STMicroelectronics

AI Accelerators Hands-on HW Design

Jul.2025

life.augmented

LAB 0

Systolic Array for Applying Matrix Multiplication

Submitted By:

Razan Megahed Raghib

Submitted to:

Eng. Ahmed Abdelsalam

Table of Contents

List of figures

I. Introduction

A systolic array is a specialized hardware architecture designed for efficient matrix multiplication. It consists of a network of interconnected processing elements (PEs) that operate in parallel, allowing for high throughput and pipelined computation. Data flows through the array in a "pulsating" manner (hence the name "systolic"), with each PE performing a small part of the overall calculation. This approach minimizes data movement and maximizes computational concurrency, making it well-suited for accelerating matrix operations.

II. Project Overview

The main objective of this project is to design a circuit that performs array multiplication for two NxN Arrays using the systolic array architecture. For this the following simplified diagram (figure 1) was realized

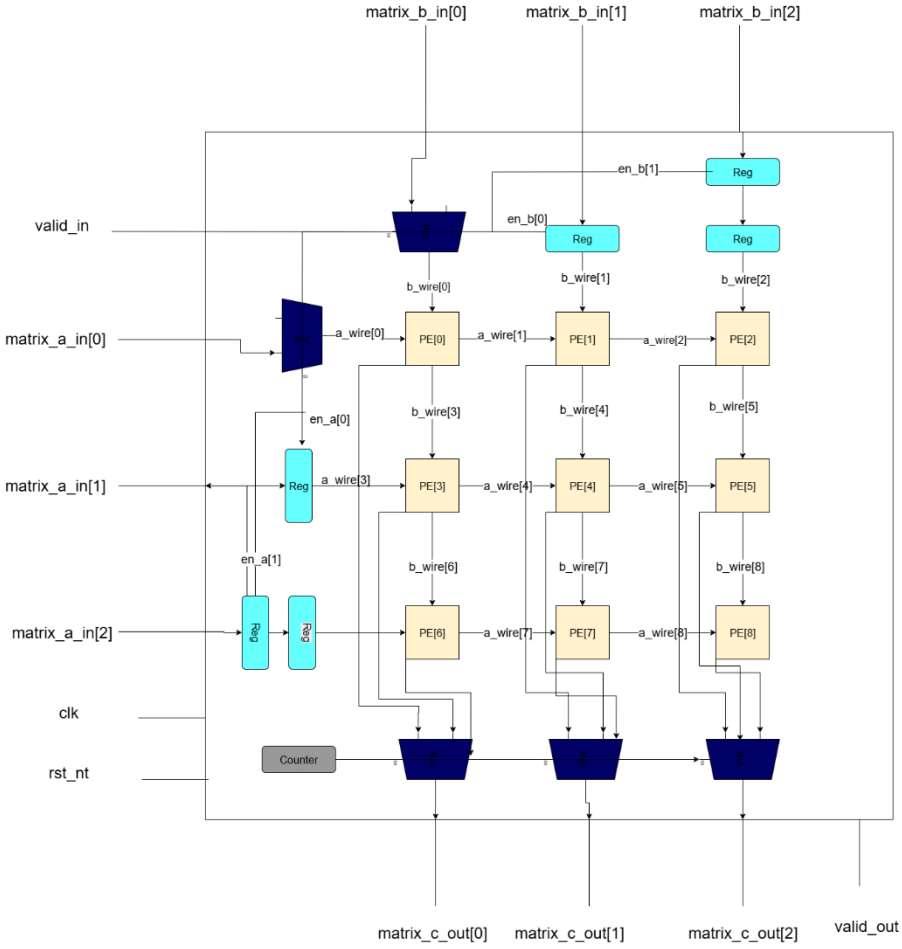

Figure 1: 3x3 Systolic Array Design

The input A is given one column at a time, the input B is given one row at a time, a valid_in signal must be asserted for the inputs to be read by the circuit. Each row (or column) is connected to delay registers corresponding to its index number (ie element 0 is not connected to a register, element 1 is connected to 1 register, and so on). The Flow of the circuit is controlled by a FSM with 3 states and the counter as an input (figure 2). the operation of FSM to be discussed in a later section.

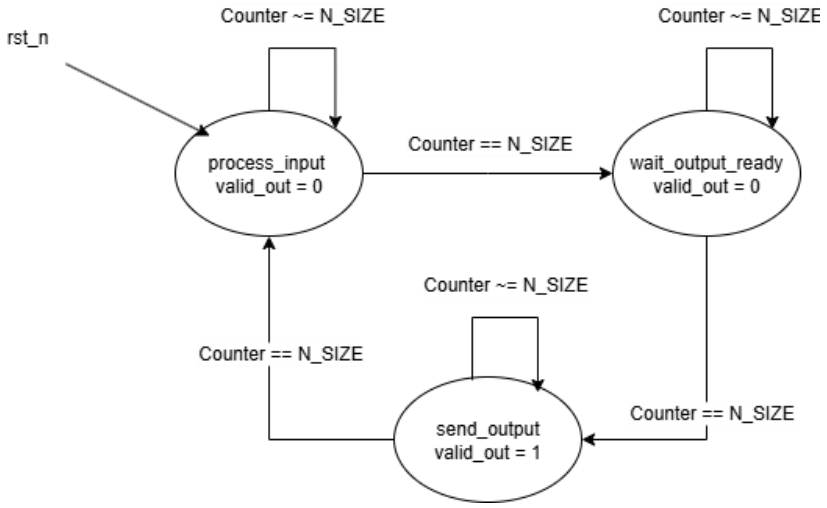

Figure 2: Finite State Machine

III. Components

The Design Contains 4 main modules:

- Processing Element sub-module
- Register with enable sub-module
- N to 1 Multiplexer sub-module
- Systolic Array Main module

1. Processing Element Module

- Port List

Port Name	Direction	Width	Description
Clk	Input	1-bit	Positive edge clock signal
rst_n	Input	1-bit	Negative edge reset
A	Input	DATAWIDTH	A single element from Matrix A sent from the previous block to be processed and sent to the next PE block
B	Input	DATAWIDTH	A single element from Matrix B to be processed and sent to the next block
A_shifted	Output	DATAWIDTH	A single element from Matrix A to be sent to the next PE block
B_shifted	Output	DATAWIDTH	A single element from Matrix B to be sent to the next PE block
C	Output	DATAWIDTH*2	The final output of the Multiplication

- Code

```
161  module PE #(parameter DATAWIDTH = 16, N_SIZE = 3) (
162      input clk,rst_n,   // Clock
163      input [DATAWIDTH - 1:0] A,B,
164      output reg [DATAWIDTH - 1:0] A_shifted, B_shifted,
165      output reg [DATAWIDTH*2 - 1:0] C
166  );
167
168      always @(posedge clk or negedge rst_n) begin
169          if(~rst_n) begin
170              C <= 0;
171              A_shifted <= 0;
172              B_shifted <= 0;
173          end else begin
174              A_shifted <= A;
175              B_shifted <= B;
176              C <= C + (A*B);
177          end
178      end
179  endmodule
```

Figure 3: PE block module

The Module Simply takes A[i][j] and B[i][j] given either from the input or from a previous PE and Multiplies and adds them to the output. It is also responsible for sending the A,B that to the next PE.

2. N to 1 Multiplexer

- Port List

Port Name	Direction	Width	Description
en	Input	1-bit	Enable signal
Din	Input	DATAWIDTH*N_SIZE	An array of N_SIZE elements
sel	input	Log2(N_SIZE)	Selection line
Dout	output	DATAWIDTH*N_SIZE	The selected output of the multiplexer

- Code

```
module MUX_N_1 #(parameter DATAWIDTH = 32, N_SIZE = 3)(
    input en,
    input [DATAWIDTH - 1:0] Din [N_SIZE],
    input reg [$clog2(N_SIZE) - 1:0] sel,
    output reg [DATAWIDTH - 1:0] Dout
);

always @(*) begin
    if(en)
        Dout = Din[sel];
    else
        Dout = 0;
end

endmodule
```

Figure 4: N to 1 MUX module

The module is responsible for choosing the element of the array to be sent to the output with a counter as its selection line, this module is instantiated N times with each instance having a column from C, (ie for a 3x3 matrix, there will be 3 instances each connected to a column, if the counter is set to 0, elements of row 0 will be selected)

3. Register with Enable

- Port List

Port Name	Direction	Width	Description
Clk	Input	1-bit	Positive edge clock signal
rst_n	Input	1-bit	Negative edge reset
en	Input	1-bit	Enable Signal
D	Input	DATAWIDTH	Input data
Q	output	DATAWIDTH	Output from the flip-flop
en_next	output	1-bit	High when data is registered

- Code

```
module register #(parameter DATAWIDTH = 16, N_SIZE = 3) (
    input clk,    // Clock
    input rst_n,en,  // Asynchronous reset active low
    input [DATAWIDTH - 1:0] D,
    output reg [DATAWIDTH - 1:0] Q,
    output reg en_next
);

always @(posedge clk or negedge rst_n) begin
    if(~rst_n) begin
        Q <= 0;
    end else begin
        if (en) begin
            Q <= D;
            en_next <= 1;
        end
        else begin
            Q <= 0;
            en_next <= 0;
        end
    end
end
endmodule
```

Figure 5: register module

This module is responsible for delaying the input to simulate the systolic array sequence, it is controlled by an enable signal in order to synchronize each register with the next, if the register is not enabled, this means that either there is no input signal, or the register before has not received the input yet, therefore in the case the enable is off, we must put 0 in the output

4. Systolic Array main module

- Parameter List

Port Name	Type	Default Value	Description
DATAWIDTH	Integer	16	Datawidth of elements in in
N_SIZE	Integer	5	The size of the resulting matrix or the Number of PEs in each row or columns we assumed square matrix for simplicity

- Port List

Port Name	Direction	Width	Description
clk	Input	1-bit	Positive edge clock signal
rst_n	Input	1-bit	Negative edge reset
valid_in	Input	1-bit	Valid signal set to 1 when a valid data are settled on ***'matric_a_in'*** and ***'matric_a_in'*** so the DUT is allowed to sample them

matrix_a_in	Input	N_SIZE* DATAWIDTH	Array of N inputs corresponding to one column of matrix A elements entering the systolic array rows.
matrix_b_in	Input	N_SIZE* DATAWIDTH	Array of N inputs corresponding to one row of matrix B elements entering the systolic array columns.
valid_out	Output	1-bit	Valid signal set to 1 when a valid row of the result matrix are settled on ***'matric_c_out'***
matrix_c_out	Output	N_SIZE*2* DATAWIDTH	Array of 5 outputs corresponding to one row of matrix C elements resulting from the array multiplication

- Code

The Module can be divided into 2 parts, the first is the generate part. This part ensures parametrized connection between each component.

```
//Generate delay registers, each row or coulmn should be delayed by idx before entering the PE
for (i = 1; i < N_SIZE; i++) begin
    reg [DATAWIDTH - 1:0] shift_wire_a [i], shift_wire_b[i];
    reg en_next_a[i], en_next_b[i]; //each register in a row (or coulmn) enables the next
    for (j = 0; j < i; j++)
    begin
        if (i == 1) //Corner case: first row (or coulmn) needs only one register from input to PE block
        begin
            register #(.DATAWIDTH(DATAWIDTH), .N_SIZE(N_SIZE)) sr1 (.clk(clk), .en(valid_in), .rst_n(rst_n), .D(matrix_a_in[i]), .Q(a_wire[i*N_SIZE]), .en_next(buff_a));
            register #(.DATAWIDTH(DATAWIDTH), .N_SIZE(N_SIZE)) sr2 (.clk(clk), .en(valid_in), .rst_n(rst_n), .D(matrix_b_in[i]), .Q(b_wire[i]), .en_next(buff_b));
        end
        else if (j == 0)
        begin //the first instance is connected to the input and enabled by valid_in
            register #(.DATAWIDTH(DATAWIDTH), .N_SIZE(N_SIZE)) sr1 (.clk(clk),.en(valid_in), .rst_n(rst_n), .D(matrix_a_in[i]), .Q(shift_wire_a[j]), .en_next(en_next_a[j]));
            register #(.DATAWIDTH(DATAWIDTH), .N_SIZE(N_SIZE)) sr2 (.clk(clk),.en(valid_in), .rst_n(rst_n), .D(matrix_b_in[i]), .Q(shift_wire_b[j]), .en_next(en_next_b[j]));
        end
        else if (j == i - 1)
        begin // the last instance is connected to the PE blocks
            register #(.DATAWIDTH(DATAWIDTH), .N_SIZE(N_SIZE)) sr1 (.clk(clk),.en(en_next_a[j - 1]), .rst_n(rst_n), .D(shift_wire_a[j - 1]), .Q(a_wire[i*N_SIZE]), .en_next(buff_a));
            register #(.DATAWIDTH(DATAWIDTH), .N_SIZE(N_SIZE)) sr2 (.clk(clk), .en(en_next_b[j - 1]), .rst_n(rst_n), .D(shift_wire_b[j - 1]), .Q(b_wire[i]), .en_next(buff_b));
        end
        else
        begin  //otherwise connect the registers to eachother in sequence
            register #(.DATAWIDTH(DATAWIDTH), .N_SIZE(N_SIZE)) sr1 (.clk(clk), .en(en_next_a[j - 1]),.rst_n(rst_n), .D(shift_wire_a[j - 1]), .Q(shift_wire_a[j]), .en_next(en_next_a[j]));
            register #(.DATAWIDTH(DATAWIDTH), .N_SIZE(N_SIZE)) sr2 (.clk(clk), .en(en_next_b[j - 1]),.rst_n(rst_n), .D(shift_wire_b[j - 1]), .Q(shift_wire_b[j]), .en_next(en_next_b[j]));
        end
    end
end
```

Figure 6: delay registers connection

This is the part where we connect the input to the delay registers of the module. The first row (or column) is connected to multiplexer with valid_in as its selector, so it is not included in the for loop. In each row, the first register is connected to the input and the enable to valid_in. the second register is connected to enable_next and Q of the first, (ie the first register enables the second) and so on, with the final register's output connected to the corresponding PE block.

```
//processing elements wiring
for(i=0; i<N_SIZE*N_SIZE; i=i+1) begin
    //for the blocks that contain outputs that we will not use (ex A_shifted for the last column)
    wire [DATAWIDTH - 1:0] buffer_a,buffer_b;
    //if the current row and coulmn are the last
    if ( ((i + 1) % N_SIZE == 0) && (i >= (N_SIZE - 1)*N_SIZE) )
        PE #(.DATAWIDTH(DATAWIDTH), .N_SIZE(N_SIZE)) pe_blocks  (.clk(clk), .rst_n(rst_n), .A(a_wire[i]), .B(b_wire[i]), .A_shifted(buffer_a), .B_shifted(buffer_b), .C(c_wire[i]));
    else if ((i + 1) % N_SIZE == 0) //if the current  coulmn is the last
        PE #(.DATAWIDTH(DATAWIDTH), .N_SIZE(N_SIZE)) pe_blocks (.clk(clk), .rst_n(rst_n), .A(a_wire[i]), .B(b_wire[i]), .A_shifted(buffer_a), .B_shifted(b_wire[i + N_SIZE]), .C(c_wire[i]));
    else if (i >= (N_SIZE - 1)*N_SIZE) //if the current row is the last
        PE #(.DATAWIDTH(DATAWIDTH), .N_SIZE(N_SIZE)) pe_blocks (.clk(clk), .rst_n(rst_n), .A(a_wire[i]), .B(b_wire[i]), .A_shifted(a_wire[i + 1]), .B_shifted(buffer_b), .C(c_wire[i]));
    else // general case
        PE #(.DATAWIDTH(DATAWIDTH), .N_SIZE(N_SIZE)) pe_blocks (.clk(clk), .rst_n(rst_n), .A(a_wire[i]), .B(b_wire[i]), .A_shifted(a_wire[i + 1]), .B_shifted(b_wire[i + N_SIZE]), .C(c_wire[i]));
end
```

Figure 7: PE wiring code

This part is responsible for connecting the processing elements to each other. For illustration the following figure shows the final connection with labeled wires

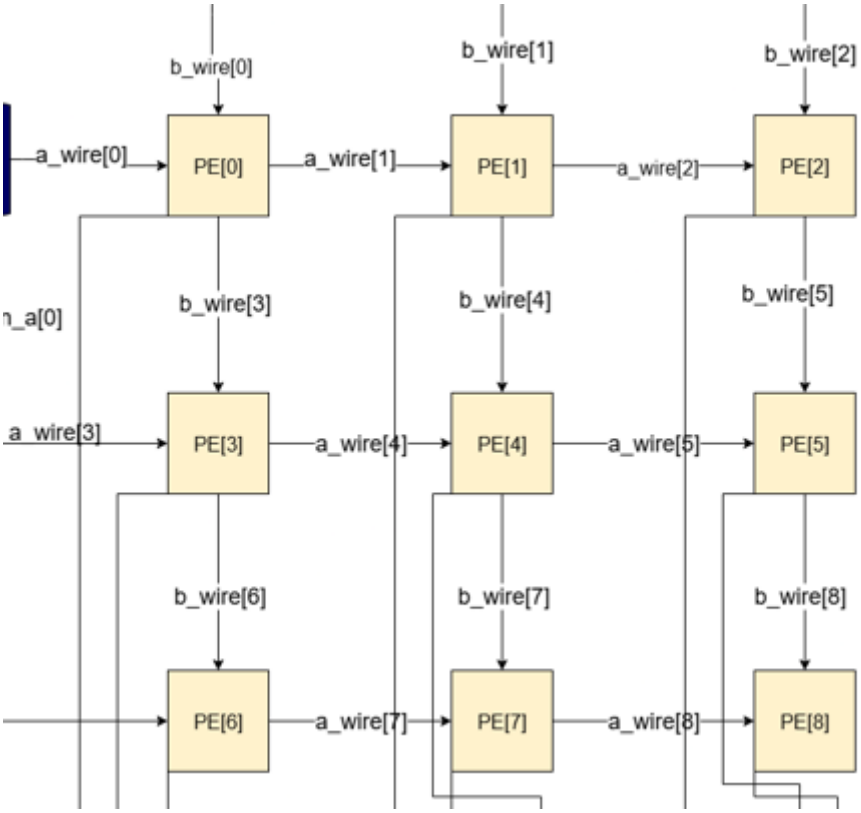

Figure 8: PE wiring diaram

Finally, the connection of the output muxes

```
for (i = 0; i < N_SIZE; i++)
begin  : for_outer
    wire [DATAWIDTH*2 - 1:0] in_bus [N_SIZE];
    for (j = 0; j < N_SIZE; j ++) begin  : for_inner
        assign in_bus[j] = c_wire[j*N_SIZE + i];
```

Figure 9: Output MUX

The second part of the code is a simple FSM of 3 states

1. Process_input: increments the counter in the case that valid_in is asserted (input is collected) otherwise it waits for a valid_in signal
2. Wait_output_ready: waits N cycles for the output to be ready
3. Collect_output: enables the valid_out signal for N cycles.

```
always @(*) begin
    case(current_state)
        process_input:
        begin
            //if N_SIZE inputs are given, move to the next state and set the counter to 0
            if (count == (N_SIZE - 1)) begin
                next_state = wait_output_ready;
                count_next = 0;
            end
            else begin //else if valid_in is asserted, increment the counter, if not keep it at the same time it was before
                next_state = process_input;
                if (valid_in)
                    count_next = count + 1;
                else
                    count_next = count;
            end
        end
        wait_output_ready:
        begin
            //output of the first row is ready after N_SIZE clock cycles
            if (count == N_SIZE) begin
                //if N_SIZE cycles have passed, move to the next state
                next_state = send_output;
                count_next = 0;
            end
            else begin
                //otherwise, stay in the same state and increment the counter
                next_state = wait_output_ready;
                count_next = count + 1;
            end
        end
        send_output:
        begin
            if (count == (N_SIZE - 1)) begin //if all outputs are sent, move back to the inital state and set the counter to 0
                next_state = process_input;
                count_next = 0;
            end
            else begin //otherwise, stay in the same state and increment the counter
                next_state = send_output;
                count_next = count + 1;
            end
        end
        default: //default state
        begin
            next_state = 2'b00;
            count_next = 0;
        end
    endcase
end
```

Figure 10: FSM code

II. Simulation Results

Two tests were performed, one on 3x3 arrays and the other on 5x5 arrays

Results from log file:

```
Testcase 1: 3x3 matix
A:
'{1, 2, 3}
'{4, 5, 6}
'{7, 8, 9}
B:
'{1, 0, 0}
'{0, 2, 0}
'{0, 0, 3}
Output:
'{1, 4, 9}
'{4, 10, 18}
'{7, 16, 27}
Testcase 2: 5x5 matixA:
'{1, 2, 3, 4, 5}
'{6, 7, 8, 9, 10}
'{11, 12, 13, 14, 15}
'{1, 2, 3, 4, 5}
'{6, 7, 8, 9, 10}
B:
'{2, 4, 6, 8, 10}
'{12, 14, 16, 18, 20}
'{22, 24, 26, 28, 30}
'{1, 2, 3, 4, 5}
'{6, 7, 8, 9, 10}
Output:
'{126, 147, 168, 189, 210}
'{341, 402, 463, 524, 585}
'{556, 657, 758, 859, 960}
'{126, 147, 168, 189, 210}
'{341, 402, 463, 524, 585}
```

Figure 11: Log file

Output Waveform:

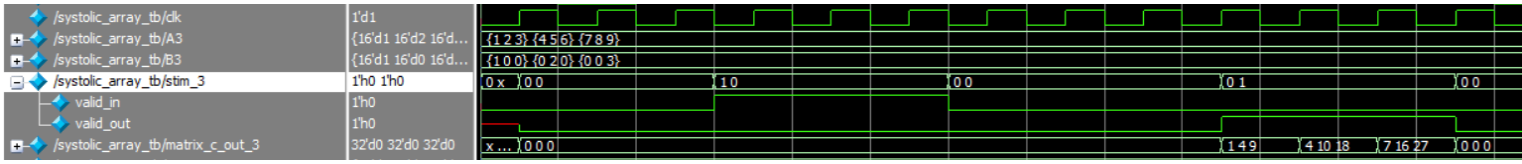

Figure 12 3x3 output waveform

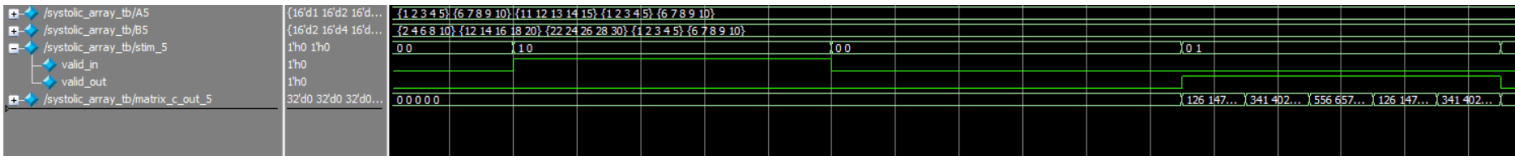

Figure 13: 5x5 output waveform

IV. Conclusion

Systolic Array algorithm is a complex yet powerful algorithm for matrix multiplication. Its design provides important practice on different design concepts and architectures. But also on important yet underused systemverilog concepts such as generate blocks and utilizing them for system wiring for parametrized use.