluate the performance of the IMEC algorithm and compare it to the state-of-the-art data-streaming architectures. We use a QNN model trained through the BNN-PYNQ framework as the starting point for our evaluation. The model is then compiled to generate the bitstream using the standard BNN-PYNQ compilation flow (that infers the use of vanilla dataflow convolvers) to generate the vanilla FINN implementation (referred to as FINN-*vanilla*). The same QNN model is also compiled using the BNN-PYNQ framework and adapted FINN libraries that infers the IMEC headers for the convolution layer and optimises the IMEC dataflow convolver to generate the FINN-IMEC implementation. We evaluate the inference accuracy using the CIFAR-10 dataset on the PYNQ-Z2 platform; the IMEC-based implementation provided an identical inference accuracy of 80.1% as the FINN-vanilla implementation, demonstrating that an IMEC integration does not incur penalties in inference accuracy.

Next, we compare the resource savings that are achieved in case of a full CNV [31] model compiled through the FINN flow inferring IMEC dataflow convolver against the vanilla dataflow convolver (referred to as FINN (*IMEC*) and FINN (*vanilla*) respectively in the table(s)). Since IMEC aims at reducing the memory elements inferred, we focus on the LUT and LUTRAM consumption at each convolutional layer in the CNV model, which is extracted and plotted in Fig. 4. LUTRAMs are LUTs that are used as distributed storage elements (distributed RAM) by the design, with each 6-input LUT on the Xilinx 7-series device capable of acting as a configurable 64 × 1-bit memory. The x-axis of Fig. 4 corresponds to the different layers of the network, while the y-axis maps the LUT count. It can be observed from the chart that IMEC dataflow convolver achieves significant reduction in LUTRAM usage across layers 1 and 2

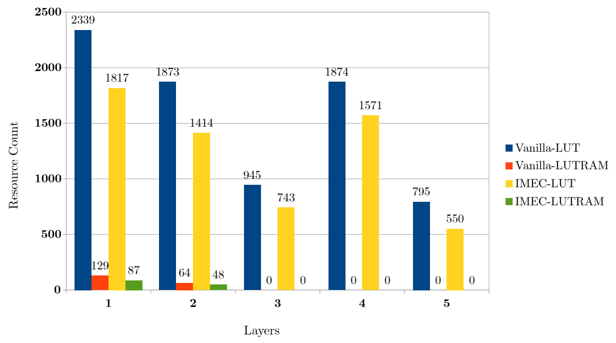

Fig. 4: A resource comparison between the convolution layers of a 1-bit quantised CNV model (FINN flow) implemented by using the vanilla dataflow convolver and IMEC dataflow convolver integrated into the FINN framework

of the model, with the higher order layers inferring no RAM elements. However, in the higher order layers, IMEC dataflow convolver is still able to reduce the number of LUTs inferred by a significant margin ($> 20\%$ across all three layers).

Table II captures the detailed resource consumption of the best and worst layers (in terms of LUTRAM/LUT resource reduction) across the different convolutional layers within the CNV model. The worst case scenario refers to the cases where IMEC dataflow convolver achieved zero or worse resource reduction over the vanilla dataflow convolver. Of the three convolutional layers with 0 LUTRAM reduction (seen in Fig. 4), layer-4 sees the smallest benefit in using IMEC convolver, with 16.7% reduction in LUTs and 0.7% increase in flip-flops (FFs) over a vanilla convolver-based implementation of this layer. In the best case (layer-1), the IMEC dataflow approach achieves nearly 20% reduction in the LUT count at the layer and 0.5% reduction in FF count, on top of the reduction in memory elements (LUTRAMs). We observe a 32.6% reduction in LUTRAMs inferred by layer-1 using the IMEC dataflow due to the optimisation of the intermediate buffers.

It is to be noted that both implementations use the same configuration for the sliding window and matrix-accumulate units of a convolution layer to ensure that other parts of the architecture remains the same, and use the same parameter settings: $i_w = 7$, $o_w = 5$, $k_w = 3$, $s_w = 1$ and $i_c = 2$. Also, in our benchmarks to generate HLS kernels, we used the `RAM_S2P_LUTRAM` pragma for memory operations and used the `Mul_LUT` pragma to perform MAC operations, in both implementations to ensure a fair comparison between the convolution approaches.

To isolate and quantify the memory optimisation that can be achieved within the convolutional layer, we implement a standalone 1-bit single-channel convolutional layer with the QCNN parameters $i_w = 7$, $o_w = 5$, $k_w = 3$, $s_w = 1$ and $i_c = 1$, both with the FINN-vanilla dataflow convolver and the FINN-IMEC dataflow convolver. This evaluation would enable the implementation tools to minimise optimisations driven by other layers of the network and should thus provide

TABLE II: Worst (layer-4) and best (layer-1) cases resource consumption breakup of single-bit convolution layers for the CNV model shown in Fig. 4

Accelerator	Resource Utilisation - Layer 4				Resource Utilisation - Layer 1			
	LUTs	FFs	BRAMs/DSPs	LUTRAMs	LUTs	FFs	BRAMs/DSPs	LUTRAMs
FINN (*vanilla*)	1874	887	0/0	0	2339	1189	0/2	129
FINN (*IMEC*)	**1517**	**894**	**0/0**	**0**	**1817**	**1181**	**0/2**	**87**
% savings	**+19.7%**	**-0.7%**	**-/-**	**-**	**22.3%**	**+0.5%**	**-/-**	**+32.6%**

TABLE III: Implementation and Performance Summary of FPGA-based Accelerators.

Accelerator	Framework	Model	Platform	Frequency (MHz)	Resource consumption				Resource saving v/s corr. FINN				Power (Watt)
					LUTs	LUTRAMs	FFs	DSPs	LUTs	LUTRAMs	FFs	DSPs	
LUTNET [9]	Tiled-LUTNET	CNV	Kintex XCKU115	200	106,776	3,786	216,513	184	-	-	-	-	6
FINN (1-bit) [11]	BNN-PYNQ	CNV	Zynq XC7Z020	200	29,635	2,438	42,053	24	1.0	1.0	1.0	1.0	1.793
This Work (1-bit)	BNN-PYNQ	CNV	Zynq XC7Z020	200	23,744	2,322	38,110	24	0.8	0.95	0.91	1.0	1.764
FINN (2-bit) [11]	BNN-PYNQ	CNV	Zynq XC7Z020	200	40,022	7,598	51,321	32	1.0	1.0	1.0	1.0	1.863
This Work (2-bit)	BNN-PYNQ	CNV	Zynq XC7Z020	200	35,001	7,273	43,738	32	0.87	0.96	0.85	1.0	1.828

the maximum achievable benefit on an FPGA device with 6-input LUT architecture. We implement the convolutional layer at multiple quantisation levels from 1-bit to 8-bit, capturing the LUTRAM utilisation in each case. Fig. 5 shows the inferred LUTRAM in case of the two implementations of the convolutional layer, with the quantisation (bit) levels on the x-axis and the number of LUTRAMs inferred on the y-axis. At each quantisation level, the IMEC-based implementation achieves close to 33.3% reduction in the number of LUTRAMs inferred, compared to the vanilla dataflow convolver. Thus, we can infer that IMEC convolver will offer significant LUTRAM reduction over the vanilla dataflow convolver at any quantisation level, which can be particularly important at higher quanisation level implementations of QCNNs on constrained FPGAs.

Finally, we also compare the overall FPGA utilisation of the entire QCNN network (with IMEC dataflow convolver) and the power consumption estimated by the Xilinx Power Estimator against other state-of-the-art FPGA NN accelerators, results

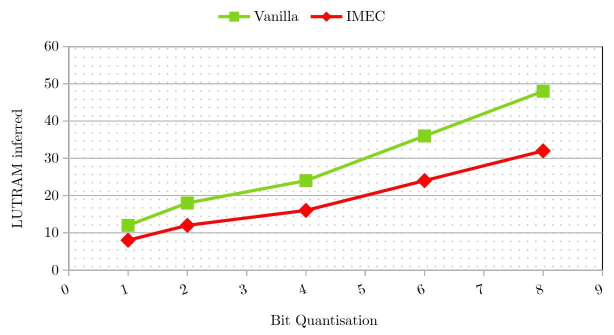

Fig. 5: Number of LUTRAMs incurred by the IMEC and Vanilla single-channel kernels across multiple quantisation levels (x axis) for the same convolutional layer design with parameters $i_w = 7$, $o_w = 5$, $k_w = 3$, $s_w = 1$ and $i_c = 1$.

of which are shown in table III. As seen in our evaluation on standalone convolutional layer, we can observe that the IMEC-based convolver achieves significant resource savings over the equivalent vanilla FINN implementation at both 1-bit and 2-bit precision, while achieving the same maximum operating frequency, throughput and inference accuracy. While the LUTRAMs inferred by the convolutional layers have reduced by over 30% across the overall network, the end-to-end datapath of the network incurs additional distributed storage elements during the implementation flow, thus restricting the overall LUTRAM reduction achieved by IMEC to 5% in both 1- and 2-bit implementations. However, the IMEC-based implementation incurs lower overall LUTs (20%, 13%) and FFs (9%, 15%) at both precision. Furthermore, the IMEC dataflow implementation is nearly 9% more energy efficient (tool estimate for all cases) than the vanilla dataflow convolver in the FINN implementation at both 1-bit and 2-bit precision levels at the same operating frequency of 200 MHz, making IMEC convolver a more resource and energy efficient choice for implementing QCNNs on constrained FPGA devices.

VI. Conclusion

In this paper, we presented the IMEC algorithm, a novel approach for optimising the memory resource utilisation of the convolution layers in CNNs. IMEC optimises the use of memory buffers by combining it with a re-organised data flow to lower the memory requirement and latency in streaming accelerators. The IMEC algorithm can be easily integrated into existing FPGA-based compilation frameworks – in this paper, we explored integration into the BNN-PYNQ flow as a header file from a trained model. Our results show that IMEC can enable a significant reduction in resource consumption within the convolution layers (at most 33% reduction in inferred distributed memory elements), and at the entire network with over 10% reduction in both LUTs and FFs at both 1-bit and

2-bit quantisation levels. Further, this resource optimisation also translates to reduced estimated power consumption ($\sim$9%), without any loss in inference accuracy state-of-the-art implementations. In the future, we aim to further optimise IMECs convolution algorithm by coalescing the sliding window with the matrix multiply accumulate units and integrating it to other end-to-end frameworks such as LUTNET [9] and Dnnweaver [19].

REFERENCES

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner., "Gradient-based Learning Applied to Document Recognition," Proceedings of the IEEE, 1998.

[2] A. Voulodimos, N. Doulamis, A. Doulamis, and E. Protopapadakis., "Deep Learning for Computer Vision: A Brief Review," 2018, Computational Intelligence and Neuroscience .

[3] K. Lee, J. Zung, P. Li, V. Jain, and S. Seung, "Superhuman Accuracy on the SNEMI3D Connectomics Challenge," 2017, arXiv preprint, 1706.00120.

[4] C. Zhang, P. Patras, and H. Haddadi., "Deep learning in mobile and wireless networking: A survey.," 2019, IEEE Communications surveys tutorials 21.3, 2224-2287.

[5] H. M. Song, J. Woo, and H. K. Kim., "In-vehicle Network Intrusion Detection Using Deep Convolutional Neural Network," 2020, Vehicular Communications 21.

[6] S. Zhou, W. Yuxin, Z. Ni, and et al., "Dorefa-net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients," 2016, arXiv preprint, 1606.06160.

[7] S. Han, H. Mao, and W. J. Dally., "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," 2015, arXiv preprint, 1510.00149.

[8] M. Courbariaux, B. Yoshua, and D. Jean-Pierre., "Binaryconnect: Training Deep Neural Networks with Binary Weights During Propagations," 2015, Advances in Neural Information Processing Systems.

[9] E. Wang, J. J. Davis, P. Y. Cheung, and G. A. Constantinides., "LUT-Net: Rethinking Inference in FPGA Soft Logic," 2019, International Symposium on on Field-Programmable Custom Computing Machines.

[10] S. Liang, L. W. Yin Shouyi Liu Leibo, and W. Shaojun., "FP-BNN: Binarized Neural Network on FPGA," 2018, Neurocomputing 275, 1072-1086.

[11] Y. Umuroglu, N. J. Fraser, G. Gambardella, and et al., "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," 2017, International Symposium on Field-Programmable Gate Arrays.

[12] C. Zhang, L. Peng, S. Guangyu, and et al., "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," 2015, International Symposium on Field-Programmable Gate Arrays.

[13] L. Lu, X. Jiaming, H. Ruirui, and et al., "An Efficient Hardware Accelerator for Sparse Convolutional Neural Networks on FPGAs," 2019, International Symposium on Field-Programmable Custom Computing Machines.

[14] C. Zhu, H. Song, M. Huizi, and W. J. Dally., "Trained Ternary Quantization," 2016, arXiv preprint, 1612.01064.

[15] M. Courbariaux, Y. Bengio, and J.-P. David., "Training Deep Neural Networks with Low Precision Multiplications," 2014, arXiv preprint, 1412.7024.

[16] S. Ioffe and C. Szegedy., "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," 2015, arXiv preprint, 1502.03167.

[17] L. Guerra, B. Zhuang, I. Reid, and T. Drummond., "Automatic Pruning for Quantized Neural Networks," 2020, arXiv preprint, 2002.00523.

[18] H. Yang, L. Duan, Y. Chen, and H. Li., "BSQ: Exploring Bit-Level Sparsity for Mixed-Precision Neural Network Quantization," 2021, arXiv preprint, 2102.10462.

[19] H. Sharma, J. Park, E. Amaro, and et al., "Dnnweaver: From High-level Deep Network Models to FPGA Acceleration," 2016, The Workshop on Cognitive Architectures.

[20] D. Caiwen, S. Liao, Y. Wang, and et. al., "CirCNN: Accelerating and Compressing Deep Neural Networks Using Block-circulant Weight Matrices," 2017, International Symposium on Microarchitecture.

[21] T. Posewsky and D. Ziener., "Throughput Optimizations for FPGA-based Deep Neural Network Inference," 2018, Microprocessors and Microsystems, 60.

[22] E. Wang, J. J. Davis, R. Zhao, and et. al., "Deep Neural Network Approximation for Custom Hardware: Where We've Been, Where We're Going," 2019, ACM Computing Surv. 52, 2, Article 40,

[23] S. I. Venieris and C.-S. Bouganis., "FpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs," 2016, International Symposium on Field-Programmable Custom Computing Machines.

[24] K. Guo, L. Sui, J. Qiu, and et.al., "Angel-Eye: A Complete Design Flow for Mapping CNN Onto Embedded FPGA," pp. 35–47, 2016, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[25] Y. Yu and N. K. Jha., "Spring: A Sparsity-aware Reduced-precision Monolithic 3D CNN Accelerator Architecture for Training and Inference," 2020, IEEE Transactions on Emerging Topics in Computing.

[26] S. Chetlur, C. Woolley, P. Vandermersch, and et. al., "CUDNN: Efficient Primitives for Deep Learning," 2014, arXiv preprint, 1410.0759.

[27] K. Chellapilla, S. Puri, and P. Simard., "High Performance Convolutional Neural Networks for Document Processing," 2006, International Workshop on Frontiers in Handwriting Recognition.

[28] Y. Guan, H. Liang, N. Xu, and et al., "FP-DNN: An Automated Framework for Mapping Deep Neural Networks Onto FPGAs with RTL-HLS Hybrid Templates," 2017, International Symposium on Field-Programmable Custom Computing Machines.

[29] M. Cho and D. Brand, "MEC: Memory-efficient Convolution for Deep Neural Network," 2017, International Conference on Machine Learning.

[30] T. Zhao, Q. Hu, X. He, and et al., "ECBC: Efficient Convolution Via Blocked Columnizing," 2021, IEEE Transactions on Neural Networks and Learning Systems.

[31] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing," pp. 1–13, 2016, International Symposium on Computer Architecture.