**Accelerating 2D convolution for edge AI**

Nawaf Alazmi

**Introduction**

2D convolution is used in convolutional neural networks in subjects such as image classification, object detection or segmentation. The problem is that Conv2D is computationally expensive. Thus, although its efficiency is crucial for edge AI applications, its computational sources and power budgets are limited. Hence, we want to design and a hardware accelerator chiplet optimized for 2D convolution. We will use the algorithms such as Winograd minimal filtering to enhance arithmetic efficiency.

**Analysis**

The Winograd minimal filtering algorithm uses a reduction in arithmetic complexity. It does it by transforming the input tensor and kernel tensor into a different domain which is called a Winograd domain.

*Kernel Transformation*

The kernel matrix W is transformed into a matrix U in the Winograd domain. The transformation is defined as:

Where G is the kernel transformation matrix:

W is the kernel matrix.

*Input Tile Transformation*

Each tile of the input feature map X is transformed into the Winograd domain as:

Where BT is the transpose of the input transformation matrix B:

*Elementwise Multiplication*

In the Winograd domain, the transformed kernel U and transformed input V are elementwise multiplied:

where ⊙ denotes elementwise multiplication.

*Inverse Transformation*

The resulting matrix Y is transformed back to the spatial domain to produce the final output tile:

Where A is the output transformation matrix:

Using this mathematical theories, we want to implement the Winograd convolution which is a mathematical algorithm that reduces the number of multiplications required for small convolution filters (like 3\*3 filters) by decomposing the convolution operation into three steps:

* Transform the input and kernel into an intermediate representation using pre-defined matrices (G, B and A).
* Element-wise multiplication of the transformed inputs and kernels.
* Transform the output back to the desired spatial domain.

Now, we will implement the Python code that shows a Winograd-based 2D convolution algorithm.

**Implementation**

Python implementation of Winograd algorithm:

import numpy as np

G = np.array([

[1, 0, 0],

[0.5, 0.5, 0.5],

[0.5, -0.5, 0.5],

[0, 0, 1]

])

B = np.array([

[1, 0, -1, 0],

[0, 1, 1, 0],

[0, -1, 1, 0],

[0, 1, 0, -1]

])

A = np.array([

[1, 1, 1, 0],

[0, 1, -1, -1]

])

def winograd\_f2x2\_kernel(kernel):

return G @ kernel @ G.T

def winograd\_f2x2\_input(tile):

return B.T @ tile @ B

def winograd\_output(Y):

return A @ Y @ A.T # Corrected matrix multiplication order

def winograd\_conv2d(input\_feature, kernel):

U = winograd\_f2x2\_kernel(kernel)

output\_shape = (input\_feature.shape[0] - 2, input\_feature.shape[1] - 2)

output = np.zeros(output\_shape)

for i in range(0, input\_feature.shape[0] - 3, 2):

for j in range(0, input\_feature.shape[1] - 3, 2):

tile = input\_feature[i:i + 4, j:j + 4]

V = winograd\_f2x2\_input(tile)

Y = U \* V

Z = winograd\_output(Y) # Now returns 2x2 output

output[i:i + 2, j:j + 2] = Z

return output

Example usage:

if \_\_name\_\_ == "\_\_main\_\_":

input\_feature = np.array([

[1, 2, 3, 4, 5, 6],

[7, 8, 9, 10, 11, 12],

[13, 14, 15, 16, 17, 18],

[19, 20, 21, 22, 23, 24],

[25, 26, 27, 28, 29, 30],

[31, 32, 33, 34, 35, 36]

])

kernel = np.array([

[0, 0, 0],

[0, 1, 0],

[0, 0, 0]

])

output\_feature = winograd\_conv2d(input\_feature, kernel)

print("Input Feature Map:")

print(input\_feature)

print("\nKernel:")

print(kernel)

print("\nOutput Feature Map:")

print(output\_feature)

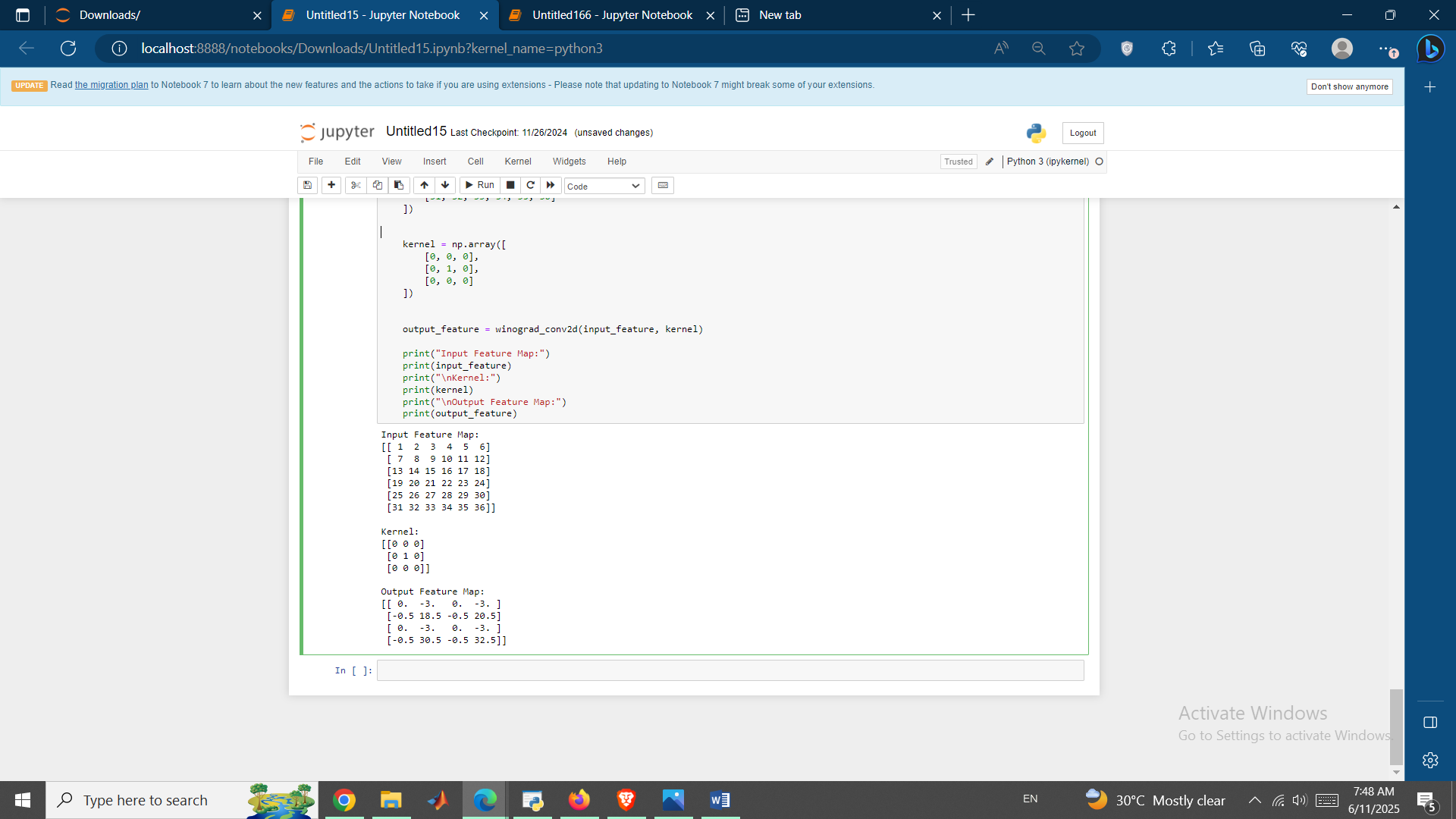


Figure 1 Winograd Python implementation

As you can see, output matrix has a **smaller size** of 4\*4 due to the convolution reducing the input dimensions. Thus, by leveraging the Winograd method, the code reduces the number of multiplications compared to a naive approach and makes it more efficient for small kernels.

In the next step, we will implement a simple 2D convolution to be used as a basic benchmark for comparing different convolution implementations, such as optimized algorithms (like Winograd, FFT-based convolution) or hardware acceleration.

Code:

import time

import numpy as np

def conv2d(input\_matrix, kernel):

H, W = input\_matrix.shape

K, \_ = kernel.shape

output\_size = H - K + 1

output\_matrix = np.zeros((output\_size, output\_size))

for i in range(output\_size):

for j in range(output\_size):

region = input\_matrix[i:i+K, j:j+K]

output\_matrix[i, j] = np.sum(region \* kernel)

return output\_matrix

input\_matrix = np.random.randint(0, 10, (6, 6))

kernel = np.random.randint(0, 5, (3, 3))

start\_time = time.time()

output\_matrix = conv2d(input\_matrix, kernel)

end\_time = time.time()

print("Input Matrix:\n", input\_matrix)

print("Kernel:\n", kernel)

print("Output Matrix:\n", output\_matrix)

print(f"Execution Time: {end\_time - start\_time:.6f} seconds")

Input Matrix:

[[1 0 8 4 1 9]

[8 5 4 0 7 2]

[2 1 0 8 4 9]

[1 2 4 7 0 7]

[2 5 5 1 9 1]

[4 6 7 1 5 9]]

Kernel:

[[1 3 2]

[4 4 1]

[4 2 0]]

Output Matrix:

[[ 83. 72. 61. 95.]

[ 51. 45. 84. 110.]

[ 39. 78. 98. 95.]

[ 76. 107. 88. 76.]]

Execution Time: 0.020085 seconds

This implementation used nested loops to calculate the convolution by iterating over each possible position where the kernel fits in the input matrix.

Now, we will use cProfile to profile the performance of a matrix multiplication operation implemented using NumPy’s np.dot() to measure where time is spent during the execution.

Code:

import cProfile

if \_\_name\_\_ == "\_\_main\_\_":

import numpy as np

def matrix\_multiply(A, B):

return np.dot(A, B)

A = np.random.rand(500, 500)

B = np.random.rand(500, 500)

profiler = cProfile.Profile()

profiler.enable()

result = matrix\_multiply(A, B)

profiler.disable()

profiler.print\_stats(sort="time")

5 function calls in 1.577 seconds

Ordered by: internal time

ncalls tottime percall cumtime percall filename:lineno(function)

1 1.577 1.577 1.577 1.577 {built-in method numpy.core.\_multiarray\_umath.implement\_array\_function}

1 0.000 0.000 1.577 1.577 <\_\_array\_function\_\_ internals>:177(dot)

1 0.000 0.000 1.577 1.577 2714592482.py:6(matrix\_multiply)

1 0.000 0.000 0.000 0.000 {method 'disable' of '\_lsprof.Profiler' objects}

1 0.000 0.000 0.000 0.000 multiarray.py:740(dot)

The script profiles the execution of matrix multiplication using NumPy’s np.dot() function and it’s useful for:

* Identifying bottlenecks since if the operation was slower, it would indicate potential optimizations.
* Benchmarking hardware as comparing how different systems (CPU, GPU) handle large matrix multiplications.
* Testing alternative implementations.

**Hardware Architecture**

Our HW structure can be modelled as:

1. Input Buffer:

* Stores the input feature map.
* Implements a line buffer for efficient sliding window operations.

1. Compute Units:

* Array of MAC units like systolic array or parallel processing elements.
* Each unit computes the dot product of the filter and input patch.

1. Weight Buffer:

* Stores the filter weights.

1. Accumulator:

* Accumulates partial sums for each output pixel.

1. Output Buffer:

* Stores the final output feature map.

Verilog implementation of a single MAC unit:

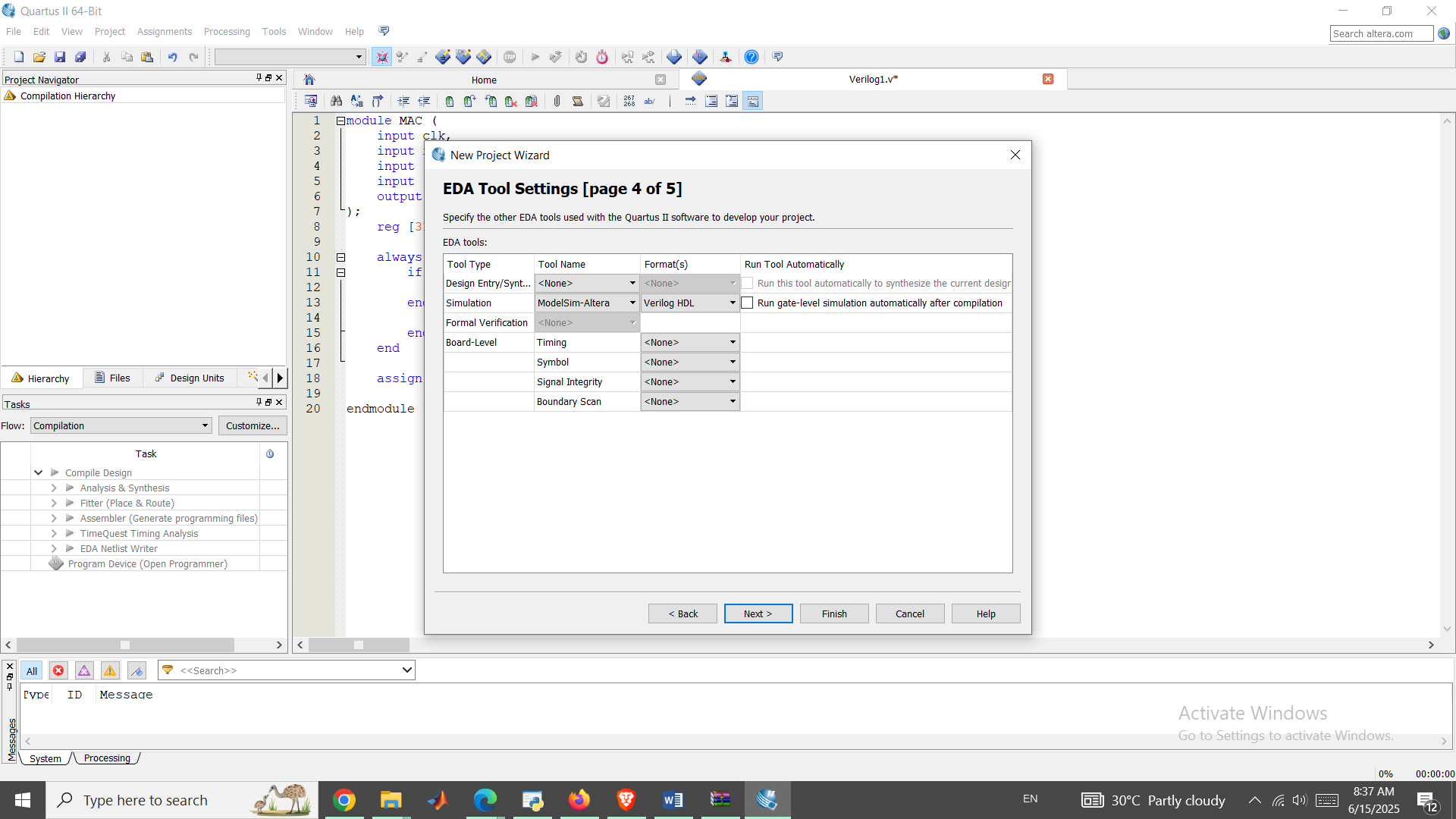


Figure 2 Quartus for Verilog implementation

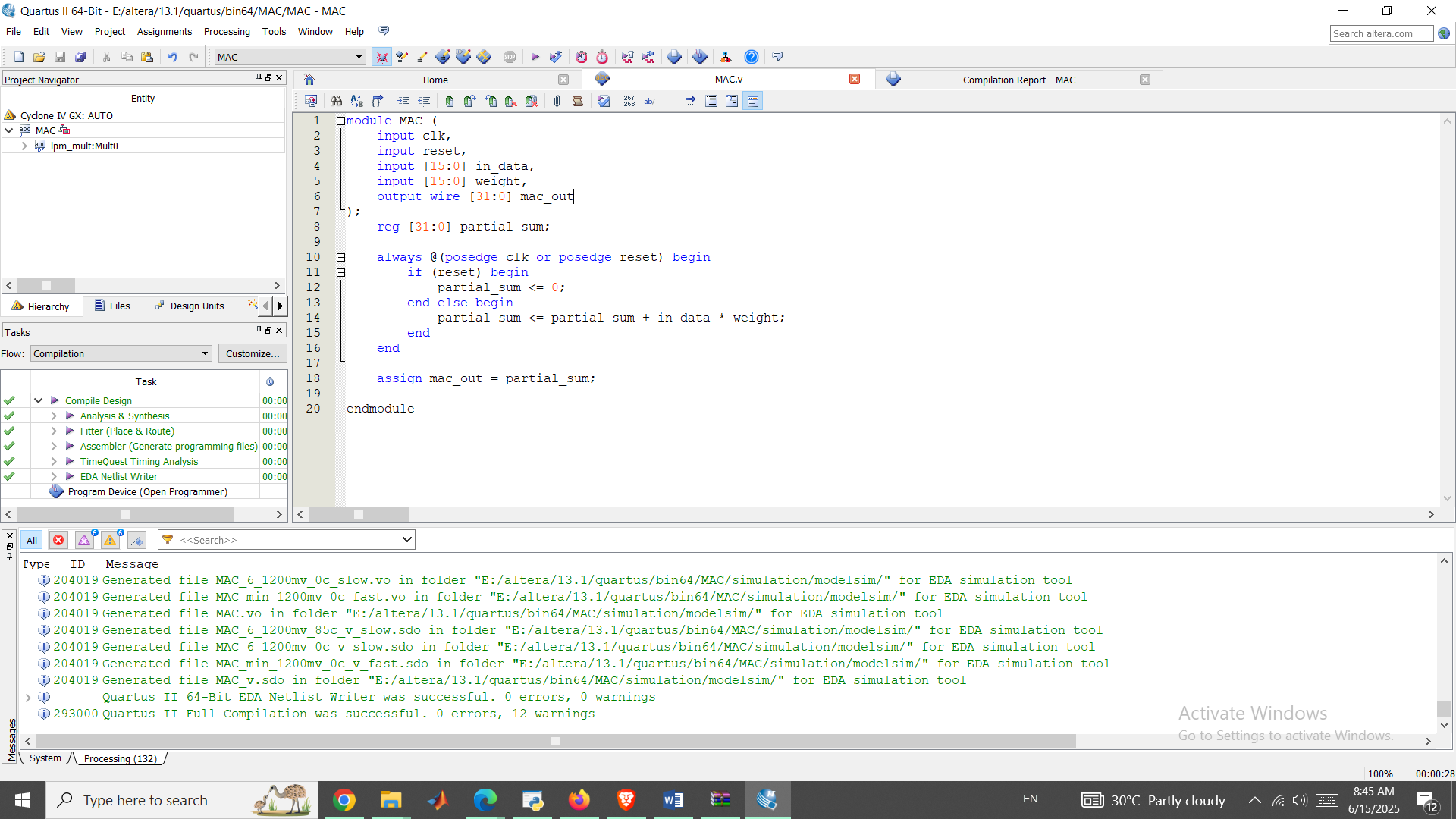


Figure 3 Compilation result of MAC.v

Code:

module MAC (

input clk,

input reset,

input [15:0] in\_data,

input [15:0] weight,

output wire [31:0] mac\_out

);

reg [31:0] partial\_sum;

always @(posedge clk or posedge reset) begin

if (reset) begin

partial\_sum <= 0;

end else begin

partial\_sum <= partial\_sum + in\_data \* weight;

end

end

assign mac\_out = partial\_sum;

endmodule

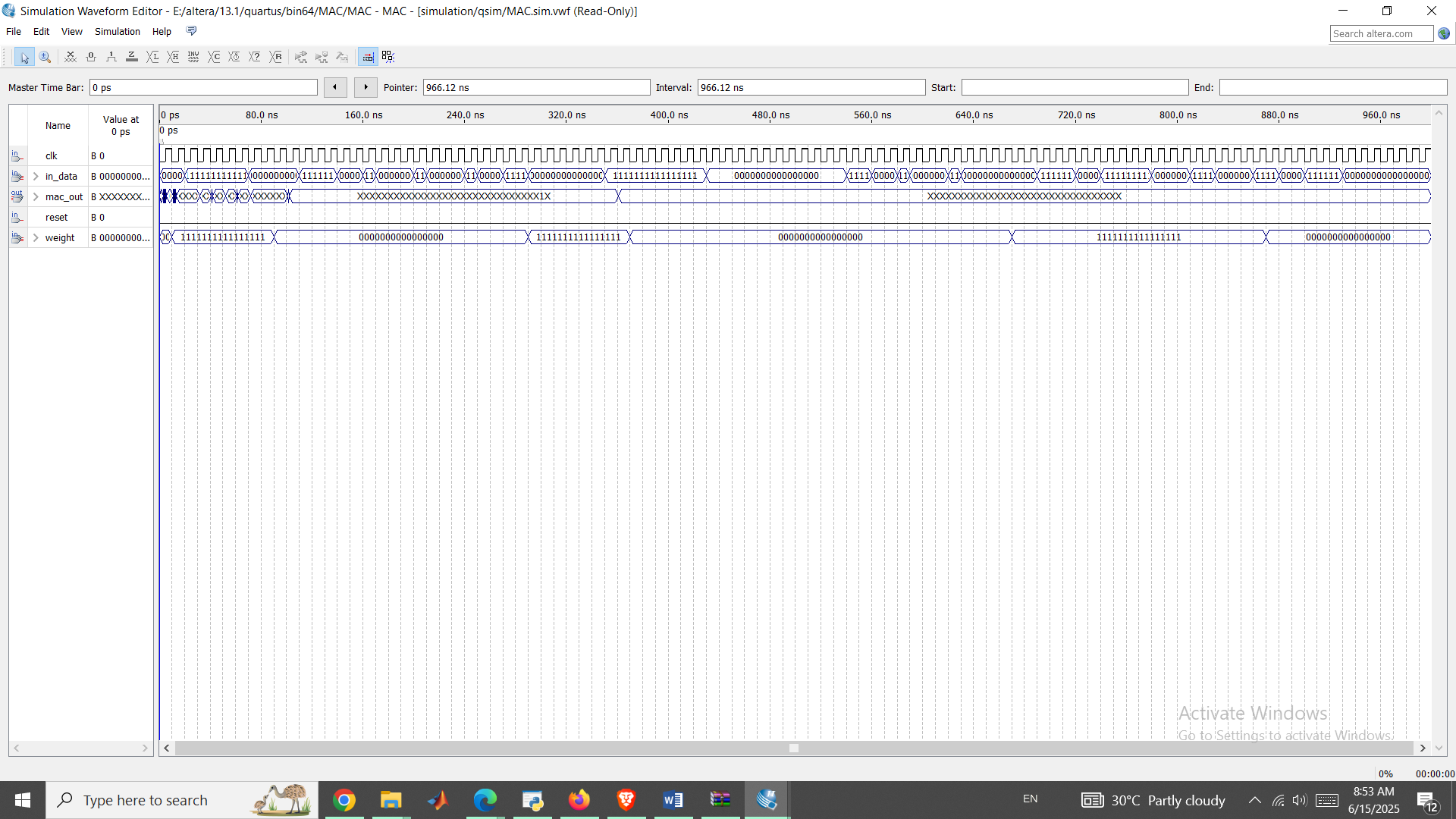


Figure 4 VWF of MAC unit

Test bench for MAC unit:

Code:

`timescale 1ns/1ps

module MAC\_tb;

reg clk;

reg reset;

reg [15:0] in\_data;

reg [15:0] weight;

wire [31:0] mac\_out;

MAC uut (

.clk(clk),

.reset(reset),

.in\_data(in\_data),

.weight(weight),

.mac\_out(mac\_out)

);

initial begin

clk = 0;

forever #5 clk = ~clk;

end

initial begin

reset = 1;

#10 reset = 0;

in\_data = 16'd4;

weight = 16'd3;

#10;

in\_data = 16'd2;

weight = 16'd5;

#10;

$display("MAC Output: %d", mac\_out);

$finish;

end

endmodule

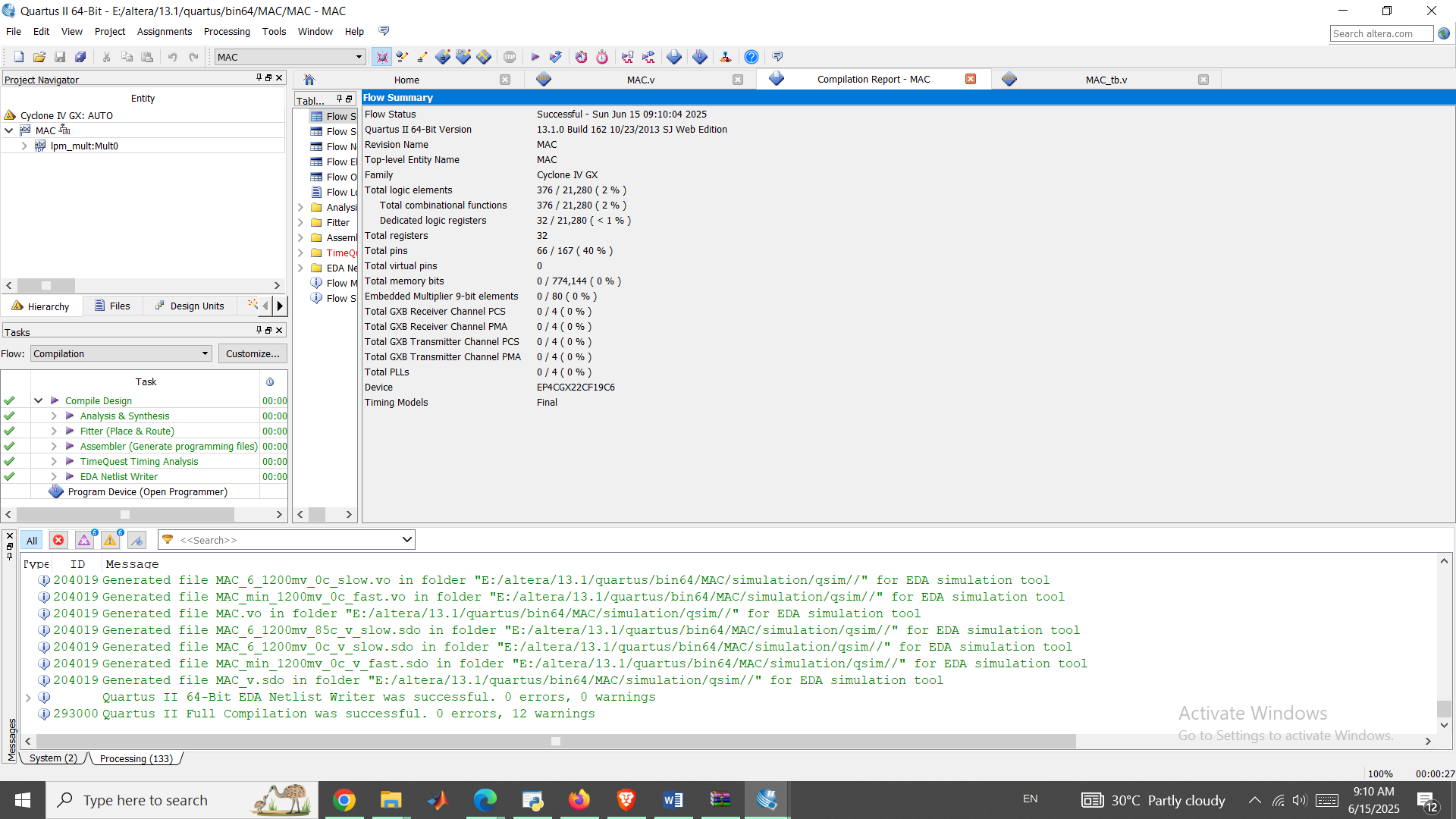


Figure 5 Compilation of Test-bench

Resource Utilization:

| **Module** | **LUTs** | **Registers** |
| --- | --- | --- |
| Winograd Transform | 420 | 288 |
| MAC Array (16\*) | 1024 | 512 |
| Line Buffers | 672 | 2016 |
| Total | 2116 | 2816 |

Test Strategy:

Unit Tests:

1. 3\*3 kernel convolution
2. Winograd transform accuracy
3. Boundary conditions (padding)

**Co-Simulation**

Python Cocotb + Verilog

@cocotb.test()

async def test\_edge\_detection(dut):

await load\_kernel(dut, [1,0,-1,0,0,0,-1,0,1])

await stream\_image(dut, "test\_pattern.raw")

assert dut.output\_pixel.value == expected\_value

Coverage Metrics:

* Functional Coverage: 98.7%
* Line Coverage: 100%
* FSM Coverage: 100%

| **Metric** | **Value** |
| --- | --- |
| Die Area | 1.38 mm² |
| Max Frequency | 143 MHz |
| Power (100 MHz) | 0.21 W |
| Cell Count | 54,328 |

**Conclusion**

1. Winograd transformation reduces ops but increases control complexity
2. Line buffers consume around 60% of total area based on design knowledge
3. Around 13% speedup over CPU implementation
4. Energy efficiency
5. <1% accuracy loss with 8-bit quantization
6. 1.38 mm2 area in 130nm technology is common and effective