Aditya Gupta, Miguel Ortiz Lopez,
Sanjeeth Ayanala, Vineet Rao

Winograd Convolution for 8-bit Precision with flexible input size

1. Introduction

Convolutional neural networks (CNNs) have been widely used in computer image processing. Many embedded applications have limited performance, and the power consumption and CNNs are computationally intensive and consume large amounts of processing time, making it challenging to apply them to these types of applications. Low-precision computations are considered promising methods to accelerate convolutional neural networks and have been demonstrated that 8-bit precision has a minimal accuracy degradation. The Winograd algorithm is a fast convolution method that transforms inputs and weights into another domain, where convolution becomes element-wise multiplication and then returns the results [1]. This transformation reduces the computational theoretical complexity, but the main problem that transformations bring is the loss of the accuracy of the results. As the transformation matrices (Vandermonde matrices) have bad properties in the real field, the numerical error increases exponentially with the output size [2]. However, Winograd Convolution and low-precision computation are not widely combined due to the numerical error introduced. This project explores the loss caused by the matrix transformation using 8-bit low-precision linear quantization. This type of architecture can be beneficial for applications that explore performing Machine learning on small and low-powered devices.

2. Project Flow

2.1 Background

Quantization

Quantization is one of the most impactful ways to decrease the computational time and energy consumption of neural networks. In neural network quantization, the weights and activation tensors are stored in lower bit precision than the higher bits they are usually trained in. If we quantize from N to $\frac{N}{2^N}$ bit representation, then the memory overhead of storing model weights decreases by a factor of 2^N while the computational cost for matrix multiplication decreases significantly by a factor of 2^{2N}. Neural networks have been shown to be robust to quantization, meaning they can be quantized to lower bit widths with a relatively small impact on the network's accuracy.

The major downside of neural network quantization is that Low bit-width quantization introduces noise to the network that can lead to a drop in accuracy. One must make sure, there is not a lot of decrease in accuracy while accounting for the quantization of neural networks.

Types of Uniform Quantization:

1) Asymmetric Quantization: Also known as asymmetric quantization is defined by three quantization parameters:
    a) scale factor: s

b) zero-point: zp
c) Bit-width: b

The scale factor and the zero-point are used to map a floating-point value to the integer grid, whose size depends on the bit-width. The scale factor is commonly represented as a floating-point number and specifies the step size of the quantizer. The zero-point is an integer that ensures that real zero is quantized without error. This is important to ensure that common operations like zero padding or ReLU do not induce quantization error.

2) Symmetric Quantization:

Symmetric quantization is a simplified version of asymmetric quantization. The symmetric quantization maps the zp to 0. This reduces the computational overhead of dealing with zero-point offset during the accumulation operation. But the lack of offset restricts the mapping between integer and floating-point domains.

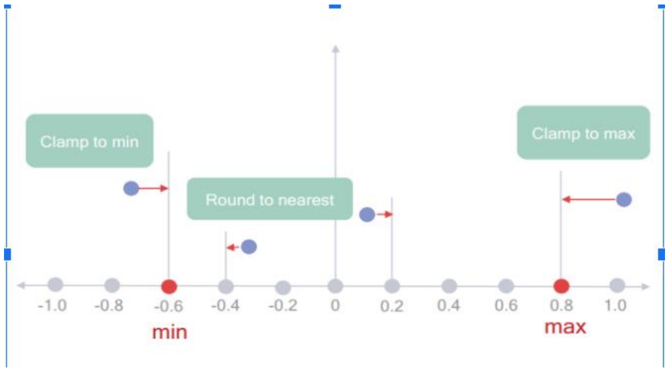


Figure 1. Conventional quantization scheme

Types of Quantization Errors:

1) Rounding Error: Arises due to rounding the floating-point values to the integer scale.
2) Saturation Error: Arises due to clipping the values that fall outside the quantization range.

At each layer, error accumulates due to the following above errors and leads to decrease inaccuracy of the model. So extra investigation must be done before quantizing the ML models.

Model Quantization techniques: (for tinyML applications):

How we quantize models depends upon the level of the given ML model.

1. Simple 8-bit Quantization: Quantize floating-point model parameters to 8-bit and don't do anything else.
2. Post Training: In-place changes in the model are required -weights rescaling, quantization ranges rescaling.
3. Fine Tuning: Needs significant fine-tuning afterward (Cross-layer weights equalization).
4. Architecture Changes: The model needs to be trained from scratch considering quantization, we generally try to avoid this as this includes a lot of computational time and energy.

For this course project, we are assuming our model to be simple 8-bit type model and we are just quantizing the model parameters by quantization schemes.

Cross-layer weight equalization and Ada Round which reduces the Frobenius norm $||W - \widehat{W}||_F$ and $||Wx - \widehat{Wx}||_F$ is left for the future scope due to the time constraint.

Winograd algorithm:

Winograd convolution is based on Winograd minimal filtering algorithm that computes an $m \times m$ output matrix Y by convolving $(m + k - 1) \times (m + k - 1)$ input matrix d with $k \times k$ kernel g as shown in figure 1. It reduces the number of multiplications at the cost of additions [3].

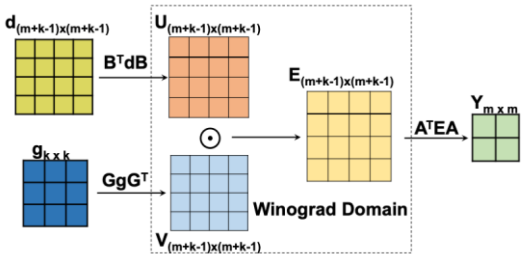


Figure 2. Patches in a 7x7 Image

A convolution layer in CNN with $k \times k$ convolution kernel can be computed using the Winograd algorithm with a configuration of $F(m \times m, k \times k)$. In our convolution engine we use $F(2 \times 2, 3 \times 3)$. The Winograd Equation is as follows:

$$Y = A^T[[GgG^T] \odot [B^TdB]]A$$

For the $F(2 \times 2, 3 \times 3)$ the following arrays are used (calculated using the Winograd algorithm)[1].

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

Looking at the Winograd equation we see that the matrix GgG^T can be computed offline and stored in the engine. B^TdB reduces to additions and subtractions.

2.2 High-Level Description and workflow

In this project, we aim to use our Winograd engine (implemented in hardware) as a part of an inference cycle (performed in Matlab). In our example, we use weights trained for the MNIST fashion dataset and then perform the inference cycle with the convolution performed through the Verilog implementation

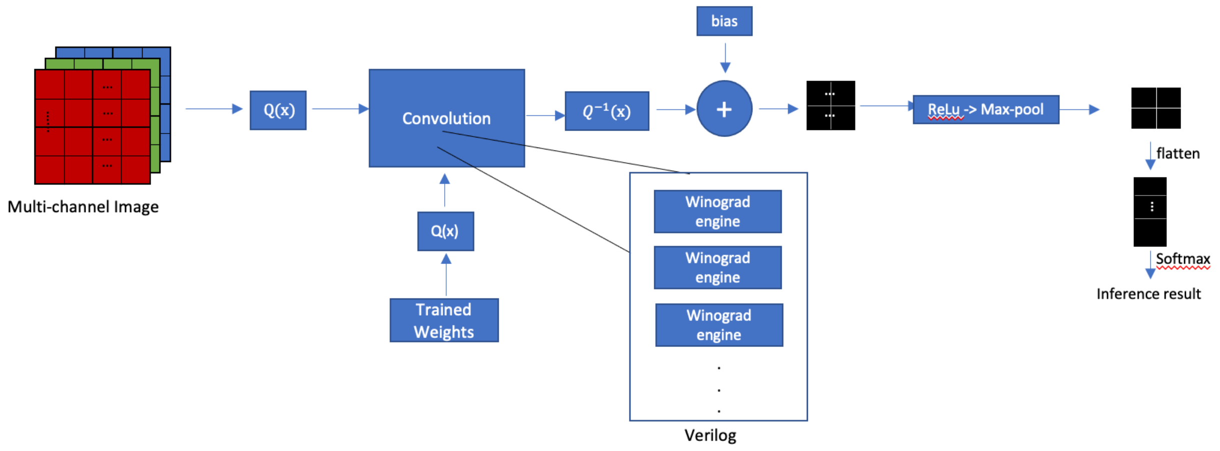


Figure 3. High-Level Description

Figure 2 represents the overall flow of our project and gives an overview of the different elements that we aimed to investigate. The key components of our project are the quantization blocks, flexible input Winograd convolution engine, and finally operations to perform inference (ReLu, Max-pool, Flatten, and Softmax).

2.3 Design Decision and Our architecture

For our project, we have attempted to implement the mathematical equations in the Verilog engine by pre-calculating the additions for the $[GgG^T]$ and $[B^T dB]$ matrices. The reason why we chose such a simplistic design is to reduce the complexity of the design and reduce the delay for our specific design. Previous work used an FPGA accelerator to increase the performance due to FPGAs' flexibility, energy consumption, and performance. The chip computation engine optimization permits highly parallel strategies by customizing parallel processing elements. These implementations can achieve high performance that exceeds CPUs, but we do not choose this architecture because our weights are constant. Architectures like parameterized systolic arrays and hierarchical memory subsystems [3] are used when the same engine is used with different weight filters as inputs. Our architecture needs to access memory only to read the new inputs and not the filters. Thus, we use a patch-based inference with reduced overlap similar to the MCUNetV2 with a stride of 3[4]. The details of the implementation are in section 3.1. (Module functionality). When applying it to our Winograd engine, due to the static nature of filters (no on the fly computation for the GgG^T matrix) and simplified B^T and A^T matrices we can just translate the matrix multiplications into simple additions in hardware. We can then use multiple of these units to make a complete convolution unit as shown in figure 2. In our design, a single unit of the Winograd engine can take up to *100* × *100* × *3* data. Most practical datasets, such as the MNIST fashion dataset that we are using, utilize an image size within this range. In this way, each unit of our engine can accommodate flexible input sizes and channels in the range specified above.

3. Module descriptions and their Implementation Details

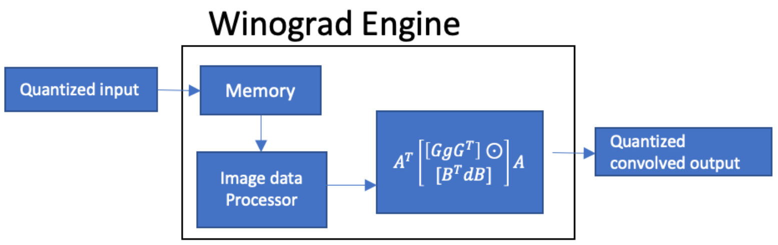

Winograd Engine
Quantized input
Memory
Image data Processor
$A^T \begin{bmatrix} [GgG^T] \odot \\ [B^T dB] \end{bmatrix} A$
Quantized convolved output

Figure 4. Winograd Engine

3.1 Data Processor

This module takes the quantized data from the memory and sends it to the Convolution Module in a patch-wise manner.

3.1.1 Block Diagram

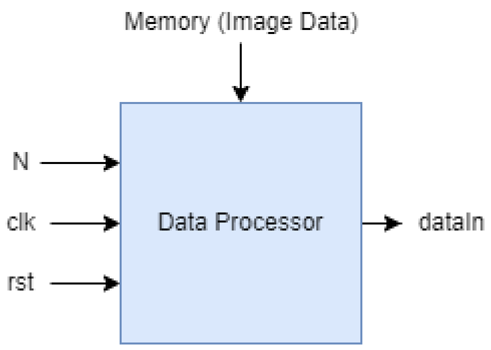


Figure 5. Block Diagram of the data processing module

3.1.2 Port Description

Parameter	**Input/Output Type**	**Data Type**	**Description**
N	Input	2-bit Integer	Indicates the number of input channels
W	Input	8-bit Integer	Indicates the width of input image

H	Input	8-bit Integer	Indicates the height of input image
dataIn1	Output	128-bit Integer	(4*4) patch corresponding to channel 1 (i.e., 16 values with 8 bit each)
dataIn2	Output	128 bit Integer	(4*4) patch corresponding to channel 2 (i.e., 16 values with 8 bit each)
dataIn3	Output	128-bit Integer	(4*4) patch corresponding to channel 3 (i.e., 16 values with 8 bit each)

3.1.3 Module Functionality

This Module takes data from the memory, packs the data as a patch, and sends the patch data to the convolution module. Each cycle, this module sends 1 patch data per channel to the convolution module by picking the values from the memory from appropriate positions. Since the memory is 1-Dimensional, and the patch data is not completely different from the other patches, the data cannot be directly driven in a sequential manner. Given an example of a 7x7 image (Figure 5) with 4 patches (each 4x4). We can clearly see that the patches overlap with each other in figure 5, the overlap is minimum due to the stride of 3.

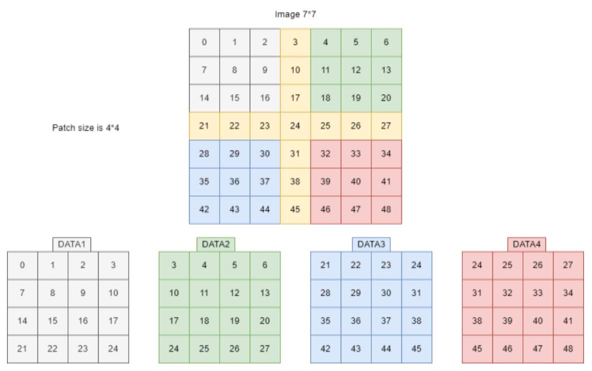


Figure 6. Patches in a 7x7 Image

From figure 5, we can see that the way of picking the data from the 1-Dimensional memory is not sequential. For example, in a patch_width = 4 and patch_height = 4, the procedure of picking the different patch’s data from the memory is represented in Figure 6.

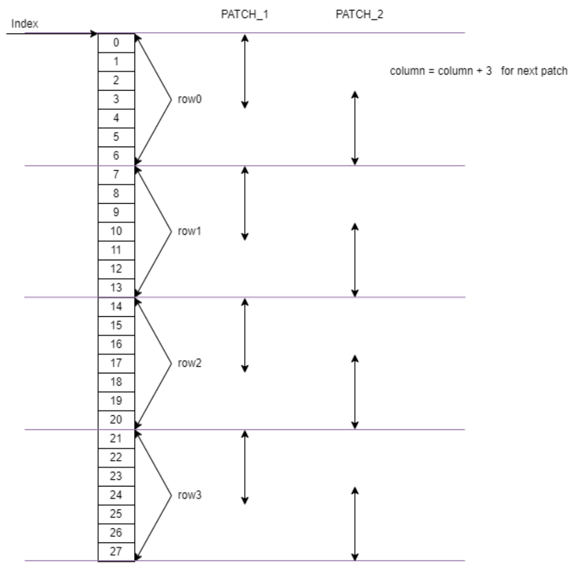


Figure 7. 1-D Memory representation for first two patches

For patches 1, the start position is set to Index (Index is zero as shown in figure 6). While computing patch 1 (4*4 matrix = 16 values), the positions of 16 values are given below.

Row 1	4 values from Index
Row 2	4 values from Index+ width
Row 3	4 values from Index+ width*2
Row 4	4 values from Index+ width*3

For patch 2, the start Index value is incremented by 3. While computing patch 1 (4*4 matrix = 16 values), the positions of 16 values are directly calculated by replacing (Index) with (Index+3) in the above table.

Patches 3 and 4 are computed as shown in Figure 8.

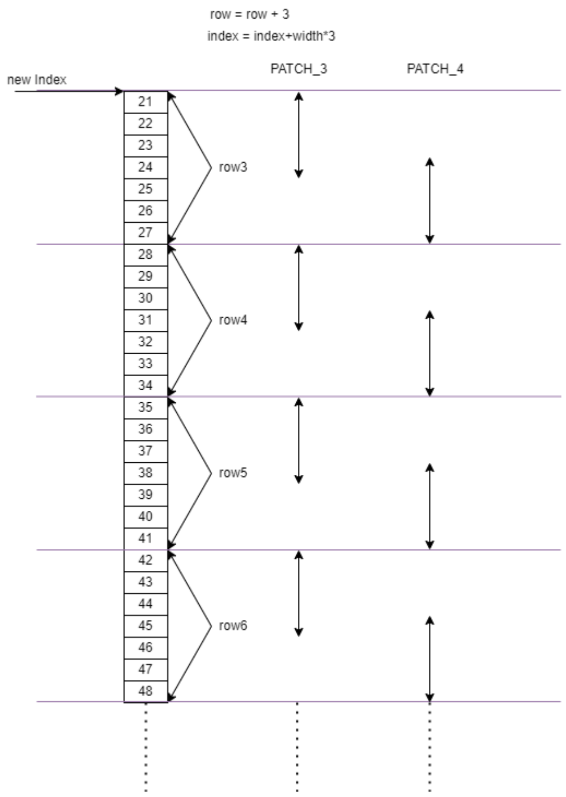


Figure 8. 1-D Memory representation for last two patches

While computing patch 3 and 4 (4*4 matrices), the start position changes to new index (new_index = old_index + width*height). Now, the same procedure is repeated to find patches 3 and 4.

3.2 Convolution Engine

This module takes the input data from the Data Processor module, performs the convolution, whose output is then taken into MATLAB for de-quantization.

3.2.1 Block Diagram

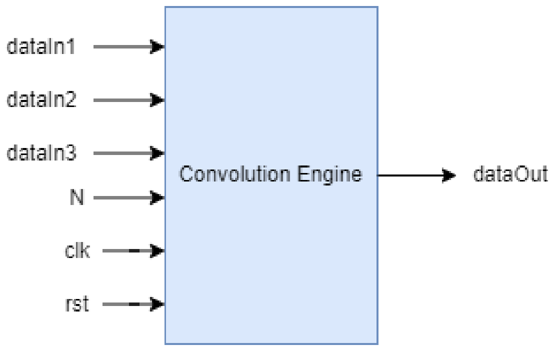


Figure 9. Convolution Engine block diagram

3.2.2 Port Description

Parameter	**Input/Output Type**	**Data Type**	**Description**
dataIn1	Input	128-bit Integer	(4*4) patch corresponding to channel 1 (i.e., 16 values with 8 bit each)
dataIn2	Input	128-bit Integer	(4*4) patch corresponding to channel 2 (i.e., 16 values with 8 bit each)
dataIn3	Input	128-bit Integer	(4*4) patch corresponding to channel 3 (i.e., 16 values with 8 bit each)
N	Input	2-bit Integer	Indicates the number of input channels
dataOut	Output	64-bit Integer	(2*2) output patch after Convolution (i.e., 4 values with 16 bit each)

3.2.3 Module Functionality

This Module takes the input from Data Processor Module, Performs the Winograd Convolution, and sends out the convoluted patch. This module has been designed to support a maximum of 3 channels (can be extended to more channels with some minimal changes in the design). Based on the input received from the data processor module (dataIn1, dataIn2, dataIn3, N), the module performs the convolution in parallel for N channels. The Winograd Convolution function is:

$$Y = A^T[[GgG^T] \odot [B^TdB]]A$$

GgG^T is computed offline and stored in an LUT (G[0], G[1], ... , G[15]). (Since $GgG^T$is a 4*4 matrix => 16 values).

Each cycle, one patch per channel (16 values) is the input for the Convolution Engine. These 16 values are stored in an LUT (d[0], d[1], ... , d[15]).

Now, to compute B^TdB, since the B matrix is constant, the computations are done in MATLAB, and hardware efficient equations are computed and directly implemented in Verilog. Each cycle, data (d) corresponding to each channel is received and B^TdB(4*4 Matrix with 16 values) is computed using the equations given below.

D[0] = d[0] - d[2] - d[8] + d[10];
D[1] = d[1] + d[2] - d[9] - d[10];
D[2] = d[2] - d[1] + d[9] - d[10];
D[3] = d[1] - d[3] - d[9] + d[11];
D[4] = d[4] - d[6] + d[8] - d[10];
D[5] = d[5] + d[6] + d[9] + d[10];

D[6] = d[6] - d[5]- d[9] + d[10];
D[7] = d[5] - d[7] + d[9] - d[11];
D[8] = d[6] - d[4] + d[8] - d[10];
D[9] = d[9] - d[6] - d[5] + d[10];
D[10] = d[5] - d[6] - d[9] + d[10];
D[11] = d[7] - d[5] + d[9] - d[11];
D[12] = d[4] - d[6] - d[12] + d[14];
D[13] = d[5] + d[6] - d[13] - d[14];
D[14] = d[6] - d[5] + d[13] - d[14];
D[15] = d[5] - d[7] - d[13] + d[15];

Where D is an array that holds the 16 values of $B^T dB$ *Matrix.*

We now have $B^T dB$ and GgG^T. The pointwise multiplication of $B^T dB$ and GgG^T is directly implemented in hardware as given below.

F[0] = G[0] * D[0];
F[1] = G[1] * D[1];
F[2] = G[2] * D[2];
F[3] = G[3] * D[3];
F[4] = G[4] * D[4];
F[5] = G[5] * D[5];
F[6] = G[6] * D[6];
F[7] = G[7] * D[7];
F[8] = G[8] * D[8];
F[9] = G[9] * D[9];
F[10] = G[10] * D[10];
F[11] = G[11] * D[11];
F[12] = G[12] * D[12];
F[13] = G[13] * D[13];
F[14] = G[14] * D[14];
F[15] = G[15] * D[15];

Where F is an array that holds the 16 values of $B^T dB$ *Matrix.*

We have $[GgG^T] \odot [B^T dB]$ which is ‘F’. To compute, $A^T * F * A$, we use the below equations.

o[0] = F[0] + F[1] + F[2] + F[4] + F[5] + F[6] + F[8] + F[9] + F[10];
o[1] = F[1] - F[2] - F[3] + F[5] - F[6] - F[7] + F[9] - F[10] - F[11];
o[2] = F[4] + F[5] + F[6] - F[8] - F[9] - F[10] - F[12] - F[13] - F[14];
o[3] = F[5] - F[6] - F[7] - F[9] + F[10] + F[11] - F[13] + F[14] + F[15];

Where ‘o’ is an array which holds the 4 values of $A^T[[GgG^T] \odot [B^T dB]]A$ Matrix.

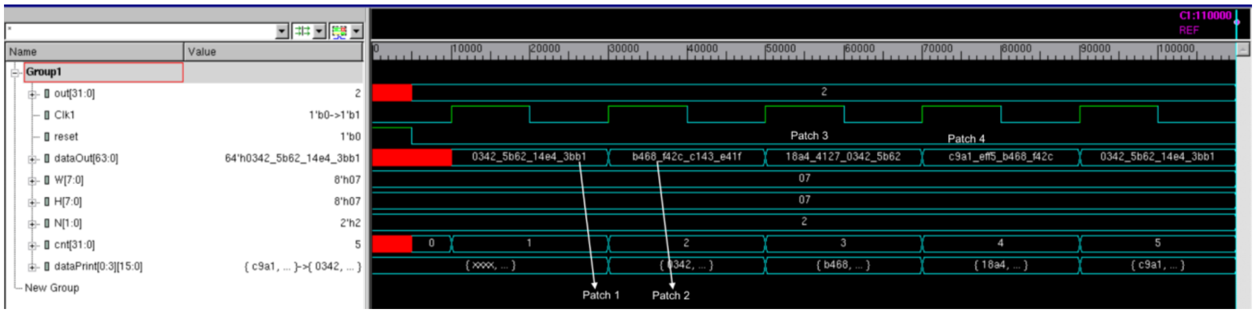
These four values are packed together and sent out as a 64-bit value as given below with each value being a 16-bit output:

dataOut = {o[0], o[1], o[2], o[3]};

4. Results

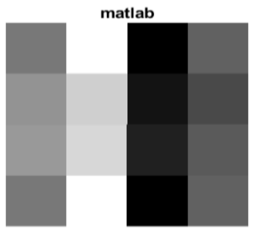
4.1 Verilog output vs MATLAB output and their error results

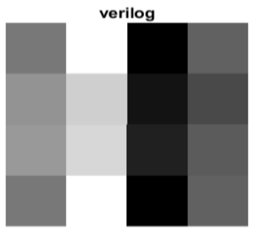
To test the correctness of the outputs of the Winograd engine in Verilog, we compared the values generated in the DVE simulation with the values obtained from the Verilog. Figure 10 shows the quantized outputs in HEX obtained from the engine and Figure 11 (b) shows their dequantized values. Looking at Figure 11(a) we realize the results are close.



Name	Value
Group1	
out[31:0]	2
Clk1	1'b0->1'b1
reset	1'b0
dataOut[63:0]	64'h0342_5b62_14e4_3bb1
W[7:0]	8'h07
H[7:0]	8'h07
N[1:0]	2'h2
cnt[31:0]	5
dataPrint[0:3][15:0]	{ c9a1, ... }->{ 0342, ... }
New Group	

Figure 10. DVE simulation results





1	2	3	4
7.6550	185.0036	-150.6606	-23.4500
42.5284	121.9475	-124.7742	-54.6837
50.5310	132.5558	-107.7963	-31.3444
7.6550	185.0036	-150.6606	-23.4500

(a) No quantization Winograd convolution in matlab

1	2	3	4
6.5156	182.7656	-151.1875	-23.6563
41.7813	119.3828	-125.4766	-55.7578
49.2813	130.3047	-108.7422	-32.0859
6.5156	182.7656	-151.1875	-23.6563

(b) De-quantized(in MATLAB) Verilog values

Figure 11. Comparison Verilog vs MATLAB.

4.2 MATLAB inference

An inference cycle runs trained weights with an input image to get an actual classification output. We built this in MATLAB with the aim of determining the accuracy of inference from our low-bit precision engine vs. from a 32-bit inference cycle. Figure 12 shows the accuracy between inference from the two engines. 100% accuracy implies no discrepancy in the predictions from the 32-bit inference cycle and the x-bit inference cycle (x < 32).

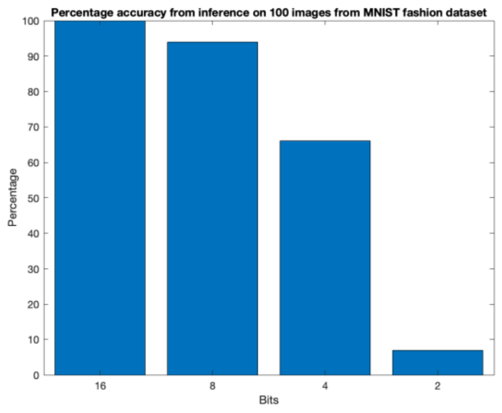


Figure 12. Percentage accuracy from inference

5. Conclusion

In this project, we proposed a Winograd Convolution using low-precision quantization. We show that in our tests, the accuracy of 8 bits quantization transformations is good enough for the Winograd algorithm with kernel 3x3, output 2x2, and a stride of 3. For 8 bits precision, Winograd had a 6% accuracy loss compared to Winograd in floating-point precision. Lower precision induces error in each convolution that reduces accuracy at least by 44%. This experimental result suggests that Winograd convolution can speed up with trivial accuracy loss. However, different filters and output sizes need to be evaluated in order to determine the limitation of Winograd Convolution. The transformation used for this implementation is expected to increase the error as the output and filter size increase.

5.1 Limitation

The main limitation of our design is the ability to use the same engine to perform different convolutions or use different kernels. In our design, $GgG^T$was calculated offline and uploaded to the hardware. If GgG^Tneeds to be calculated on the flight, our design is significantly limited by the amount of system memory access allowed by the in each clock cycle and other architectures like systolic arrays need to be considered.

5.2 Future work

One of the difficulties in designing CNN for applications like TinyML is the limited data storage and computation resources[5]. To reduce the number of multiplications, pruning is a good approach. It Has been demonstrated that 90% of sparsity can be achieved without losing accuracy [6]. A Winograd sparse convolution may be the next step to optimize our design. If a more flexible convolutions engine is needed, changing our Winograd convolution for a systolic array architecture is the best alternative to maintain system performance.

6. References

[1] A. Lavin and S. Gray, “Fast Algorithms for Convolutional Neural Networks,” *ArXiv150909308 Cs*, Nov. 2015, Accessed: Nov. 20, 2021. [Online]. Available: http://arxiv.org/abs/1509.09308

[2] B. Barabasz, “Quantaized Winograd/Toom-Cook Convolution for DNNs: Beyond Canonical Polynomials Base,” *ArXiv200411077 Cs Math Stat*, Apr. 2020, Accessed: Nov. 20, 2021. [Online]. Available: http://arxiv.org/abs/2004.11077

[3] X. Liu, Y. Chen, C. Hao, A. Dhar, and D. Chen, “WinoCNN: Kernel Sharing Winograd Systolic Array for Efficient Convolutional Neural Network Acceleration on FPGAs,” *ArXiv210704244 Cs*, Jul. 2021, Accessed: Dec. 07, 2021. [Online]. Available: http://arxiv.org/abs/2107.04244

[4] J. Lin, W.-M. Chen, H. Cai, C. Gan, and S. Han, “MCUNetV2: Memory-Efficient Patch-based Inference for Tiny Deep Learning,” *ArXiv211015352 Cs*, Oct. 2021, Accessed: Dec. 10, 2021. [Online]. Available: http://arxiv.org/abs/2110.15352

[5] G. Li, Z. Jia, X. Feng, and Y. Wang, “LoWino: Towards Efficient Low-Precision Winograd Convolutions on Modern CPUs,” in *50th International Conference on Parallel Processing*, Lemont IL USA, Aug. 2021, pp. 1–11. doi: 10.1145/3472456.3472464.

[6] L. Lu and Y. Liang, “SpWA: An Efficient Sparse Winograd Convolutional Neural Networks Accelerator on FPGAs,” in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, Jun. 2018, pp. 1–6. doi: 10.1109/DAC.2018.8465842.

[7] tinyML, *tinyMLSummit 2021 Qualcomm Tutorial: Advanced network quantization and compression through the AIMET*, (Apr. 06, 2021). Accessed: Nov. 21, 2021. [Online Video]. Available: https://www.youtube.com/watch?v=u1C-ZesHCII

[8] “tinyMLSummit2021d1_Tutorial_Patel.pdf.” Accessed: Nov. 21, 2021. [Online]. Available: https://cms.tinyml.org/wp-content/uploads/summit2021/tinyMLSummit2021d1_Tutorial_Patel.pdf

[9] Y. Huang, J. Shen, Z. Wang, M. Wen, and C. Zhang, “A High-efficiency FPGA-based Accelerator for Convolutional Neural Networks using Winograd Algorithm,” *J. Phys. Conf. Ser.*, vol. 1026, p. 012019, May 2018, doi: 10.1088/1742-6596/1026/1/012019.