ISSN: 2584-0495 Vol. 2, Issue 3, pp. 651-657

International Journal of Microsystems and IoT

ISSN: (Online) Journal homepage: https://www.ijmit.org

Enhancing Image Segmentation with Optimized Winograd Algorithm for Convolution Neural Network

Ambati Sathvik, Mitul Tyagi, Shubham Kar, Brajesh Pandey and Sachin B Patkar

Cite as: Ambati, S., Tyagi, M., Kar, S., Pandey, B., & Patkar, S. B. (2024). Enhancing Image Segmentation with Optimized Winograd Algorithm for Convolution Neural Network. International Journal of Microsystems and IoT, 2(3), 651-657. https://doi.org/10.5281/zenodo.10884657

Published online: 11 March 2024

DOI: https://doi.org/10.5281/zenodo.10884657

International Journal of Microsystems and IoT
Vol. 2, Issue 3, pp. 651-657; DOI: https://doi.org/10.5281/zenodo.10884657

Enhancing Image Segmentation with Optimized Winograd Algorithm for Convolution Neural Network

Ambati Sathvik, Mitul Tyagi, Shubham Kar, Brajesh Pandey and Sachin B Patkar

Department of Electrical Engineering, Indian Institute of Technology Bombay, India

ABSTRACT
The U-Net architecture has emerged as a popular choice for image segmentation, encompassing convolution, max pooling, upsampling, and concatenation layers. The work aims on enhancing the Winograd algorithm used for convolutions, by introducing a Convolution Pooling Engine that incorporates alterations in the data transformation stage and integrates bias adjustments into the existing adaptations of Winograd convolution. This approach leads to a notable reduction of 7% in sub/add operations. Besides, the convolution computation immediately after an upsampling layer is often inefficient due to redundant data. To address this issue, we propose a novel Upsampling Convolution Engine that results in 25% reduction in add operations. Using HLS4ML flow, we have compared results of custom U-Net IP with HLS4ML IP. These IPs were integrated with MicroBlaze processor on Kintex (KC705 REV 1.2) board. We found that customised IP is having 3.2 times better latency.

KEYWORDS
Convolution pooling engine (CPE), upsampling convolution engine (UCE), visual geometric group (VGG)

1. INTRODUCTION

U-Net architecture is computationally demanding. Its major constituent is convolution layers. Faster algorithms for the convolutional computations reduce overall timing of neural networks. This can be achieved by use of minimal filtering algorithms. Use of such approaches minimizes the number of multiplications [1]-[2]. Input feature of the convolution layer can be transformed once and be re-used for convolving with all the filters, thereby reducing the time. Such designs are implemented using HLS [3]-[4].

There are multiple algorithms that can be employed for transformation of the convolution layer. Some of those are Strassen, Winograd, FFT-based convolution and so on. A normal n x n multiplication algorithm requires (n3 + O(n2)) arithmetic operations. Strassen algorithm for n x n matrix multiplication reduces the total number of operations by O(n2.82) by recursively multiplying 2n x 2n matrices using 7 n x n matrices. Winograd algorithm on the other hand almost halves the number of multiplications but also adds more addition operations. [5] FFT-based convolutions require more floating-point operations than its Winograd alternative for a moderate matrix size. [6] Strassen algorithm outperforms Winograd's algorithm when n is sufficiently large but for all practical cases of moderate matrix size, Winograd's algorithm performs 20% faster than Strassen's algorithm in the best-case scenario.

Therefore, we focus our attention on improving the efficiency to perform Winograd multiplication for our U-Net implementation rather than develop a general solution for all the transformation algorithms.

It is widely acknowledged that the fast Fourier transform (FFT) significantly reduces computational complexity for filter sizes comparable to input feature size. However, many modern-day filters are of size 3x3. The Winograd algorithm, based on the Chinese remainder theorem (CRT), utilizes the minimum number of multiplications for convolution and is suitable for small filters with a stride of 1 [7]. Efforts have been made to extend the Winograd algorithm to accommodate stride-2 convolution [8]. It is to be noted that pooling layer succeeding a convolution layer can be smartly avoided by performing the maximum operation at the end of the convolution layer itself [9]. We have used U-Net architecture to segment images pixel-wise [10]. In encoding stage, convolution layers are followed by 2 × 2 max-pooling layers. We introduce CPE which reduces number of sub/add operations. Further, introduction of UCE, eliminates the latency and hardware requirements associated with the upsampling layer. This enhancement contributes to faster and more efficient segmentation results. Additionally, the UCE also reduces the memory requirement and the latency of convolution layer in the UCE. By minimizing the memory footprint, the UCE enables the U-Net architecture to be more compatible with resource-limited environments. We used dataset from [11] – [12].

By use of industry standard implementation of IPs using Winograd algorithm as well as use of HLS4ML we have implemented and compared IPs [13]. To generate test pattern, we have used RBM [14]. On the contrary, if GAN is used for test pattern generation, Wino-transCONV can be used to make the GAN efficient [15].

This paper is organised as follows. Section 2 describes in detail, the proposed methodologies, HLS4ML flow and implementation details. Section 3 presents observed results while section 4 wraps this article with conclusion.

2. PROPOSED METHODOLOGY, HLS4ML FLOW AND IMPLEMENTATION DETAILS

2.1 Proposed methodology

Convolution Pooling Engine: The encoder stage of U-Net architecture generally comprises convolution layers followed by 2 × 2 max-pooling layers. The Convolution engine proposed in [9] elaborates the incorporation of the Winograd-based convolution to reduce the number of multiplication operations significantly, the implementation considered the input tile of 4 × 4 size. In this work, we aim to build on the idea of adopting Winograd algorithm to perform convolutions, thereby reducing the multiplication operations and also propose optimizations to this idea, to further reduce the number of sub/add operations involved at the data transformation (also referred as input transformation) and inverse transformation stages. Winograd algorithm is being used here instead of other transformation algorithms as the kernel size is not large enough to be optimal for Strassen [5] and neither can we afford more floating-point operations by using FFT-based convolution here as we will be using integer based Winograd transformation matrix. If instead, we were using a matrix of the form [-1/c, -c, c, 1/c], then we would have benefited from FFT-based convolution if the kernel as well as the image size was large enough [16].

Using the Winograd algorithm, the input tile d of size 4×4 is transformed into d′4×4 = BTdB, where B is also a 4×4 constant matrix [[1, 0, 0, 0], [0, 1,−1, 1], [−1, 1, 1, 0], [0, 0, 0,−1]]. The computation of d′ requires 32 sub/add operations. After computing the convolution of current input tile, the neighbouring 4×4 tile (say e) with stride 2 is then selected to transform. The overlap of such adjacent input tiles is thus 2 × 4 values. The sub/add operations performed amongst these common values can only be applied once for both the neighbouring tiles to reduce the overall sub/add operations. d′= BTdB =

$$\begin{bmatrix} \begin{matrix} d_{00}-d_{20} \\ -d_{02}+d_{22} \end{matrix} & \begin{matrix} d_{01}-d_{21} \\ +d_{02}-d_{22} \end{matrix} & \begin{matrix} -d_{01}+d_{21} \\ +d_{02}-d_{22} \end{matrix} & \begin{matrix} d_{01}-d_{21} \\ -d_{03}+d_{23} \end{matrix} \\ \\ \begin{matrix} d_{10}+d_{20} \\ -d_{12}-d_{22} \end{matrix} & \begin{matrix} d_{11}+d_{21} \\ +d_{12}+d_{22} \end{matrix} & \begin{matrix} -d_{11}-d_{21} \\ +d_{12}+d_{22} \end{matrix} & \begin{matrix} d_{11}+d_{21} \\ -d_{13}-d_{23} \end{matrix} \\ \\ \begin{matrix} -d_{10}+d_{20} \\ +d_{12}-d_{22} \end{matrix} & \begin{matrix} -d_{11}+d_{21} \\ -d_{12}+d_{22} \end{matrix} & \begin{matrix} d_{11}-d_{21} \\ -d_{12}+d_{22} \end{matrix} & \begin{matrix} -d_{11}+d_{21} \\ +d_{13}-d_{23} \end{matrix} \\ \\ \begin{matrix} d_{10}-d_{30} \\ -d_{12}+d_{32} \end{matrix} & \begin{matrix} d_{11}-d_{31} \\ +d_{12}-d_{32} \end{matrix} & \begin{matrix} -d_{11}+d_{31} \\ +d_{12}-d_{32} \end{matrix} & \begin{matrix} d_{11}-d_{31} \\ -d_{13}+d_{33} \end{matrix} \end{bmatrix}$$

Clearly the term di,j+2 is same as term ei,j for i ϵ {0, 3}, j ϵ {0, 1}. Without having to calculate afresh, operations among such similar terms can be reused. There are 8 such operations that can be bypassed for each input tile. Hence, only 24 sub/add operations will be required for the input transformation step of input tile, except for the first one of each row.

Element-wise multiplication is performed on the transformed input with the transformed filter and the channel-wise obtained 4 × 4 products are accumulated as single o′4×4 matrix.

At the inverse transformation step, the accumulated product term o′4×4 is multiplied with constant matrices A and AT, where A is a fixed matrix [[1, 0], [1, 1], [1, −1], [0, −1]], to obtain output of convolution o2×2, to which a pre decided bias value is added to all the four output values there by requiring 4 add operations for the bias addition step. However, the term o′11 is present in all the four output terms, so the bias can be directly initialised at corresponding index of the accumulator. The 4 × 4 product accumulator can now be initialised with all zeroes except the second column, second row, that is initialised with the bias itself.

$$o_{2\times2} = A^T o' A = \begin{bmatrix} \begin{matrix} o'_{00}+o'_{01}+o'_{02} \\ +o'_{10}+\mathbf{o'_{11}}+o'_{12} \\ +o'_{20}+o'_{21}+o'_{22} \end{matrix} & \begin{matrix} o'_{01}+\mathbf{o'_{11}}+o'_{21} \\ -o'_{02}-o'_{12}-o'_{22} \\ -o'_{03}-o'_{13}-o'_{23} \end{matrix} \\ \\ \begin{matrix} o'_{10}-o'_{20}-o'_{30} \\ +\mathbf{o'_{11}}-o'_{21}-o'_{31} \\ +o'_{12}-o'_{22}-o'_{32} \end{matrix} & \begin{matrix} \mathbf{o'_{11}}-o'_{21}-o'_{31} \\ -o'_{12}+o'_{22}+o'_{32} \\ -o'_{13}+o'_{23}+o'_{33} \end{matrix} \end{bmatrix}$$

The abstract view of the Convolution Pooling Engine CPE, consisting of the optimized input/data transform engine (PE1*), element-wise multiplication engine (PE2), accumulator, and the optimized inverse transform (PE3*), and the maximum function is shown in Fig. 1.

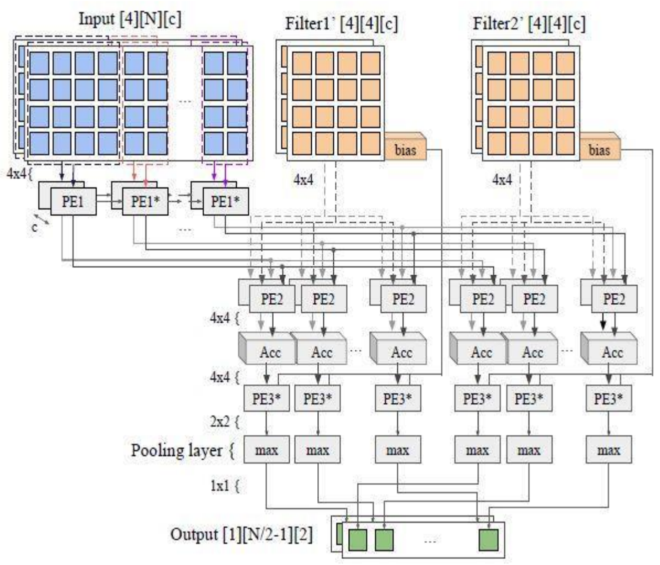


Fig. 1. Convolution Pooling Engine abstract view

Upsampling Convolution Engine: At the decoder stage of U-Net architecture [17] – [18], convolution layers often follow upsampling layers. The upsampling layer typically includes a line buffer with a single row and the same number of channels as the input channels. This line buffer serves to store the original input values and produces duplicates in a 2 × 2 fashion (nearest neighbour upsampling). In this scenario, the immediate convolution layer also has a line buffer with at least four rows and the same channels as in the upsampling layer buffer, shown in Fig.2.

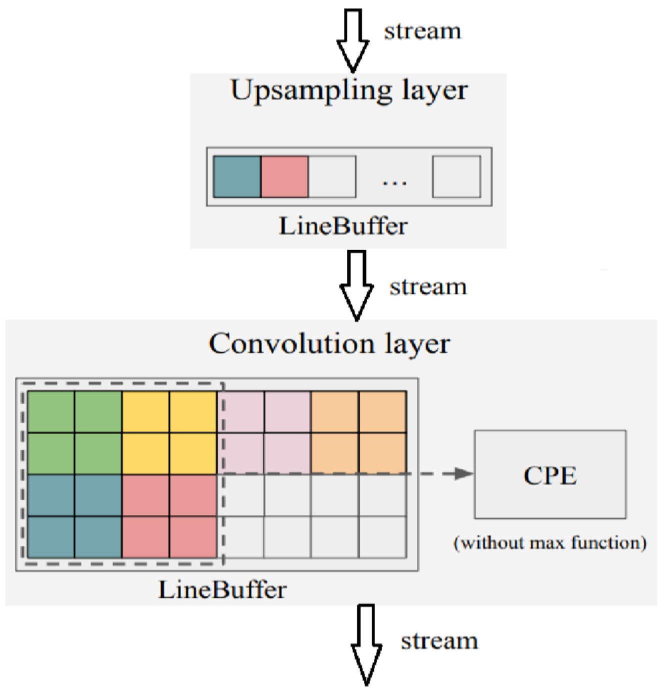


Fig. 2. Upsampling layer and convolution layer pair

Moreover, these many buffers are solely used for duplicating data. In order to mitigate the need for these additional buffers and leverage the duplicated data more efficiently, we bypass the upsampling layer and accept the inputs without being upsampled. Instead, we modify the input transformation of the convolution layer. The proposed implementation overview is shown in Fig.3.

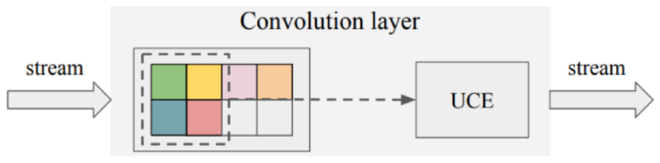


Fig. 3. Upsampling and convolution layers merged into a single convolution layer

To modify the input transform of the convolution, we first assume each 2 × 2 data is upsampled to form d4×4, and find

$$d = \begin{bmatrix} d_{00} & d_{00} & d_{01} & d_{01} \\ d_{00} & d_{00} & d_{01} & d_{01} \\ d_{10} & d_{10} & d_{11} & d_{11} \\ d_{10} & d_{10} & d_{11} & d_{11} \end{bmatrix}$$

the transformed input d′. Let e = (d00+d10), f = (d00−d10),

g = (d01 + d11), h = (d01 − d11), then

$$d' = \begin{bmatrix} f-h & f+h & -f+h & f-h \\ e-g & e+g & -e+g & e-g \\ -f+h & -f-h & f-h & -f+h \\ f-h & f+h & -f+h & f-h \end{bmatrix}$$

The input transform requires only 11 sub/add operations on the inputs. Though the input matrix d is considered for transformation, we require only the values d00, d01, d10 and d11. So instead of upsampling each 2 × 2 input tile to 4 × 4 and then transforming them, we now directly accept 2 × 2 input tile and apply the transformation as shown above by first evaluating the terms e, f, g, h.

After finding the convolution for this 2 × 2 input tile, we shall accept the next tile that is only 1 pixel distant from the previous. As the upsampled 4 × 4 inputs should have been 2 pixels apart, the actual versions should be 1 pixel apart. Similar to the input transformation of proposed CPE, the adjacent input tiles overlap can be utilized to decrease the operations count to only 9 from 11.

Implementing the upsampling, convolution layers using the proposed UCE minimizes the storage requirement greatly and also reduces the operation count of the input transformation stage from initial 32 to almost 9.

Convolution layer after concatenation: In the U-Net architecture, there are skip connections from convolution layers in the encoder stage to the convolution layers in decoder stage. These skip connections are meant to reuse the features learnt by the network at the pre-decoded stage for the encoded result and reduce noise in the final generated output. Such Winograd-based convolution layers results are computed tile-wise, meaning that the output values are first computed for out[i][j][c], out[i][j+1][c], out[i+1][j][c], and out[i+1][j+1][c] before out[i][j][c+1], out[i][j+1][c+1], out[i+1][j][c+1], and out[i+1][j+1][c+1]. Therefore, when these results are received at the decoder convolution layer, the data cannot be immediately processed as each input tile is of size 2 × 2. To start convolving the first 4 × 4 input tile, all the channels and the columns of the first row should be received totally. Moreover, if the corresponding encoder convolution is at the architecture beginning, the output channel count is usually very huge, thereby requiring to increase the storage capacity and wait time for its decoder stage convolution layer. In order to reduce the storage elements in such layers, which is a potential bottleneck, we utilized the following approach.

Instead of storing 4 input rows, each 2 × 2 input is assumed as 4 different inputs of size 4 × 4. Each of these 4 inputs contain the actual 2 × 2 input at one of the four corners, and zeroes as the remaining values. This way, we account for all the four configurations possible by that particular 2 × 2 input chunk. The Winograd engines are used to calculate the convolutions of all four 4 × 4 inputs. So, each input is received, processed immediately, the intermediate outputs are stored and the inputs can be discarded. Because the size of decoder stage convolutions is very small compared to their encoder counterparts, this technique results in overall storage savings. The configuration of such inputs having three quadrants of zero values simplifies the input transform to only sign reversal operations. Hence, we get rid of the input buffers by using more DSP blocks to perform the same task. Thus, we can trade off storage elements with computing elements which have good headroom.

Further, weight pruning is a technique that can be applied to CPE or UCE to reduce the number of active connections in the network. This can be done by identifying and removing the filters and inputs that have the least impact on the network's output. As a result, the network requires less area and computation to run [19].

2.2 Proposed methodology HLS4ML Flow

To write HDL code for complex systems like U-Net architectures takes time. To circumvent it, we have used HLS4ML platform to generate IP [20]. HLS4ML is used for generating HLS code for synthesizing based on the high-level model architecture defined in Keras/TensorFlow/PyTorch model, helping us in abstracting the model creation phase from the IP synthesis.

CNN implementation in HLS4ML is based on streams in HLS code, which in turn is synthesized as First In First Out (FIFO) buffers. Shift registers are used to track the track the last (h - 1) rows of input pixel where h is kernel height. We can maintain a shifting snapshot of the convolutional kernel as they are filled into the FIFO buffers via an I/O stream. As soon as the kernel is filled, the element-wise multiplication operation is carried out followed by the addition operations for each element of the convolutional kernel. We can prune the architecture for nodes with weights almost negligible before using the HLS code generated by HLS4ML for synthesis which can result in a reduced usage of multiplication operations but that would also result in a similar effect for the proposed IP implementation. Therefore, we have not pruned the model architecture by default in our comparison. HLS4ML implements the hardware design such that each layer of output values is calculated independently in a pipelined fashion to increase the throughput of the design.

The activation layer is implemented using LUTs for exponentials that are calculated for each input element to the activation layer. The LUT here is used with the input to the activation node being used as the lookup value in the table. This helps in using pre-computed values of activation if they are non-trivial in nature.

HLS4ML framework allows to set configurations such as input and filter dimensions, input padding (if required), strides, resolution of output feature maps for each layer of ML model. A flow depicting HLS4ML is shown in Fig.4. We used HLS4ML to generate IP which uses standard convolution. There are a multitude of other parameters we can change in HLS4ML itself which helps in us controlling the way the IP is synthesized. Some of them which are pertinent to our experiment are discussed here.

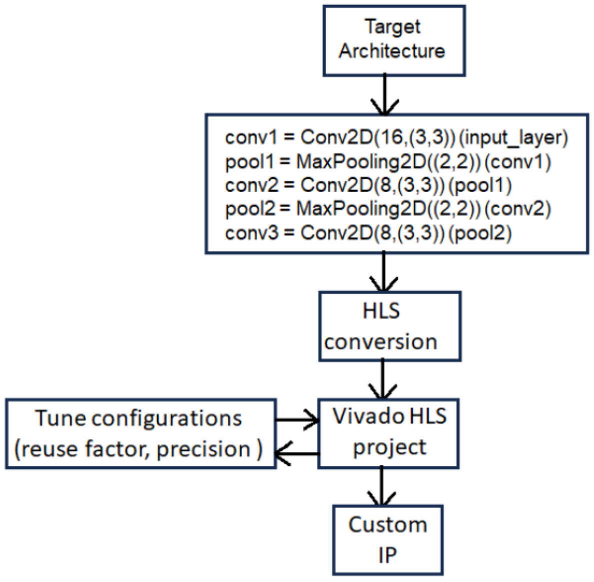

Target Architecture
conv1 = Conv2D(16,(3,3)) (input_layer)
pool1 = MaxPooling2D((2,2)) (conv1)
conv2 = Conv2D(8,(3,3)) (pool1)
pool2 = MaxPooling2D((2,2)) (conv2)
conv3 = Conv2D(8,(3,3)) (pool2)
HLS conversion
Tune configurations (reuse factor, precision)
Vivado HLS project
Custom IP

Fig. 4. HLS4ML flow for IP generation

We can set precision to be used for calculations in each convolution layer individually. We are using the default precision of fixed point<16,6> for models imported over from Keras since it is quite accurate out of the box. Increasing precision might yield diminishing returns with higher resource utilization.

Resources used for implementing the multiplication and the addition operations can be reused to reduce resource utilization at the expense of layer latency. The reuse factor can be explained easily using the following diagram. Using a reuse factor higher than one helps in increasing the IP usage while also implementing a sequential circuit rather than a combinational circuit for carrying out the arithmetic operations. We used the default reuse factor of one for our HLS4ML IP implementation to decrease the latency observed upon synthesis as shown in Fig.5.

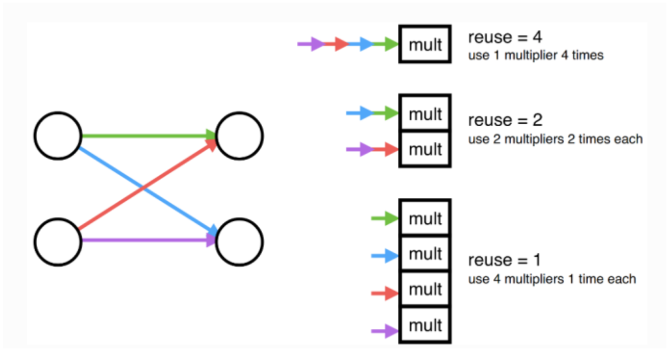


Fig. 5. Illustration of the reuse factor in HLS4ML

We have also not used a quantized Keras model for IP generation to keep the implementation strategy consistent with respect to the accuracy afforded by each arithmetic operation.

The suggested changes in the decoder of the proposed model were not modelled in HSL4ML as it requires a custom layer which performs upsampling along with convolution with no extra buffers being synthesized for the upsampling layer. Modelling that in HLS4ML as two separate layers one after the other would not work as that will result in FIFO buffers getting generated for each layer separately. Since our motivation is to reduce the memory requirement, we would be needed to program the layers at the HLS granularity rather than in the Python layer of abstraction. Hence, the original U-Net model was generated using HLS4ML using standard convolutional layer settings rather than the custom IP proposed for upsampling in the decoder stage.

Making a concatenated Upsampling Convolutional Engine would require a special layer that duplicates the same 2 x 2 input tile over a 4 x 4 grid in the four separate corners to directly feed into the decoder layers, hence bypassing any storage elements being used in the encoder stage for holding the values over for modelling skip connections. If we notice the architecture implemented in HLS4ML closely, we notice that its slightly different from the target architecture of U-Net wherein we are ignoring any skip connections in the HLS4ML IP since it does not model them properly as it interferes with the pipeline design generated by the tool.

2.3 Implementation details

For all reported results we have used Vivado tool kit for simulation purposes and used Kintex 7 FPGA KC705 kit for implementation of our IPs.

3. RESULTS

It is observed that by use of proposed modified Winograd convolutions CPE and UCE strategies there is reduction of around 9% and 25% respectively in the sub/add operations count over the standard Winograd convolution. For an input of size H × W, with C channels and K filters arithmetic complexity is shown in table I, Th is the number of horizontal tiles of a row.

Table 1 Reduction in arithmetic complexity

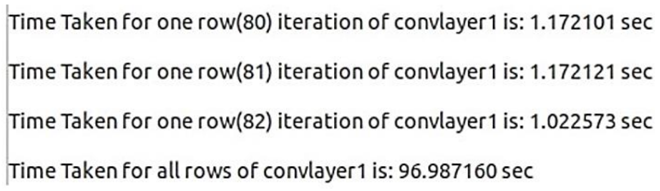
	Winograd Convolution	Proposed CPE	Proposed UCE
No of +/-	(32C+16CK+(24+4)K)T_h	8C+(24C+16CK+24K)T_h	2C+(9C+16CK+24K)T_h
No of +/- per output pixel	8C/K+4C+6+1	2C/KT_h +6C/K+4C+6	2C/KT_h +2.25C/K+4C+6
Encoder conv layer C=1, K=16	12	11	-
Decoder conv layer C=16, K=3	112	-	82

With encouraging results of proposed Winograd algorithm at algorithmic level implemented hardware, we incorporated proposed Winograd algorithm in U-Net architecture and checked the performance on implemented hardware. We observed that there is an increase in hardware resource utilization for modified Winograd based implementation with exception being FF. Even though there is increase in hardware resources, latency is better for proposed architecture by 4.8 times (according to HLS). Details of observations are summarised in table II.

Table 2 Latencies and core utilization of U-Net architectures

	U-Net Floating point	Proposed U-Net	
		Floating point	Fixed point (22-bit)
BRAM	241	414	440
DSP	39	167	435
FF	171 K	86 K	61 K
LUT	54 K	72 K	94 K
Latency (cycles)	18.9 M	3.9 M	0.83 M
Initial Interval (cycles)	18.3 M	2.2 M	0.67 M

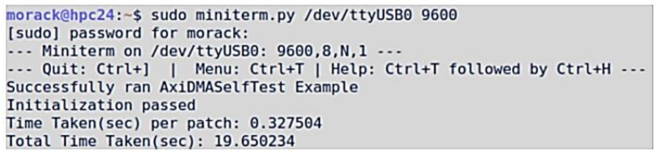
To show implication of our proposed modified Winograd algorithm work at IP level which mimic industry grade complexity, we used Vivado tool kit to integrate our IP with modified Winograd algorithm to 32-bit MicroBlaze processor on Kintex 7 board and observed execution timings. We observed that time taken for software implementation of conv1 layer is 96 s Fig.6.



```
Time Taken for one row(80) iteration of convlayer1 is: 1.172101 sec

Time Taken for one row(81) iteration of convlayer1 is: 1.172121 sec

Time Taken for one row(82) iteration of convlayer1 is: 1.022573 sec

Time Taken for all rows of convlayer1 is: 96.987160 sec
```

Fig. 6. Output of a single convolution layer using int16 on MicroBlaze with proposed IP

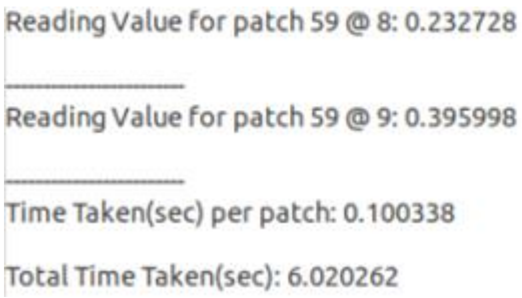
Estimated acceleration of IP is around 11,000 times. Screen grab of output is shown in Fig.7.



```
morack@hpc24:~$ sudo miniterm.py /dev/ttyUSB0 9600
[sudo] password for morack:
--- Miniterm on /dev/ttyUSB0: 9600,8,N,1 ---
--- Quit: Ctrl+]  |  Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
Successfully ran AxiDMASelfTest Example
Initialization passed
Time Taken(sec) per patch: 0.327504
Total Time Taken(sec): 19.650234
```

Fig. 7. Execution time of Winograd algorithm-based IPs

We observed these results at 70MHz. Further to this, we designed an IP with Winograd algorithm and integrated with MicroBlaze 32-bit processor on same hardware and compared results. We observed that latency is improved by 3.2 times (on FPGA) for modified Winograd algorithm at 70 MHz frequency, Fig.8.



```
Reading Value for patch 59 @ 8: 0.232728

-----------------------

Reading Value for patch 59 @ 9: 0.395998

-----------------------

Time Taken(sec) per patch: 0.100338

Total Time Taken(sec): 6.020262
```

Fig. 8. Execution time of modified Winograd algorithm-based IPs

Comparison of IPs performances using HLS4ML

We have compared results of 16-bit fixed point of default HLS4ML convolution scheme with 22-bit fixed point of proposed changes. Actual and normalized resource utilization as well latencies of IP generated from HLS4ML is given in Table III. Even though there are extra bits in our proposed IP hardware resources being utilized are very less compared to referenced IP. However, latency is higher. Further to this, we have compared our IP with other's work [2]. Observed results are reported in Table 4.

Table 3 Observed values of IPs generated by use of HLS4ML

	HLS4ML IP		Proposed IP	
	Actual Value	Normalized Value	Actual Value	Normalized Value
BRAM	2759	6.3	440	1
DSP	589	1.35	435	1
FF	845 K	13.7	62 K	1
LUT	1.5 M	15	94 K	1
Clock period	5 ns		10 ns	
Initial Interval	1.1 ms	0.12	9 ms	1
Latency	1.1 ms	0.1	11 ms	1

Table 4 Performance estimates of IPs

	Liqiang Lu IP	Proposed IP
Frequency	200 MHz	69 MHz
Devices	ZCU102	KC705
Logic cell efficiency (GOP/s/cells	4.9	1.1
DSP efficiency (GOP/s/DSPs)	1.16	0.23
Winograd PE	26	64
Input tile size	6	4
Total peak OP/cycle		1486
Total OP/cycle	14700	
Total GOPs		103.36
Total OP/cycle * freq	2940.7	

4. CONCLUSION

Our work focused on refining the implementation of the Winograd algorithm in the convolution layer of any CNNs, resulting in a significant reduction in the number of addition operations. Additionally, we introduced a novel and highly efficient approach tailored for handling upsampled convolutions in architectures similar to U-Net, leveraging an industry-grade environment for our experiments. With modified Winograd algorithm we have built IPs and compared results with Winograd IPs. Further to this we presented HLS4ML design methodology for IPs and results are compared for both the scenarios. Generated results are promising as our approach and framework to build IPs mimics industrial grade implementation practices. Compared results shows that modified Winograd algorithm could be an efficient way of implementing convolution algorithms.

REFERENCES

1. Lavin, A., & Gray, S. (2016). Fast algorithms for convolutional neural networks. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 4013-4021). https://doi.org/10.48550/arXiv.1509.09308
2. Winograd, S. (1980). Arithmetic complexity of computations (Vol. 33). Siam. https://doi.org/10.1137/1.9781611970364

3. Lu, L., Liang, Y., Xiao, Q., & Yan, S. (2017, April). Evaluating fast algorithms for convolutional neural networks on FPGAs. In 2017 IEEE 25th annual international symposium on field-programmable custom computing machines (FCCM) (pp. 101-108). IEEE. https://doi.org/10.1109/FCCM.2017.64
4. Huang, Y., Shen, J., Wang, Z., Wen, M., & Zhang, C. (2018, May). A high-efficiency FPGA-based accelerator for convolutional neural networks using Winograd algorithm. In Journal of Physics: Conference Series (Vol. 1026, No. 1, p. 012019). IOP Publishing. doi :10.1088/1742-6596/1026/1/012019
5. Brent, R. P. (1970, March). Algorithms for matrix multiplication. https://maths-people.anu.edu.au/~brent/pd/rpb002.pdf
6. Liu, X., & Turakhia, Y. (2016). Pruning of winograd and fft based convolution algorithm. In *Proc. Convolutional Neural Netw. Vis. Recognit.* (pp. 1-7). http://cs231n.stanford.edu/reports/2016/pdfs/117_Report.pdf
7. Cai, L., Wang, C., & Xu, Y. (2021). A real-time FPGA accelerator based on winograd algorithm for underwater object detection. Electronics, 10(23), 2889. https://doi.org/10.3390/electronics10232889
8. Yepez, J., & Ko, S. B. (2020). Stride 2 1-D, 2-D, and 3-D Winograd for convolutional neural networks. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 28(4), 853-863. https://doi.org/10.1109/TVLSI.2019.2961602
9. Podili, A., Zhang, C., & Prasanna, V. (2017, July). Fast and efficient implementation of convolutional neural networks on FPGA. In 2017 IEEE 28Th international conference on application-specific systems, architectures and processors (ASAP) (pp. 11-18). IEEE. http://dx.doi.org/10.1109/ASAP.2017.7995253
10. Cai, L., Wang, C., & Xu, Y. (2021). A real-time FPGA accelerator based on winograd algorithm for underwater object detection. Electronics, 10(23), 2889. https://doi.org/10.3390/electronics10232889
11. Burguera, A. (2021). Multi-Class Segmentation of Side-Scan Sonar Data using a Neural Network. Mallorca (Spain). Retrieved September 25, 2022, from https://github.com/aburguera/NNSSS
12. Burguera, A., & Bonin-Font, F. (2020). On-line multi-class segmentation of side-scan sonar imagery using an autonomous underwater vehicle. Journal of Marine Science and Engineering, 8(8), 557. https://doi.org/10.3390/jmse8080557
13. FastML Team. (2023). hls4ml (v0.7.1). Zenodo. https://doi.org/10.5281/zenodo.7933047
14. Hebron, P. (2017). Learning machines. https://www.patrickhebron.com/learning-machines/
15. Di, X., Yang, H. G., Jia, Y., Huang, Z., & Mao, N. (2020). Exploring efficient acceleration architecture for winograd-transformed transposed convolution of GANs on FPGAs. Electronics, 9(2), 286. https://doi.org/10.3390/electronics9020286
16. Zlateski, A., Jia, Z., Li, K., & Durand, F. (2018). A Deeper Look at FFT and Winograd Convolutions. https://mlsys.org/Conferences/doc/2018/28.pdf
17. Nikhil, T. (2021). What is UNET? https://idiotdeveloper.com/what-is-unet
18. Ronneberger, O., Fischer, P., & Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. In Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18 (pp. 234-241). Springer International Publishing. https://doi.org/10.48550/arXiv.1505.04597
19. Lu, L., & Liang, Y. (2018, June). SpWA: An efficient sparse winograd convolutional neural networks accelerator on FPGAs. In Proceedings of the 55th Annual Design Automation Conference (pp. 1-6). https://doi.org/10.1145/3195970.3196120
20. Duarte, J., Han, S., Harris, P., Jindariani, S., Kreinar, E., Kreis, B., Ngadiuba, J., Pierini, M., Rivera, R., Tran, N. & Wu, Z. (2018). Fast inference of deep neural networks in FPGAs for particle physics. Journal of instrumentation, 13(07), P07027. https://doi.org/10.1088/1748-0221/13/07/P07027

Ambati Sathvik received his BTech degree in electrical engineering from IIT Tirupati, India in 2021 and MTech degree in integrated circuit and systems from IIT Bombay, India in 2023. His areas of interest are digital VLSI design and cryptography.

Email: sathvikambati123@gmail.com

Mitul Tyagi is a Digital Design Engineer in the TI Radar Business Unit, focusing on the validation and verification of Radar SoCs. He holds a B.Tech degree in Electronics and Communication from BIAS, India, and an M.Tech degree from IIT Bombay, India. Mitul's expertise lies in VLSI Design, Firmware Development, and Embedded Software Development.

Email: mitultyagi45@gmail.com

Shubham Kar is currently a firmware engineer in NVIDIA Graphics Private Limited, primarily responsible for firmware development for DGX GPU systems being used by CSPs. He received his B.Tech degree in Electrical Engineering from IIT Bombay, India in 2022.His areas of interest are VLSI Design, Graphics Programming, and low level System Software.

Email: karshubham257@gmail.com

Brajesh Pandey possess more than two decades of experience in the field of VLSI Design. He has hand-on delivery of Silicon proven products in academia as well as MNCs. As faculty member of top tier institutes, he has nurtured best talents in India. He received his MSc degree in Electronics from Gorakhpur University in 1998, M.Tech degree in Microelectronics from Panjab University Chandigarh in 2001 and PhD degree in VLSI Design from IIT Bombay in 2011. His areas of interest include Hardware Software Co-design, System Design, SoC design and verification, ASIC development and Novel Device Design, modeling and characterization.

Email: brajesh153@gmail.com

Sachin B Patkar is a Professor in the Department of Electrical Engineering of IIT Bombay. He is head of high performance embedded computing (HePC) lab. He received his B.Tech degree in Computer science from IIT Bombay in 1986, M.Tech degree in Computer Science from IIT Madras in 1987 and PhD degree in Computer Science from IIT Bombay in 1992. His research area of interest includes, Combinatorial Optimization, High Performance Computing, Algorithm Design and Analysis, Graph Theory, Geometric Design and Graphics and Software/Hardware Development projects.

Email: patkar@ee.iitb.ac.in