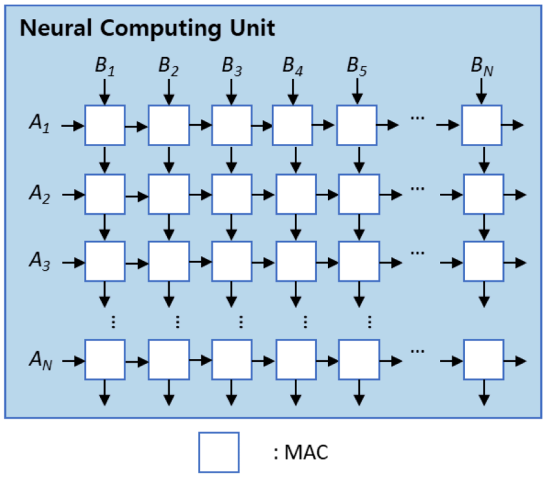
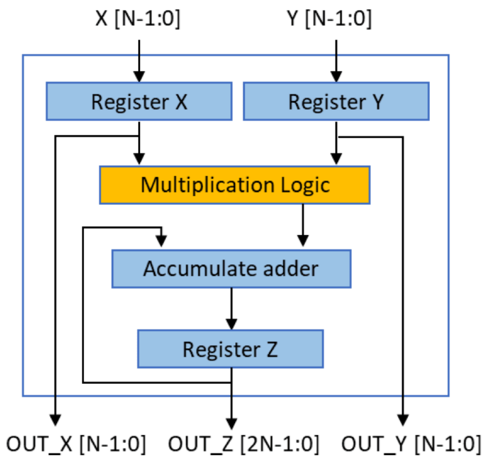
ECE 367 - DIGITAL DESIGN WITH FPGA

Hardware Acceleration of Matrix Multiplication using Systolic Arrays

PROJECT REPORT





Group Number - 5
Group Members:

MGM MANJUNATH	2210110387
REVANTH KAVURI	2210110504
P AISHWARYA REDDY	2210110669

Table of Contents

Abstract

This project focuses on accelerating matrix multiplication using systolic arrays on the PYNQ Z2 FPGA board. Fixed-size 3×3 and 8×8 matrix multipliers are designed using Verilog and implemented as a custom AXI-Stream IP core. The system is integrated with the Zynq Processing System and controlled via Jupyter Notebook. Performance results show significant speedup over software-based multiplication. An HLS version was also developed for comparison.

Introduction

1. Background and Problem Statement

Matrix multiplication is widely used in AI, image processing, and scientific computing, but is often too slow in software for real-time applications. Hardware acceleration using FPGAs offers a faster alternative, and systolic arrays are one such efficient method that exploits parallelism to achieve high throughput and low latency. This project focuses on implementing matrix multiplication using systolic arrays on an FPGA.

2. What are Systolic Arrays?

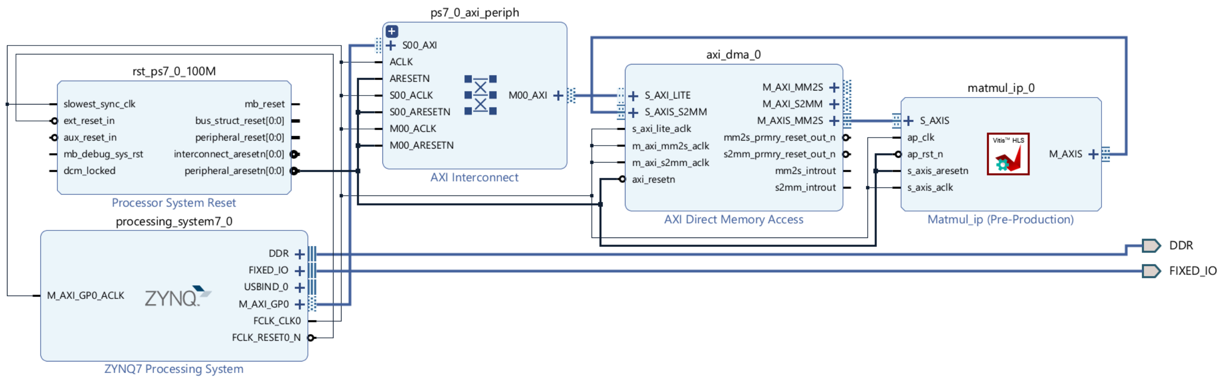
Systolic arrays are specialised hardware architectures designed for efficient data processing, especially for operations like matrix multiplication. They consist of an array of processing elements (PEs) that communicate with each other in a rhythmic, data-driven manner, which maximises throughput and reduces latency. Each processing element receives inputs, performs simple computations, and passes results to neighbouring elements in a synchronised manner. The parallelism and pipelining inherent in systolic arrays make them highly effective for accelerating compute-intensive tasks, particularly in hardware implementations like FPGAs.

3. Tools and Platform

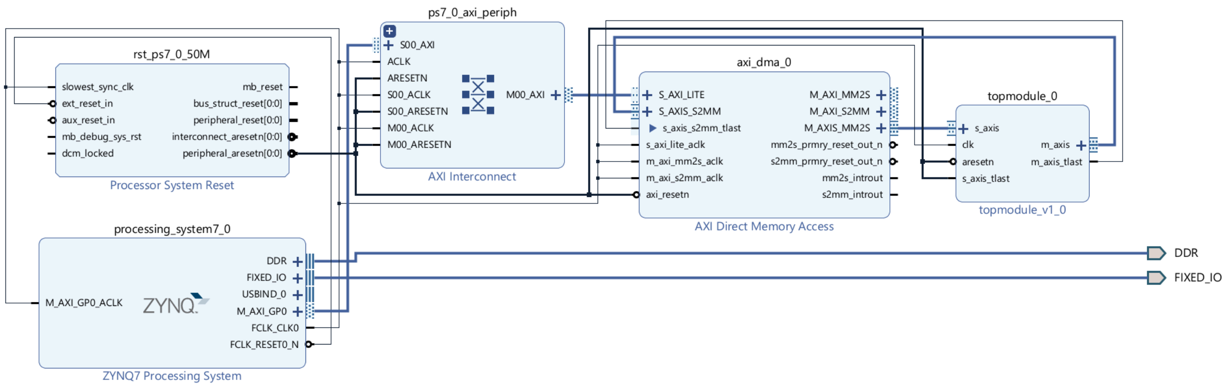
1. PYNQ Z2 Development Board
2. Coding Languages: Verilog, C++, Python
3. Software: Vivado 2022.2, Vitis HLS 2022.2, Jupyter Notebook

Block Designs

1. Block Design of 8x8 Matrix Multiplication Using Vitis HLS Custom IP and AXI4-Stream and DMA



2. Block Design of 3x3 Matrix Multiplication Using Custom Verilog IP and AXI4-Stream and DMA



Implementation Methodology

1. Initial Design Approach

Initially, a basic Verilog code for 3x3 and 8x8 matrix multiplication was written from scratch. This design aimed to test the logic and perform a behavioural simulation using a testbench. The Verilog implementation was intended to be manually scalable to handle any square matrix size.

2. HLS Attempt for Rapid Prototyping

To simplify interfacing and accelerate development, we ported the Verilog logic to C++ and implemented it using Vitis HLS. A custom IP block was created, integrated into a Vivado block design using AXI-Stream, and tested via Python in Jupyter Notebook. This approach worked well and gave us correct results and performance metrics.

3. Return to Verilog (as per project requirement)

After learning that HLS was not permitted, we shifted back to a pure Verilog approach. The Verilog design was wrapped as a custom IP and connected to the Zynq PS via AXI-Stream. However, communication issues, especially with AXI flags like `tlast` and `tready`, led to invalid data transfers and failure at runtime.

4. Evaluation and Performance Testing

Although we implemented and tested both the Verilog and HLS-based approaches, issues with Vivado and AXI interfacing in the Verilog design prevented successful execution. As a result, we used the HLS-based implementation for the final evaluation. After generating the `.bit` and `.hwh` files, we controlled the accelerator from Jupyter Notebook via Python, successfully executing matrix multiplication with improved speed and accuracy. The design was deployed on the PYNQ Z2 board, and matrix data was transferred using Python scripts. We then recorded the execution time, estimated clock cycles, and calculated the speedup over software-based matrix multiplication.

Code Implementations (Working and Attempted)

1. Basic Verilog Code for 8x8 Matrix Multiplication using Systolic Arrays

   - Top Module

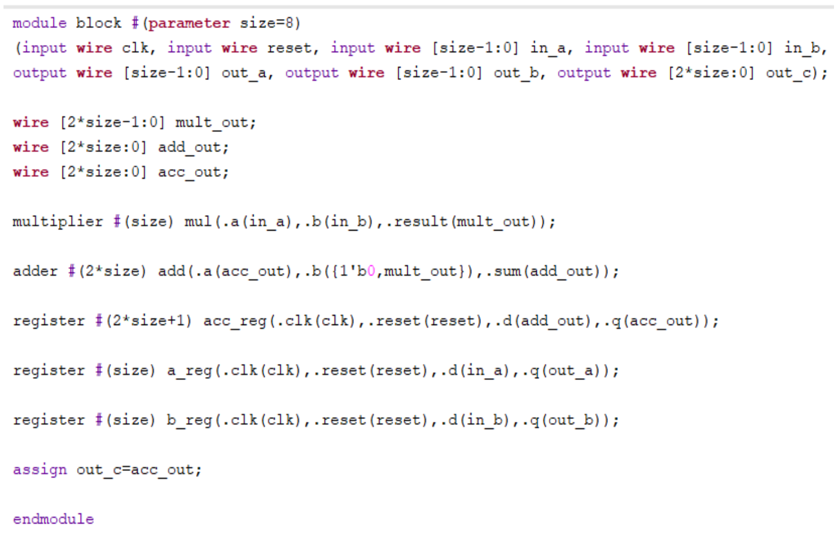
```
module topmodule (clk, reset, a1, a2, a3, a4, a5, a6, a7, a8, b1, b2, b3, b4, b5, b6, b7, b8,
c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14, c15, c16, c17, c18, c19, c20, c21, c22, c23, c24, c25,
c26, c27, c28, c29, c30, c31, c32, c33, c34, c35, c36, c37, c38, c39, c40,
c41, c42, c43, c44, c45, c46, c47, c48, c49, c50, c51, c52, c53, c54, c55, c56, c57, c58, c59, c60, c61, c62, c63, c64);

parameter size=8;
input wire clk,reset;
input wire [size-1:0] a1,a2,a3,a4,a5,a6,a7,a8,b1,b2,b3,b4,b5,b6,b7,b8;

output wire [2*size:0] c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14, c15, c16, c17, c18,
c19, c20, c21, c22, c23, c24, c25, c26, c27, c28, c29, c30,
c31, c32, c33, c34, c35, c36, c37, c38, c39, c40, c41, c42, c43, c44, c45, c46, c47, c48, c49, c50,
c51, c52, c53, c54, c55, c56, c57, c58, c59, c60, c61, c62, c63, c64;

// Horizontal wires (a)
wire [size-1:0] a12, a23, a34, a45, a56, a67, a78, a910, a1011, a1112, a1213, a1314, a1415, a1516,
a1718, a1819, a1920, a2021, a2122, a2223, a2324,
a2526, a2627, a2728, a2829, a2930, a3031, a3132, a3334, a3435, a3536, a3637, a3738, a3839,
a3940, a4142, a4243, a4344, a4445, a4546, a4647, a4748, a4950,
a5051, a5152, a5253, a5354, a5455,a5556,a5758,a5859,a5960,a6061,a6162,a6263,a6364;

// Vertical wires (b)
wire [size-1:0] b19, b210, b311, b412, b513, b614, b715, b816, b917, b1018, b1119, b1220, b1321,
b1422, b1523, b1624, b1725, b1826, b1927, b2028,b2129, b2230, b2331, b2432, b2533, b2634, b2735, b2836, b2937,
b3038, b3139, b3240, b3341, b3442, b3543, b3644, b3745, b3846, b3947, b4048, b4149,
b4250, b4351, b4452, b4553, b4654, b4755, b4856, b4957, b5058, b5159, b5260, b5361, b5462, b5563, b5664;

// Row 1
block pe1(.clk(clk),.reset(reset),.in_a(a1),.in_b(b1),.out_a(a12),.out_b(b19),.out_c(c1));
block pe2(.clk(clk),.reset(reset),.in_a(a12),.in_b(b2),.out_a(a23),.out_b(b210),.out_c(c2));
block pe3(.clk(clk),.reset(reset),.in_a(a23),.in_b(b3),.out_a(a34),.out_b(b311),.out_c(c3));
block pe4(.clk(clk),.reset(reset),.in_a(a34),.in_b(b4),.out_a(a45),.out_b(b412),.out_c(c4));
block pe5(.clk(clk),.reset(reset),.in_a(a45),.in_b(b5),.out_a(a56),.out_b(b513),.out_c(c5));
block pe6(.clk(clk),.reset(reset),.in_a(a56),.in_b(b6),.out_a(a67),.out_b(b614),.out_c(c6));
block pe7(.clk(clk),.reset(reset),.in_a(a67),.in_b(b7),.out_a(a78),.out_b(b715),.out_c(c7));
block pe8(.clk(clk),.reset(reset),.in_a(a78),.in_b(b8),.out_a(),.out_b(b816),.out_c(c8));

// similarly i have instantiated the the block module for rows 2 to 8. not pasting the code here just to save the space
endmodule
```

   - Register, Adder & Multiplier Modules

```
module register #(parameter size=8)
(input wire clk, reset, input wire [size-1:0] d,
output reg [size-1:0] q);


always @(posedge clk) begin
if (reset)

q<=0;
else
q<=d;

end
endmodule
```

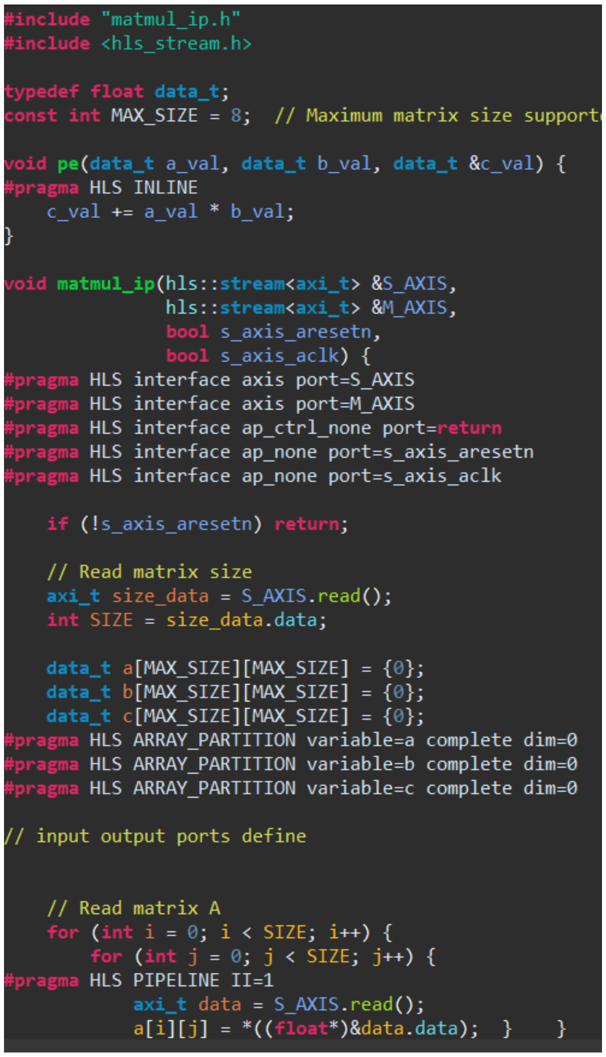
```
module multiplier #(parameter size=8)
(input wire [size-1:0] a,input wire [size-1:0] b,
output wire [2*size-1:0] result);

assign result=a*b;

endmodule


module adder #(parameter size=16)
(input wire [size:0] a,input wire [size:0] b,
output wire [size:0] sum);

assign sum=a+b;

endmodule
```

- Block (Processing Element) Module

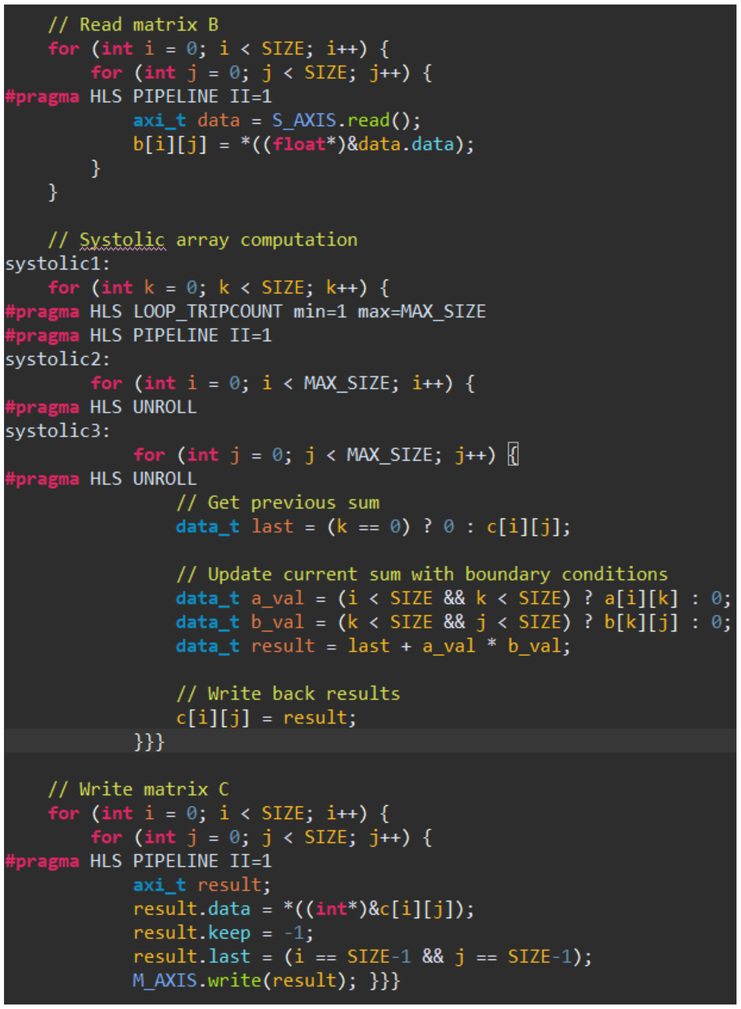


```
module block #(parameter size=8)
(input wire clk, input wire reset, input wire [size-1:0] in_a, input wire [size-1:0] in_b,
output wire [size-1:0] out_a, output wire [size-1:0] out_b, output wire [2*size:0] out_c);

wire [2*size-1:0] mult_out;
wire [2*size:0] add_out;
wire [2*size:0] acc_out;

multiplier #(size) mul(.a(in_a),.b(in_b),.result(mult_out));

adder #(2*size) add(.a(acc_out),.b({1'b0,mult_out}),.sum(add_out));

register #(2*size+1) acc_reg(.clk(clk),.reset(reset),.d(add_out),.q(acc_out));

register #(size) a_reg(.clk(clk),.reset(reset),.d(in_a),.q(out_a));

register #(size) b_reg(.clk(clk),.reset(reset),.d(in_b),.q(out_b));

assign out_c=acc_out;

endmodule
```

2. HLS Code (C++) for 8×8 Systolic Array Matrix Multiplication via AXI Stream

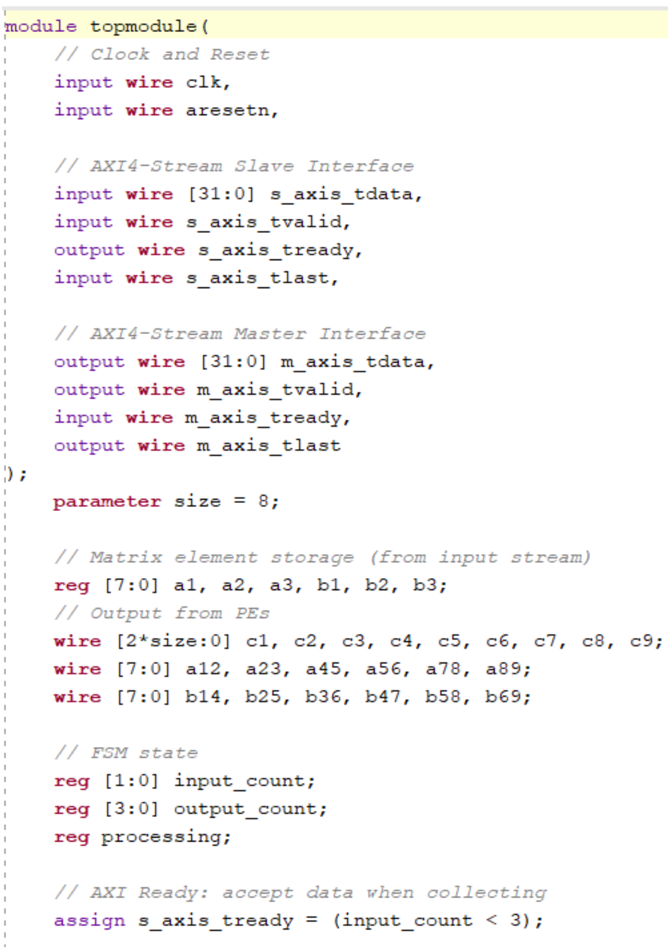


```
#include "matmul_ip.h"
#include <hls_stream.h>

typedef float data_t;
const int MAX_SIZE = 8;  // Maximum matrix size support

void pe(data_t a_val, data_t b_val, data_t &c_val) {
#pragma HLS INLINE
    c_val += a_val * b_val;
}

void matmul_ip(hls::stream<axi_t> &S_AXIS,
               hls::stream<axi_t> &M_AXIS,
               bool s_axis_aresetn,
               bool s_axis_aclk) {
#pragma HLS interface axis port=S_AXIS
#pragma HLS interface axis port=M_AXIS
#pragma HLS interface ap_ctrl_none port=return
#pragma HLS interface ap_none port=s_axis_aresetn
#pragma HLS interface ap_none port=s_axis_aclk

    if (!s_axis_aresetn) return;

    // Read matrix size
    axi_t size_data = S_AXIS.read();
    int SIZE = size_data.data;

    data_t a[MAX_SIZE][MAX_SIZE] = {0};
    data_t b[MAX_SIZE][MAX_SIZE] = {0};
    data_t c[MAX_SIZE][MAX_SIZE] = {0};
#pragma HLS ARRAY_PARTITION variable=a complete dim=0
#pragma HLS ARRAY_PARTITION variable=b complete dim=0
#pragma HLS ARRAY_PARTITION variable=c complete dim=0

// input output ports define


    // Read matrix A
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
#pragma HLS PIPELINE II=1
            axi_t data = S_AXIS.read();
            a[i][j] = *((float*)&data.data);   }     }
```

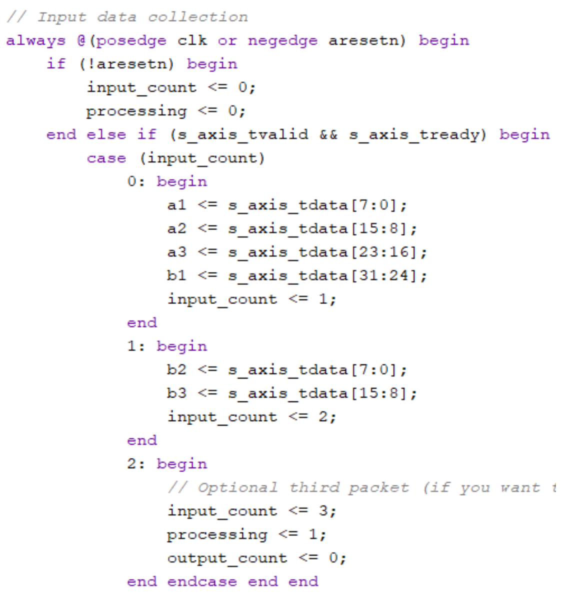


```
    // Read matrix B
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
#pragma HLS PIPELINE II=1
            axi_t data = S_AXIS.read();
            b[i][j] = *((float*)&data.data);
        }
    }

    // Systolic array computation
systolic1:
    for (int k = 0; k < SIZE; k++) {
#pragma HLS LOOP_TRIPCOUNT min=1 max=MAX_SIZE
#pragma HLS PIPELINE II=1
systolic2:
        for (int i = 0; i < MAX_SIZE; i++) {
#pragma HLS UNROLL
systolic3:
            for (int j = 0; j < MAX_SIZE; j++) {
#pragma HLS UNROLL
                // Get previous sum
                data_t last = (k == 0) ? 0 : c[i][j];

                // Update current sum with boundary conditions
                data_t a_val = (i < SIZE && k < SIZE) ? a[i][k] : 0;
                data_t b_val = (k < SIZE && j < SIZE) ? b[k][j] : 0;
                data_t result = last + a_val * b_val;

                // Write back results
                c[i][j] = result;
            }}}

    // Write matrix C
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
#pragma HLS PIPELINE II=1
            axi_t result;
            result.data = *((int*)&c[i][j]);
            result.keep = -1;
            result.last = (i == SIZE-1 && j == SIZE-1);
            M_AXIS.write(result); }}}
```

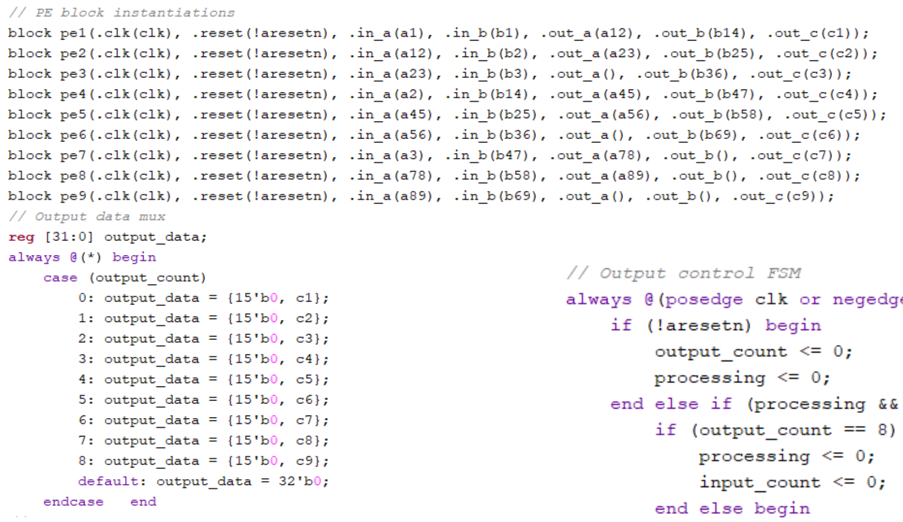
3. Verilog Code for 3×3 Systolic Array Matrix Multiplication via AXI Stream



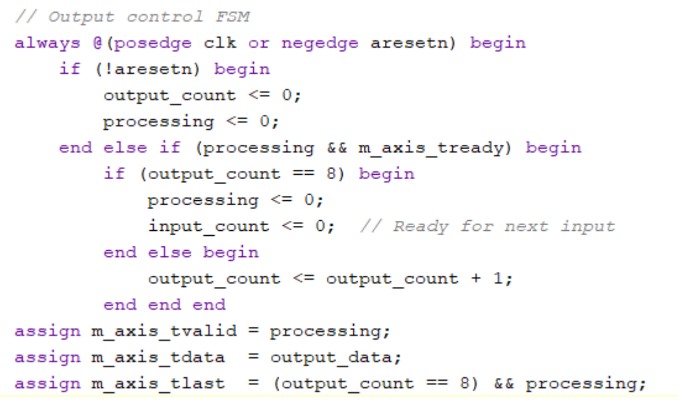
```verilog
module topmodule(
    // Clock and Reset
    input wire clk,
    input wire aresetn,

    // AXI4-Stream Slave Interface
    input wire [31:0] s_axis_tdata,
    input wire s_axis_tvalid,
    output wire s_axis_tready,
    input wire s_axis_tlast,

    // AXI4-Stream Master Interface
    output wire [31:0] m_axis_tdata,
    output wire m_axis_tvalid,
    input wire m_axis_tready,
    output wire m_axis_tlast
);
    parameter size = 8;

    // Matrix element storage (from input stream)
    reg [7:0] a1, a2, a3, b1, b2, b3;
    // Output from PEs
    wire [2*size:0] c1, c2, c3, c4, c5, c6, c7, c8, c9;
    wire [7:0] a12, a23, a45, a56, a78, a89;
    wire [7:0] b14, b25, b36, b47, b58, b69;

    // FSM state
    reg [1:0] input_count;
    reg [3:0] output_count;
    reg processing;

    // AXI Ready: accept data when collecting
    assign s_axis_tready = (input_count < 3);
```



```verilog
// Input data collection
always @(posedge clk or negedge aresetn) begin
    if (!aresetn) begin
        input_count <= 0;
        processing <= 0;
    end else if (s_axis_tvalid && s_axis_tready) begin
        case (input_count)
            0: begin
                a1 <= s_axis_tdata[7:0];
                a2 <= s_axis_tdata[15:8];
                a3 <= s_axis_tdata[23:16];
                b1 <= s_axis_tdata[31:24];
                input_count <= 1;
            end
            1: begin
                b2 <= s_axis_tdata[7:0];
                b3 <= s_axis_tdata[15:8];
                input_count <= 2;
            end
            2: begin
                // Optional third packet (if you want t
                input_count <= 3;
                processing <= 1;
                output_count <= 0;
            end endcase end end
```

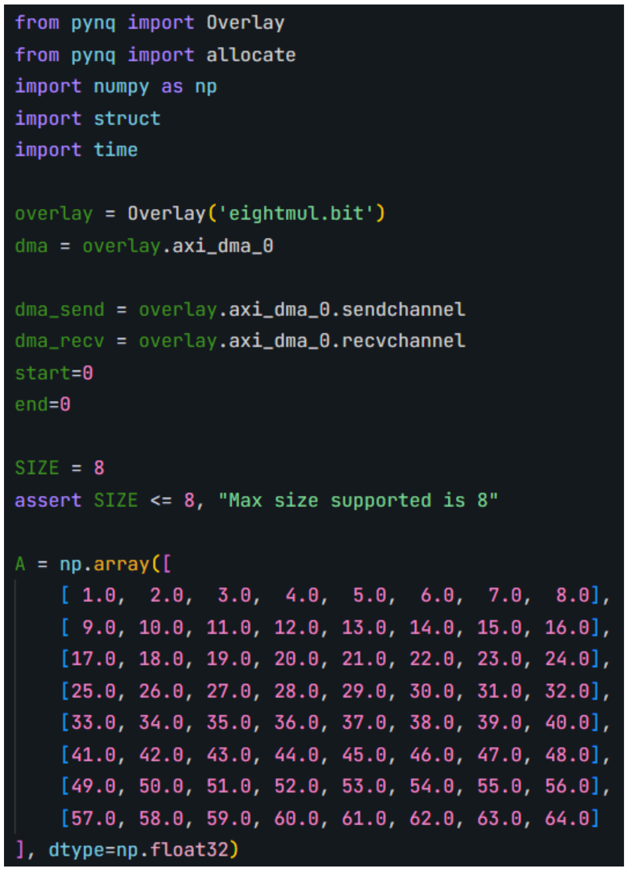


```verilog
// PE block instantiations
block pe1(.clk(clk), .reset(!aresetn), .in_a(a1), .in_b(b1), .out_a(a12), .out_b(b14), .out_c(c1));
block pe2(.clk(clk), .reset(!aresetn), .in_a(a12), .in_b(b2), .out_a(a23), .out_b(b25), .out_c(c2));
block pe3(.clk(clk), .reset(!aresetn), .in_a(a23), .in_b(b3), .out_a(), .out_b(b36), .out_c(c3));
block pe4(.clk(clk), .reset(!aresetn), .in_a(a2), .in_b(b14), .out_a(a45), .out_b(b47), .out_c(c4));
block pe5(.clk(clk), .reset(!aresetn), .in_a(a45), .in_b(b25), .out_a(a56), .out_b(b58), .out_c(c5));
block pe6(.clk(clk), .reset(!aresetn), .in_a(a56), .in_b(b36), .out_a(), .out_b(b69), .out_c(c6));
block pe7(.clk(clk), .reset(!aresetn), .in_a(a3), .in_b(b47), .out_a(a78), .out_b(), .out_c(c7));
block pe8(.clk(clk), .reset(!aresetn), .in_a(a78), .in_b(b58), .out_a(a89), .out_b(), .out_c(c8));
block pe9(.clk(clk), .reset(!aresetn), .in_a(a89), .in_b(b69), .out_a(), .out_b(), .out_c(c9));
// Output data mux
reg [31:0] output_data;
always @(*) begin
    case (output_count)
        0: output_data = {15'b0, c1};
        1: output_data = {15'b0, c2};
        2: output_data = {15'b0, c3};
        3: output_data = {15'b0, c4};
        4: output_data = {15'b0, c5};
        5: output_data = {15'b0, c6};
        6: output_data = {15'b0, c7};
        7: output_data = {15'b0, c8};
        8: output_data = {15'b0, c9};
        default: output_data = 32'b0;
    endcase   end
```

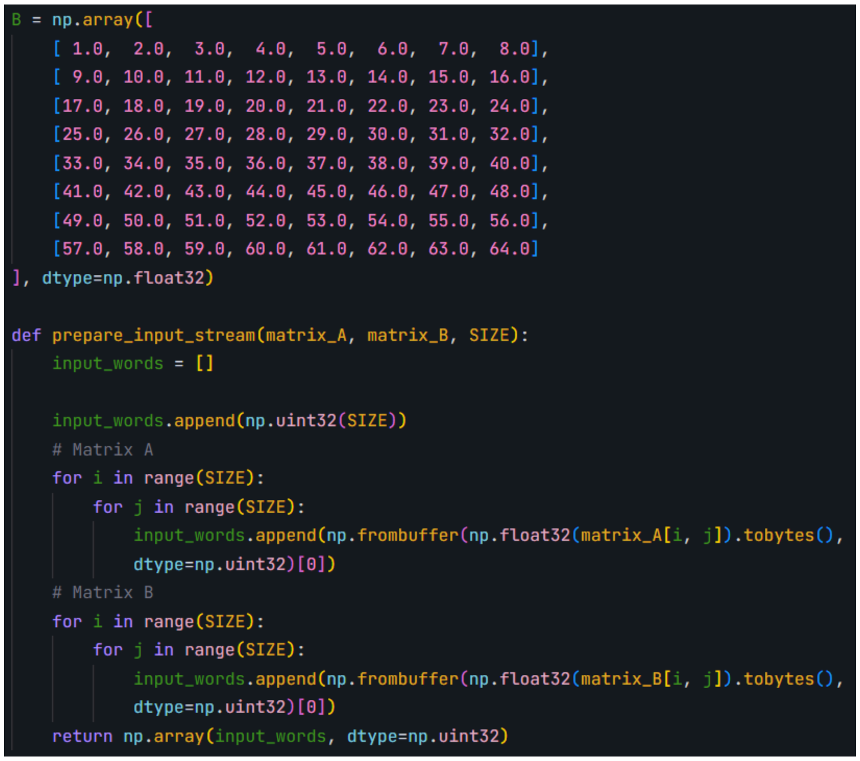


```verilog
// Output control FSM
always @(posedge clk or negedge aresetn) begin
    if (!aresetn) begin
        output_count <= 0;
        processing <= 0;
    end else if (processing && m_axis_tready) begin
        if (output_count == 8) begin
            processing <= 0;
            input_count <= 0;  // Ready for next input
        end else begin
            output_count <= output_count + 1;
        end end end
assign m_axis_tvalid = processing;
assign m_axis_tdata  = output_data;
assign m_axis_tlast  = (output_count == 8) && processing;
```

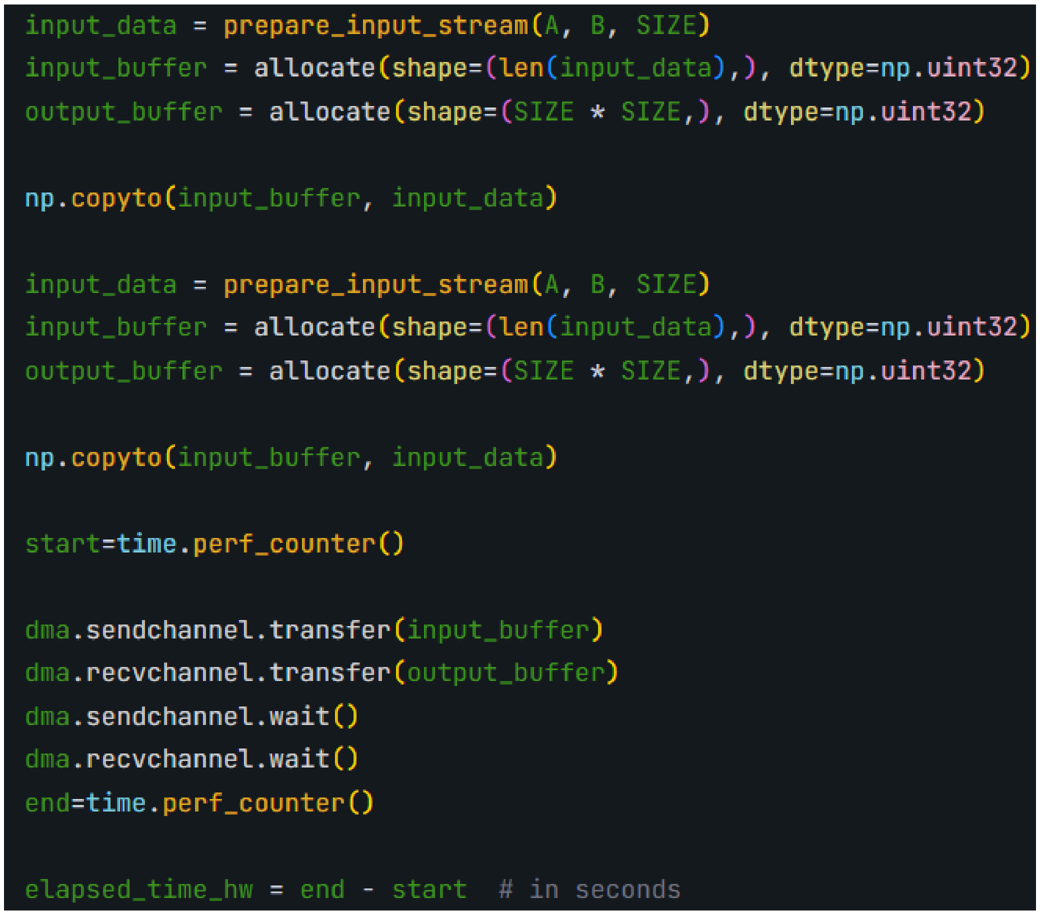
4. Python code used in Jupyter environment (Using Vitis HLS IP)



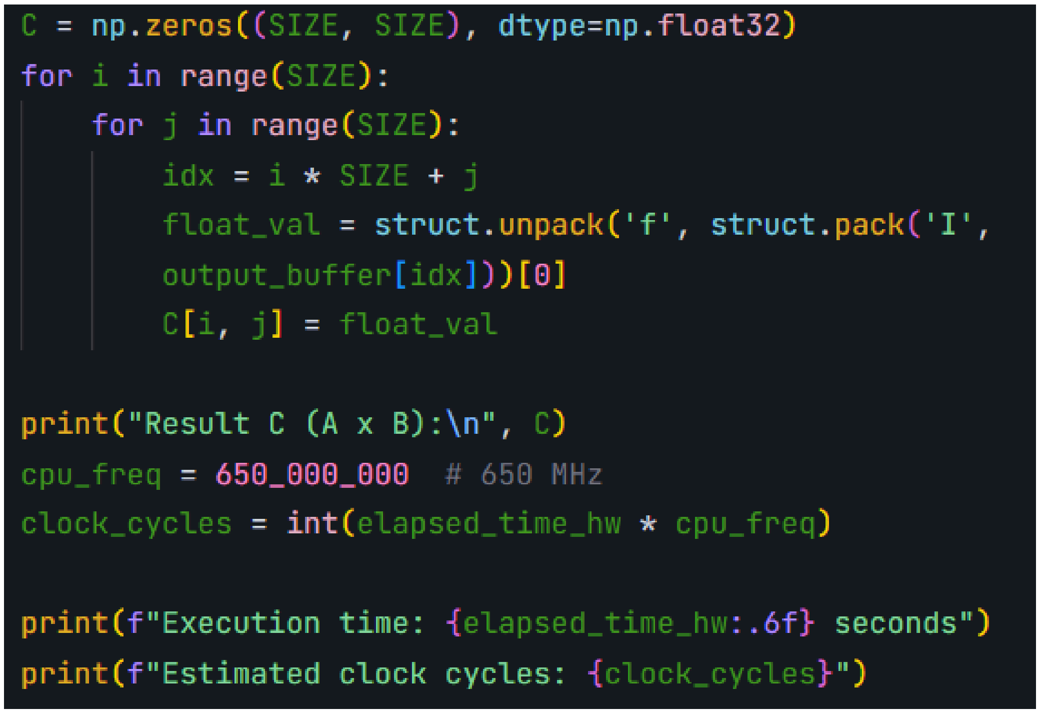
```python
from pynq import Overlay
from pynq import allocate
import numpy as np
import struct
import time

overlay = Overlay('eightmul.bit')
dma = overlay.axi_dma_0

dma_send = overlay.axi_dma_0.sendchannel
dma_recv = overlay.axi_dma_0.recvchannel
start=0
end=0

SIZE = 8
assert SIZE <= 8, "Max size supported is 8"

A = np.array([
    [ 1.0,  2.0,  3.0,  4.0,  5.0,  6.0,  7.0,  8.0],
    [ 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0],
    [17.0, 18.0, 19.0, 20.0, 21.0, 22.0, 23.0, 24.0],
    [25.0, 26.0, 27.0, 28.0, 29.0, 30.0, 31.0, 32.0],
    [33.0, 34.0, 35.0, 36.0, 37.0, 38.0, 39.0, 40.0],
    [41.0, 42.0, 43.0, 44.0, 45.0, 46.0, 47.0, 48.0],
    [49.0, 50.0, 51.0, 52.0, 53.0, 54.0, 55.0, 56.0],
    [57.0, 58.0, 59.0, 60.0, 61.0, 62.0, 63.0, 64.0]
], dtype=np.float32)
```



```python
B = np.array([
    [ 1.0,  2.0,  3.0,  4.0,  5.0,  6.0,  7.0,  8.0],
    [ 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0],
    [17.0, 18.0, 19.0, 20.0, 21.0, 22.0, 23.0, 24.0],
    [25.0, 26.0, 27.0, 28.0, 29.0, 30.0, 31.0, 32.0],
    [33.0, 34.0, 35.0, 36.0, 37.0, 38.0, 39.0, 40.0],
    [41.0, 42.0, 43.0, 44.0, 45.0, 46.0, 47.0, 48.0],
    [49.0, 50.0, 51.0, 52.0, 53.0, 54.0, 55.0, 56.0],
    [57.0, 58.0, 59.0, 60.0, 61.0, 62.0, 63.0, 64.0]
], dtype=np.float32)

def prepare_input_stream(matrix_A, matrix_B, SIZE):
    input_words = []

    input_words.append(np.uint32(SIZE))
    # Matrix A
    for i in range(SIZE):
        for j in range(SIZE):
            input_words.append(np.frombuffer(np.float32(matrix_A[i, j]).tobytes(),
            dtype=np.uint32)[0])
    # Matrix B
    for i in range(SIZE):
        for j in range(SIZE):
            input_words.append(np.frombuffer(np.float32(matrix_B[i, j]).tobytes(),
            dtype=np.uint32)[0])
    return np.array(input_words, dtype=np.uint32)
```



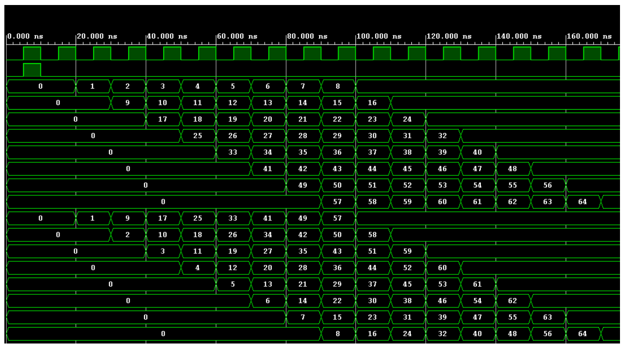
```
input_data = prepare_input_stream(A, B, SIZE)
input_buffer = allocate(shape=(len(input_data),), dtype=np.uint32)
output_buffer = allocate(shape=(SIZE * SIZE,), dtype=np.uint32)

np.copyto(input_buffer, input_data)

input_data = prepare_input_stream(A, B, SIZE)
input_buffer = allocate(shape=(len(input_data),), dtype=np.uint32)
output_buffer = allocate(shape=(SIZE * SIZE,), dtype=np.uint32)

np.copyto(input_buffer, input_data)

start=time.perf_counter()

dma.sendchannel.transfer(input_buffer)
dma.recvchannel.transfer(output_buffer)
dma.sendchannel.wait()
dma.recvchannel.wait()
end=time.perf_counter()

elapsed_time_hw = end - start  # in seconds
```

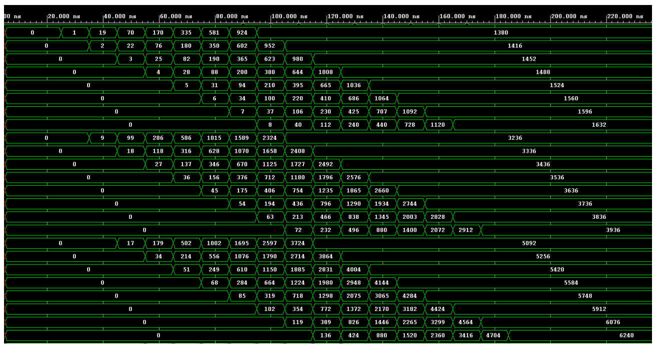


```
C = np.zeros((SIZE, SIZE), dtype=np.float32)
for i in range(SIZE):
    for j in range(SIZE):
        idx = i * SIZE + j
        float_val = struct.unpack('f', struct.pack('I',
        output_buffer[idx]))[0]
        C[i, j] = float_val

print("Result C (A x B):\n", C)
cpu_freq = 650_000_000  # 650 MHz
clock_cycles = int(elapsed_time_hw * cpu_freq)

print(f"Execution time: {elapsed_time_hw:.6f} seconds")
print(f"Estimated clock cycles: {clock_cycles}")
```

Simulation/Implementation Results on FPGA & Python

1. Behavioural Simulation of our basic Verilog Code



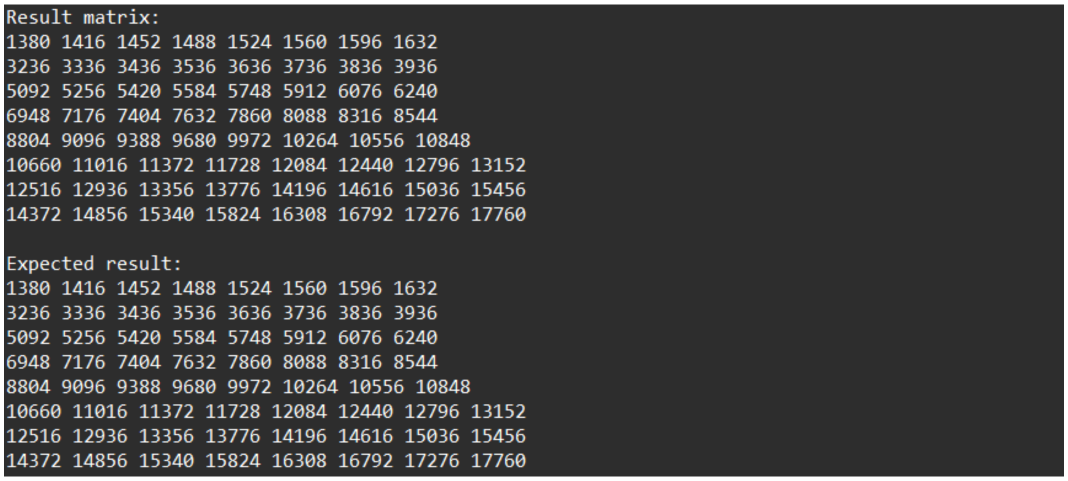


```
Time=25000
 Row1:    1     0     0     0     0     0     0     0
 Row2:    0     0     0     0     0     0     0     0
 Row3:    0     0     0     0     0     0     0     0
 Row4:    0     0     0     0     0     0     0     0
 Row5:    0     0     0     0     0     0     0     0
 Row6:    0     0     0     0     0     0     0     0
 Row7:    0     0     0     0     0     0     0     0
 Row8:    0     0     0     0     0     0     0     0

Time=35000
 Row1:   19     2     0     0     0     0     0     0
 Row2:    9     0     0     0     0     0     0     0
 Row3:    0     0     0     0     0     0     0     0
 Row4:    0     0     0     0     0     0     0     0
 Row5:    0     0     0     0     0     0     0     0
 Row6:    0     0     0     0     0     0     0     0
 Row7:    0     0     0     0     0     0     0     0
 Row8:    0     0     0     0     0     0     0     0

Time=45000
 Row1:   70    22     3     0     0     0     0     0
 Row2:   99    18     0     0     0     0     0     0
 Row3:   17     0     0     0     0     0     0     0
 Row4:    0     0     0     0     0     0     0     0
 Row5:    0     0     0     0     0     0     0     0
 Row6:    0     0     0     0     0     0     0     0
 Row7:    0     0     0     0     0     0     0     0
 Row8:    0     0     0     0     0     0     0     0
```

```
Time=235000
 Row1: 1380 1416 1452 1488 1524 1560 1596 1632
 Row2: 3236 3336 3436 3536 3636 3736 3836 3936
 Row3: 5092 5256 5420 5584 5748 5912 6076 6240
 Row4: 6948 7176 7404 7632 7860 8088 8316 8544
 Row5: 8804 9096 9388 9680 9972 10264 10556 10848
 Row6: 10660 11016 11372 11728 12084 12440 12796 13152
 Row7: 12516 12936 13356 13776 14196 14616 15036 15456
 Row8: 14372 14856 15340 15824 16308 16792 17276 17760
```

To save space and avoid redundancy, the attached screenshots only display the operations for the first few rows and initial/final cycles of execution. These are representative of the full matrix multiplication process which is correctly handled by the systolic array design across all cycles using a synchronised multiply-and-accumulate approach.

2. Behavioural Simulation of our basic Verilog Code

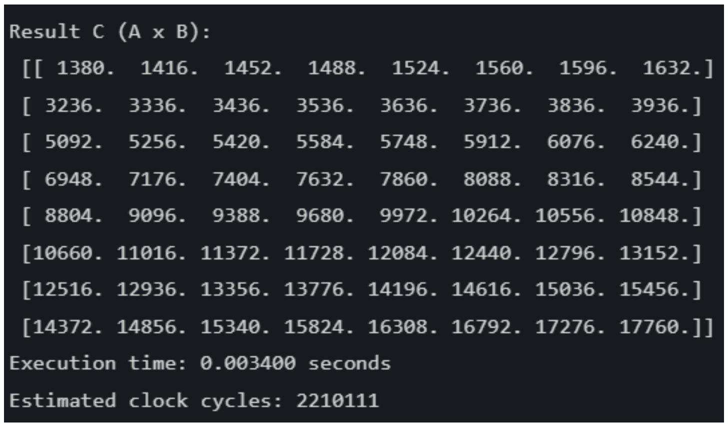


```
Result matrix:
1380 1416 1452 1488 1524 1560 1596 1632
3236 3336 3436 3536 3636 3736 3836 3936
5092 5256 5420 5584 5748 5912 6076 6240
6948 7176 7404 7632 7860 8088 8316 8544
8804 9096 9388 9680 9972 10264 10556 10848
10660 11016 11372 11728 12084 12440 12796 13152
12516 12936 13356 13776 14196 14616 15036 15456
14372 14856 15340 15824 16308 16792 17276 17760

Expected result:
1380 1416 1452 1488 1524 1560 1596 1632
3236 3336 3436 3536 3636 3736 3836 3936
5092 5256 5420 5584 5748 5912 6076 6240
6948 7176 7404 7632 7860 8088 8316 8544
8804 9096 9388 9680 9972 10264 10556 10848
10660 11016 11372 11728 12084 12440 12796 13152
12516 12936 13356 13776 14196 14616 15036 15456
14372 14856 15340 15824 16308 16792 17276 17760
```

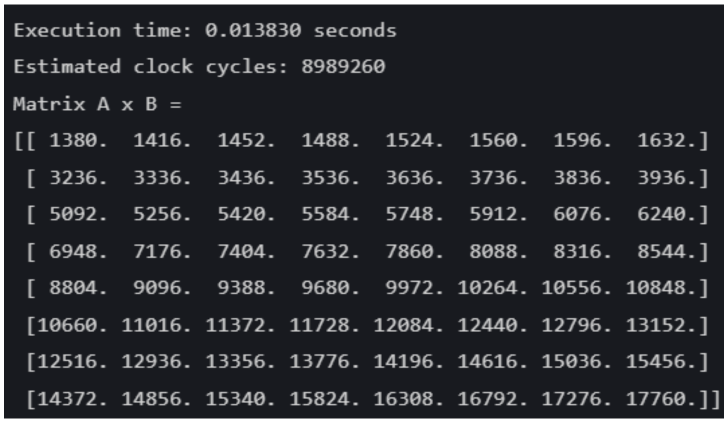
Results and Comparison

Since the Verilog implementations using AXI-Stream and AXI4-Lite did not produce the desired results, we are providing the results and comparisons made using Vitis HLS. Below, we present the hardware and software simulation results, including the final matrix multiplication outcomes, clock cycles, computation time, and the speedup achieved.

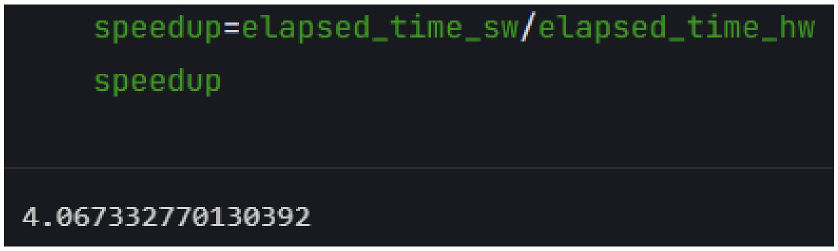
- Attached are two screenshots: one from the hardware simulation and one from the software simulation, showing the matrix results, clock cycles, computation time, and speedup of both approaches.



```
Result C (A x B):
 [[ 1380.  1416.  1452.  1488.  1524.  1560.  1596.  1632.]
 [ 3236.  3336.  3436.  3536.  3636.  3736.  3836.  3936.]
 [ 5092.  5256.  5420.  5584.  5748.  5912.  6076.  6240.]
 [ 6948.  7176.  7404.  7632.  7860.  8088.  8316.  8544.]
 [ 8804.  9096.  9388.  9680.  9972. 10264. 10556. 10848.]
 [10660. 11016. 11372. 11728. 12084. 12440. 12796. 13152.]
 [12516. 12936. 13356. 13776. 14196. 14616. 15036. 15456.]
 [14372. 14856. 15340. 15824. 16308. 16792. 17276. 17760.]]
Execution time: 0.003400 seconds
Estimated clock cycles: 2210111
```



```
Execution time: 0.013830 seconds
Estimated clock cycles: 8989260
Matrix A x B =
[[ 1380.  1416.  1452.  1488.  1524.  1560.  1596.  1632.]
 [ 3236.  3336.  3436.  3536.  3636.  3736.  3836.  3936.]
 [ 5092.  5256.  5420.  5584.  5748.  5912.  6076.  6240.]
 [ 6948.  7176.  7404.  7632.  7860.  8088.  8316.  8544.]
 [ 8804.  9096.  9388.  9680.  9972. 10264. 10556. 10848.]
 [10660. 11016. 11372. 11728. 12084. 12440. 12796. 13152.]
 [12516. 12936. 13356. 13776. 14196. 14616. 15036. 15456.]
 [14372. 14856. 15340. 15824. 16308. 16792. 17276. 17760.]]
```



```
speedup=elapsed_time_sw/elapsed_time_hw
speedup

4.067332770130392
```

Hardware (Vitis HLS):

- Execution time: 0.003400 seconds
- Estimated clock cycles: 2,210,111

Software (CPU):

- Execution time: 0.013830 seconds
- Estimated clock cycles: 8,989,260

Speedup: The hardware implementation achieved a speedup of 4.07x over the software solution.

Insights:

- The hardware design is **4.07 times** faster than the software implementation, highlighting the advantage of hardware acceleration for computationally intensive tasks.
- The hardware solution is more efficient in terms of clock cycles, completing the task in fewer cycles, which indicates lower potential power consumption and faster data processing.
- This performance improvement demonstrates the scalability of the hardware design, which can be further optimised for larger matrices or other applications requiring high throughput.

Conclusion

This project focused on accelerating matrix multiplication using a systolic array architecture on an FPGA. Despite challenges with AXI-Stream and AXI4-Lite interfaces during the Verilog implementation, we successfully developed a working solution using Vitis HLS.

The HLS-based design achieved a **4.07x speedup** over the software implementation, with improved execution time and clock cycles. These results confirm that hardware acceleration provides substantial performance benefits for matrix operations.

Though the Verilog implementation faced issues, the project highlighted the potential of FPGA-based hardware acceleration for real-time applications requiring high-performance computation. Future work could involve refining the Verilog implementation and exploring optimisations for larger matrices.

Acknowledgements

In conclusion, we sincerely thank our university and Professor Dr. Rakesh Palisetty for the opportunity to undertake this project as part of this course.

We are especially grateful to him for allowing us to work in the labs during weekends and off-hours, which played a crucial role in helping us make consistent progress on our implementation.

We also sincerely thank Mohan Sir, our lab in-charge, and Aditya, Neha & Prithvi, our UG TAs, for their guidance and assistance throughout the course and lab sessions.

A special thanks to our CR, Varun, for ensuring smooth communication between the professor and the class, for taking charge of lab access and coordination during weekends, and for helping with our project.

References

1. https://ieeexplore.ieee.org/document/10308496
2. https://ecelabs.njit.edu/ece459/lab3.php
3. https://www.mdpi.com/2079-9292/9/2/338
4. https://youtu.be/2VrnkXd9QR8?si=3bibvcZnE4nnOQfl
5. https://youtu.be/gJ1hh11V_2k?si=SLs1PDrdrU-TpebG
6. https://www.researchgate.net/publication/252951671_Design_and_FPGA_Implementation_of_Systolic_Array_Architecture_for_Matrix_Multiplication
7. https://zipcpu.com/blog/2021/08/28/axi-rules.html
8. https://youtu.be/Ii9JeVHCv_o?si=889O8q3LYHBvYQXG
9. https://youtu.be/BONXPKXyJTk?si=8r_TDtBKK1crZf4h
10. https://github.com/vinodpa/OpenProjects/tree/master/MyCodes/axi_ip
11. https://chatgpt.com/