Department of Electrical Engineering

MINI PROJECT II

Design, Synthesize and Test Array Multiplier Working in Systolic Manner

I. Introduction

In modern digital design, efficient matrix multiplication is critical for applications in signal processing, machine learning, and scientific computation. To address the need for high-performance and power-efficient computation, systolic array architectures have emerged as a powerful solution. This mini-project focuses on the design, synthesis, and testing of a 4x4 array multiplier operating in a systolic manner, utilizing Verilog/VHDL and Xilinx Vivado/Vitis tools.

A systolic array is an arrangement of processing elements (PEs) that work in parallel, passing data between neighboring elements in a rhythmic, pulse-like manner. This approach reduces memory access and interconnect complexity, making it ideal for implementing matrix multiplications in hardware. The project extends the 4x4 design to larger scales, including 16x16, 32x32, and 64x64 multipliers, allowing for a comprehensive analysis of power, performance, and area (PPA) trade-offs. By leveraging bit-accurate half-precision floating-point multiply-accumulate units (MACs) developed in previous work, this project aims to create a highly parallel, scalable, and efficient multiplier design suitable for deployment on FPGA platforms.

II. Design and Operation of a 4x4 Systolic Array Multiplier

1. Array Multiplier Architecture

An array multiplier is a structured approach to implementing multiplication using a grid of processing elements (PEs) that operate in parallel. In the case of a 4x4 array multiplier, there are 16 PEs arranged in a 4x4 matrix. Each PE is responsible for computing partial products and adding them to produce a final result. This parallel architecture allows for the simultaneous computation of different parts of the multiplication, making the array multiplier highly efficient for matrix multiplications. The design is particularly effective for fixed-point and floating-point operations due to its regular structure and predictable data flow.

An array multiplier uses a grid of processing elements (PEs) to implement the multiplication process. For example, in a 4x4 array multiplier, the goal is to multiply two 4x4 matrices, AAA and BBB, to produce a resulting matrix C:

$$C = A \times B$$

Where:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

The result is:

$$C = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

Each element c_{ij} is computed as:

$$c_{ij} = \sum_{k=1}^{4} a_{ik} \times b_{kj}$$

For example, c_{11} would be calculated as:

$$c_{11} = a_{11} \times b_{11} + a_{12} \times b_{21} + a_{13} \times b_{31} + a_{14} \times b_{41}$$

This multiplication and addition process is distributed across the PEs in the array, allowing for parallel computation.

2. Systolic Array Architecture

A systolic array is a specialized architecture where a series of PEs are connected in a way that allows data to flow between them in a rhythmic, pulse-like manner—similar to the way a heart pumps blood. Each PE in the array performs a simple operation, such as multiplication or addition, and passes the result to its neighboring PEs. This structure minimizes the need for complex memory access patterns by keeping data movement localized to adjacent elements. It also supports a pipeline-like process where data flows through the array in a synchronized manner, leading to high throughput and efficient use of computational resources.

A systolic array is characterized by data flow between neighboring PEs. Consider the same multiplication C=A×B:

- Matrix AAA elements are fed into the array from the left, while matrix BBB elements are fed from the top.
- Each PE computes a partial product and accumulates the result from previous computations. For instance, in a single PE, the operation can be represented as:

$$\text{PE}_{ij}: \quad c_{ij} = c_{ij} + a_{ij} \times b_{ij}$$

As data moves through the array, the partial sums c_{ij} are propagated to the next PEs. For example, if a_{11} and b_{11} are input into the top-left PE, the multiplication $a_{11} \times b_{11}$ is performed, and the result is passed to the next PE in the row and column.

3. Working of a 4x4 Systolic Array Multiplier

In a 4x4 systolic array multiplier, the two matrices to be multiplied are typically input from different edges of the array. For example, matrix A is provided row-wise while matrix B is input column-wise. Each PE within the array computes a multiply-accumulate (MAC) operation, multiplying its inputs and adding the result to an accumulated sum. As data propagates through the array, each PE contributes to a part of the final output. The systolic data flow ensures that intermediate results move through the array, with final results emerging at the output edge of the array. This structure makes the 4x4 systolic array an effective design for hardware-accelerated matrix multiplication tasks.

Let's illustrate the flow of data with a smaller example using a 2x2 array multiplier for simplicity. Assume A and B are 2x2 matrices:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

The systolic array calculates each element of the result C as:

$$C = \begin{bmatrix} 1 \times 5 + 2 \times 7 & 1 \times 6 + 2 \times 8 \\ 3 \times 5 + 4 \times 7 & 3 \times 6 + 4 \times 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

As AAA and BBB flow through the systolic array, each PE handles one multiplication and accumulation step. For example, c_{11} is computed by the top-left PE, and its intermediate result is passed along the array to help compute the subsequent elements.

4. Scaling to Larger Sizes (16x16, 32x32, 64x64)

The systolic array design can be extended to handle larger matrices by increasing the number of PEs in the grid. For instance, a 16x16 systolic array would consist of 256 PEs, while a 32x32 design would require 1,024 PEs. The principles of operation remain the same, with data flowing through the array in a pipelined manner. However, scaling introduces new challenges, such as increased interconnect complexity and resource usage. The larger designs maintain their parallelism, but the area occupied on the FPGA and the power consumption increase. These factors must be balanced to achieve optimal performance for each matrix size.

The same principle of data flow applies to larger matrices. For example, a 16x16 array would multiply:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1,16} \\ \vdots & \vdots & \ddots & \vdots \\ a_{16,1} & a_{16,2} & \dots & a_{16,16} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1,16} \\ \vdots & \vdots & \ddots & \vdots \\ b_{16,1} & b_{16,2} & \dots & b_{16,16} \end{bmatrix}$$

The resulting matrix CCC is obtained by performing the same MAC operations across a larger grid of 256 PEs. Scaling to 32x32 or 64x64 increases the number of PEs accordingly, creating a more extensive array that can handle larger computations with greater parallelism.

5. PPA Analysis (Power, Performance, and Area)

PPA analysis is essential to understanding the trade-offs involved in the design of systolic arrays of different sizes:

- **Power**: The power consumption of the design is influenced by the number of active PEs and the frequency of data movement. As the array size increases, the power requirement generally rises due to the larger number of active components.
- **Performance**: Performance metrics include latency (the time required to complete a multiplication) and throughput (the number of operations completed per second). A larger systolic array typically has higher throughput due to its increased parallelism, but latency may also increase depending on data propagation time through the array.
- **Area**: The area represents the silicon space occupied by the design on the FPGA. It is directly related to the number of PEs and the complexity of each PE. As the size of the systolic array scales up, the area required also increases, affecting the overall resource utilization on the FPGA.

This PPA analysis helps identify the optimal design size for a given application, balancing the benefits of increased parallelism with the constraints of power and area.

III. Design and Implementation of the 4x4 Systolic Array Multiplier

1.1 Overview of the Systolic Array Multiplier

A systolic array multiplier is designed for efficient matrix multiplication using a network of parallel processing elements (PEs). Each PE computes partial products and accumulates them, passing intermediate results to neighboring PEs in a pipelined manner. This architecture is well-suited for FPGA implementation due to its structured data flow and parallelism.

In this project, Vitis HLS is used to describe the systolic array's behavior in C/C++ and synthesize it into HDL for deployment on FPGAs. Vitis HLS simplifies the process of optimizing data movement and parallel computation, enabling a scalable design for matrix multiplication.

1.2 Algorithm Description

The core algorithm of the systolic array relies on multiply-accumulate (MAC) operations performed by each PE:

$$c_{ij} = c_{ij} + a_{ik} \times b_{kj}$$

Where c_{ij} is the partial sum at PE (i,j), a_{ij} is an element from matrix A, and b_{ij} is an element from matrix B.

For example, multiplying two 4x4 matrices A and B:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The result is:

$$C = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

This parallel approach reduces latency and improves throughput, as PEs perform computations simultaneously and efficiently propagate intermediate sums.

1.3 Vitis HLS Implementation

To implement the 4x4 systolic array multiplier, three main source files are created. Each file has a specific role in defining, implementing, and testing the functionality of the systolic array.

1.3.1 systolic_array.h:

This header file serves as the definition file for the project. It defines the custom data type data_t, which represents the fixed-point format for matrix elements, and declares the function systolic_array. This ensures that data_t and systolic_array are consistently referenced across the implementation and testbench files.

```
#ifndef SYSTOLIC_ARRAY_H
#define SYSTOLIC_ARRAY_H

#include <ap_fixed.h>

// Define the data type (e.g., half-precision floating point)
typedef ap_fixed<16, 8> data_t;

#define N 4 // Matrix size (4x4)

void systolic_array(data_t A[N][N], data_t B[N][N], data_t C[N][N]);

#endif
```

1.3.2 systolic_array.cpp:

This file contains the implementation of the systolic_array function, which performs the 4x4 matrix multiplication in a systolic manner. Inside this function:

- Local buffers are used to hold intermediate results.
- The #pragma HLS directives optimize memory partitioning and pipelining, enabling efficient parallel processing on an FPGA.
- The core multiply-accumulate (MAC) operations for each processing element (PE) are performed within nested loops, where elements of matrices $\bar{A}$ and $\bar{B}$ are multiplied and accumulated to produce the final matrix

systolic_array.cpp

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

void systolic_array(data_t A[N][N], data_t B[N][N], data_t C[N][N]) {
    #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

    data_t local_A[N][N];
    data_t local_B[N][N];
    #pragma HLS array_partition variable=local_A complete dim=0
    #pragma HLS array_partition variable=local_B complete dim=0

    data_t buffer[N][N];
    #pragma HLS array_partition variable=buffer complete dim=0
```

```
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            buffer[i][j] = 0;
            local_A[i][j] = A[i][j];
            local_B[i][j] = B[i][j];
        }
    }

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < N; k++) {
                #pragma HLS PIPELINE II=1
                buffer[i][j] += local_A[i][k] * local_B[k][j];
            }
            C[i][j] = buffer[i][j];
        }
    }
}
```

1.3.3 tb_systolic_array.cpp:

This testbench file validates the functionality of the systolic_array function. It includes multiple test cases with predefined input matrices $\bar{A}$ *and* $\bar{B}$ and their expected output matrices. The testbench:

- Calls the systolic_array function with each test case.
- Compares the computed output matrix $\bar{C}$ against expected results.
- Displays a message indicating "No error" if the test passes or "There is an error" if there is a discrepancy.

```
#include "systolic_array.h"
#include <iostream>

int main() {
    data_t A[N][N] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12},
        {13, 14, 15, 16}
    };

    data_t B[N][N] = {
        {1, 0, 0, 0},
        {0, 1, 0, 0},
        {0, 0, 1, 0},
        {0, 0, 0, 1}
    };

    data_t C[N][N] = {0};
```

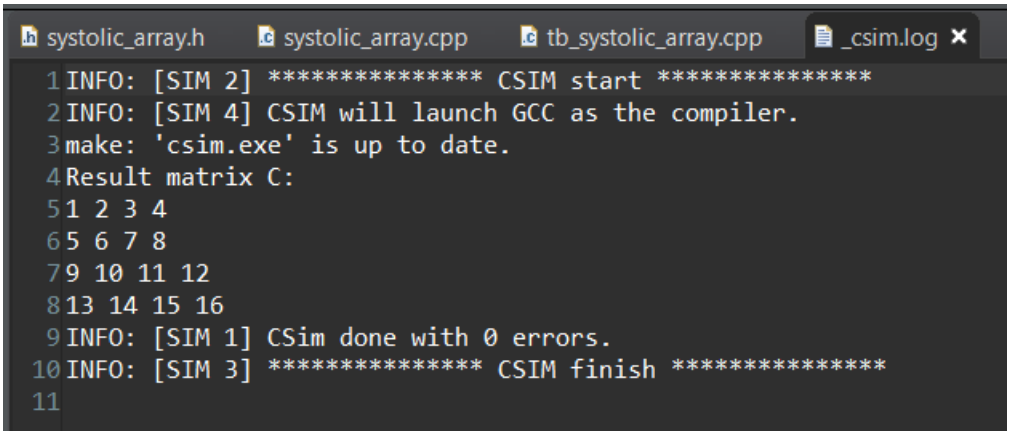
```
    // Call the systolic array function
    systolic_array(A, B, C);

    // Display the result matrix C
    std::cout << "Result matrix C:\n";
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            std::cout << C[i][j] << " ";
        }
        std::cout << "\n";
    }

    return 0;
}
```

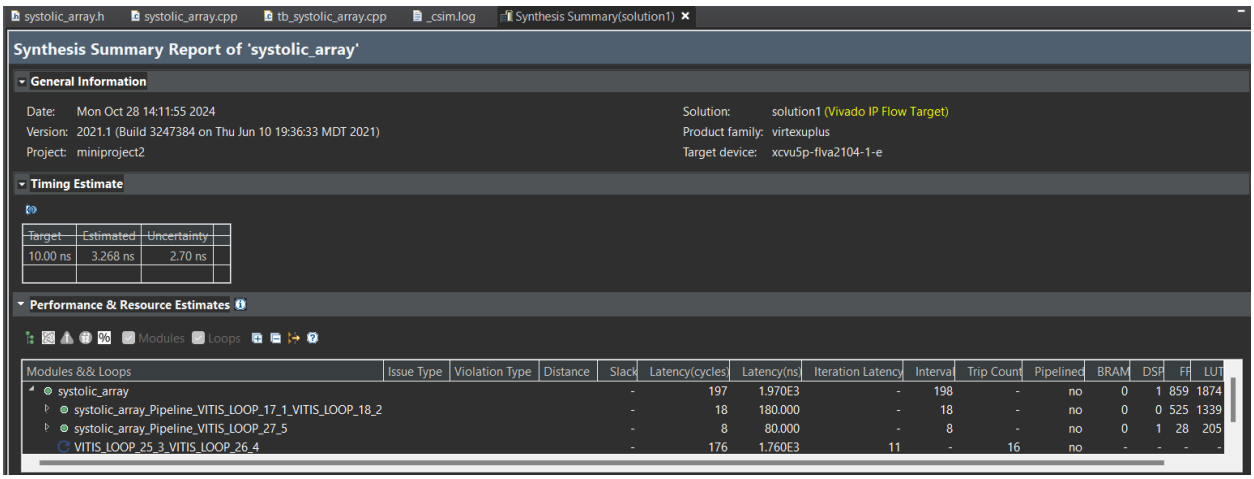
1.4 Run C simulation to view the expected output

The files will be compiled, and the output is shown in the Console window



1.5 Synthesis the design with the defaults and view the synthesis results

When synthesis is completed, the Synthesis Results will be displayed



The estimated clock period (3.268 ns) is significantly lower than the target of 10.00 ns, indicating that the design is capable of running at a higher frequency than specified. This translates to a maximum achievable frequency of approximately 306 MHz, which is beneficial for high-throughput applications. The design's uncertainty of 2.70 ns is within a manageable range, implying that timing closure is achievable without excessive optimization efforts.

With a latency of 197 cycles, the design efficiently completes a 4x4 matrix multiplication within a short time frame. The 1.9707 µs latency is relatively low, indicating that the systolic array architecture is effective for rapid matrix multiplication. This low latency, combined with the high clock frequency, makes the design suitable for applications that require fast computations, such as signal processing or machine learning.

The pipelining of inner loops significantly reduces latency, as shown by the lower cycle counts. The design's utilization of pipelining allows different parts of the computation to execute in parallel, enhancing data throughput. This configuration is advantageous for systolic arrays, where elements of matrices need to flow through the array concurrently.

The resource usage shows that the design does not use any BRAM or DSP blocks, which are typical resources for memory storage and arithmetic operations in FPGA designs. Instead, the design relies on **LUTs** (Look-Up Tables) and **FFs** (Flip-Flops) for implementing the logic and holding intermediate values. This approach might be due to the use of custom fixed-point arithmetic (ap_fixed type) rather than DSP blocks, making it adaptable for FPGAs with limited DSP resources.

- **LUTs**: The total number of LUTs utilized is 1,339, indicating moderate resource consumption. The design appears optimized for FPGAs, where LUT-based implementations allow greater flexibility in resource allocation.
- **FFs**: The design uses 1,874 Flip-Flops, which is relatively efficient and supports the fast operation of the systolic array without excessive resource use.

Overall Analysis and Suitability

- **Efficiency**: The design achieves high efficiency with low latency and relatively low resource consumption, especially without using DSP or BRAM blocks.
- **Performance**: With a high operating frequency (up to 306 MHz) and low latency, this systolic array multiplier is well-suited for real-time applications, especially where matrix multiplications are frequent and speed is critical.
- **FPGA Suitability**: Given the low BRAM and DSP usage, this design is optimal for FPGAs that may have limited DSP resources, such as older or cost-effective models. However, if higher performance is needed, exploring DSP usage might further reduce latency.

1.6 Run the C/RTL Co-simulation with the default settings of Verilog and verify that the simulation passes

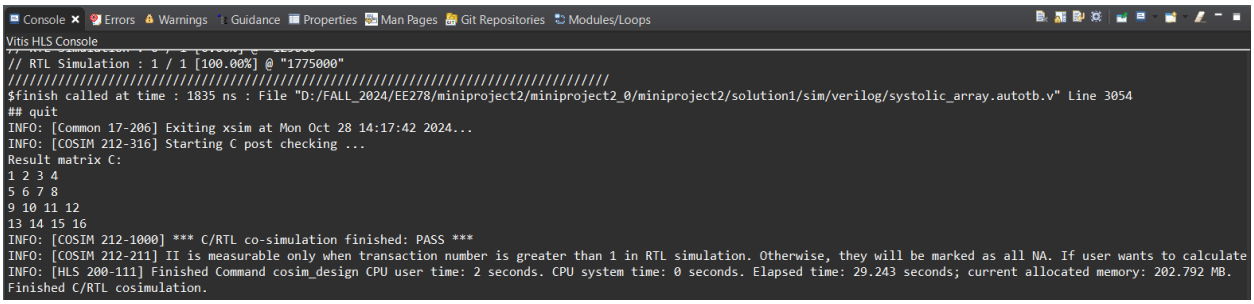
The C/RTL Co-simulation will run, generating and compiling several files, and then simulating the design. It goes through three stages

• First, the Verilog test bench is executed to generate input stimuli for the RTL design

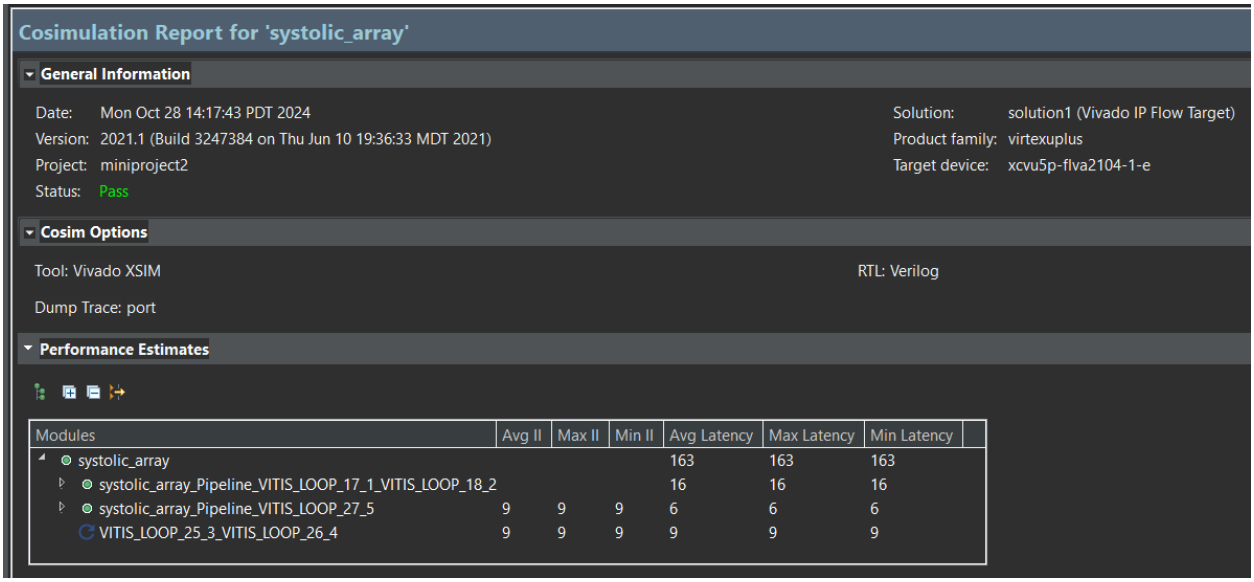
• Second, an RTL test bench with newly generated input stimuli is created and the RTL simulation is then performed

• Finally, the output from the RTL is re-applied to the Verilog test bench to check the results

In the console window you can see the progress and a message that the test is passed. This eliminates writing a separate testbench for the synthesized design



The co-simulation report provides a detailed summary of the execution and latency performance of the systolic_array function, focusing on the function's timing and pipeline efficiency.



The co-simulation report confirms that the systolic_array design is both functionally correct and highly efficient. Key highlights include:

- **Status**: Pass – The design passed all test cases, confirming functional accuracy.
- **Overall Latency**: 163 cycles – Consistent across all simulations, indicating stable performance.
- **Pipeline Efficiency**:
    - Loop VITIS_LOOP_18_2 has an initiation interval (II) of 16 cycles with 16-cycle latency.
    - Loop VITIS_LOOP_27_5 has an II of 9 cycles and a 6-cycle latency.

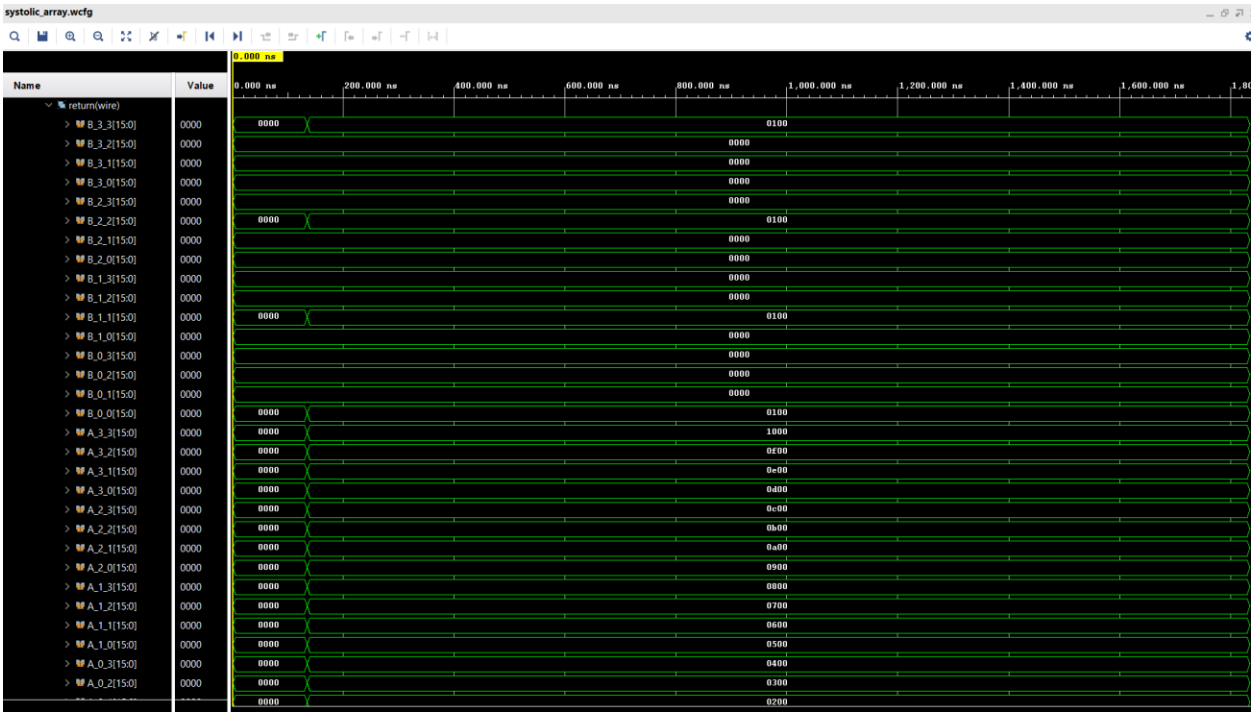
- Loop VITIS_LOOP_26_4 has an II and latency of 9 cycles.

These low II values ensure efficient pipelining, allowing multiple operations to overlap and enhancing throughput. This report suggests that the design is well-optimized for FPGA deployment, achieving fast, consistent matrix multiplication with minimal resource usage.

1.7 Viewing Simulation Results in Vivado

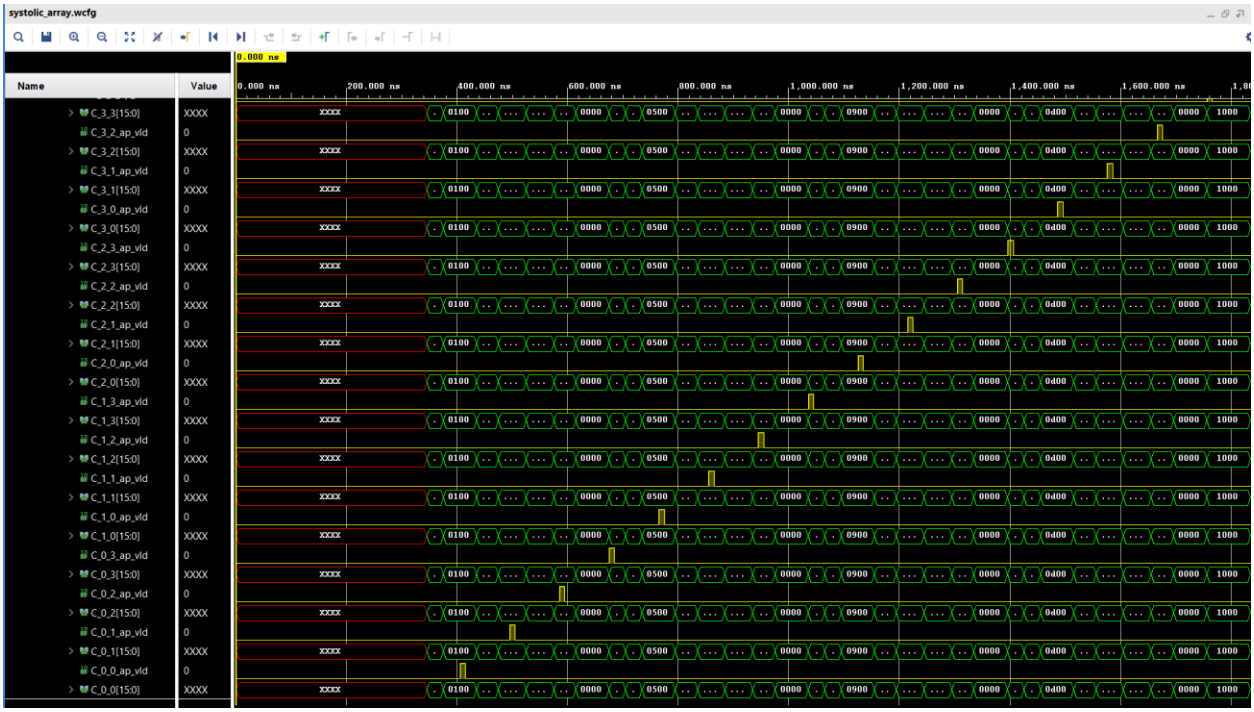
Input Matrix Signals (A and B):

- Each signal named A_<row>_<col> and B_<row>_<col> represents elements of matrices A and B, respectively.
- The values of these signals seem to be loaded correctly at the start, showing the initialization of input matrices A and B in a 4x4 format.
- For instance, A_0_0 through A_3_3 and B_0_0 through B_3_3 are present, indicating the 16 elements of each matrix.



Output Matrix Signals (C):

- Signals such as C_0_0, C_1_0, and C_3_3 represent elements of the output matrix C.
- These output signals show changes over time, which is expected as each processing element (PE) completes its multiply-accumulate (MAC) operations.
- The waveform shows incremental updates to the values in the output matrix, indicating the step-by-step accumulation of results across the pipeline.



Validity Signals (_vld):

- Each output element in C has a corresponding _vld signal (e.g., C_0_0_ap_vld) that indicates when the data is stable and valid.
- Observing the _vld signals going high (1) is crucial, as it marks when the corresponding output element in C is ready and stable.
- In the waveform, the _vld signals become active at different times, reflecting the completion of computation for each element.

Summary

- **Data Flow**: The systolic array's pipeline behavior is evident, with elements in matrix CCC progressively accumulating partial sums.
- **Output Stabilization**: Each C element stabilizes at the correct value upon completion of its computation, as indicated by the corresponding _vld signals.
- **Verification**: By comparing the final values of each C element with the expected matrix multiplication results, you can confirm that the design functions as intended.

IV. Developing the Design with 16x16, 32x32, and 64x64 cases

Expanding the systolic array multiplier design to accommodate larger matrix sizes (16x16, 32x32, and 64x64) requires strategic modifications to ensure that the design remains scalable, efficient, and resource-effective. This section outlines the key adjustments needed in the design,

covering matrix size parameterization, memory management, performance optimizations, resource allocation, and testing. These changes allow the design to efficiently handle a range of matrix sizes without compromising on performance or resource utilization.

1. Matrix Size Parameterization

To support flexible matrix sizes, the systolic array design must be parameterized. This approach enables the same core design to handle different matrix dimensions (e.g., 16x16, 32x32, 64x64) without needing separate implementations for each size.

- **Objective**: Ensure that the systolic array multiplier can be scaled to different matrix sizes without requiring core logic alterations.
- **Implementation**: Introduce a template parameter for matrix size in the function declarations and definitions. This parameterization allows the design to dynamically adjust the array size while reusing the core multiplication logic.
- **File Adjustments**:
    - **Header File (systolic_array.h)**: Add a template parameter (e.g., SIZE) to the systolic_array function declaration, enabling the design to adapt to different matrix sizes.
    - **Source File (systolic_array.cpp)**: Implement the systolic array function as a templated function. This templating allows us to instantiate the function with various matrix sizes (16x16, 32x32, 64x64) without modifying the underlying logic.

2. Memory Partitioning and Data Flow Management

Managing data flow efficiently is essential, especially as matrix sizes increase. Larger matrices require more data to be processed simultaneously, and managing this efficiently is critical to maintaining performance.

- **Objective**: Optimize memory access patterns to facilitate high-speed data flow and reduce latency, particularly for larger matrix sizes.
- **Implementation**: Use array partitioning pragmas (e.g., #pragma HLS ARRAY_PARTITION) for the input matrices AAA and BBB and the output matrix CCC. Array partitioning allows multiple elements to be accessed concurrently, a necessity for larger matrices where bottlenecks in data access can degrade performance.
- **File Adjustments**:
    - **Source File (systolic_array.cpp)**: Apply array partitioning pragmas to the input and output matrices to enable parallel data access across processing elements (PEs).

This adjustment helps maintain efficient data flow and minimizes latency as matrix size grows.

3. Performance Optimization Through Loop Pipelining and Unrolling

Larger matrix sizes increase the computational demand, making it essential to maximize throughput and minimize latency. Pipelining and loop unrolling are critical optimizations to achieve these goals.

- **Objective**: Maintain high throughput and low latency by fully utilizing parallelism, even as computational requirements increase with larger matrices.
- **Implementation**: Apply loop pipelining (#pragma HLS PIPELINE) and unrolling (#pragma HLS UNROLL) pragmas to the inner loops responsible for the multiply-accumulate (MAC) operations. Pipelining allows each PE to process a new input every cycle, while loop unrolling can further reduce the processing time by minimizing loop overhead.
- **File Adjustments**:
    - **Source File (systolic_array.cpp)**: Incorporate pipelining and unrolling pragmas into the MAC operation loops within the systolic array function. This optimization enhances the design's capability to handle larger matrices without a proportional increase in computation time.

4. Resource Management for Large Matrix Sizes

Efficient utilization of FPGA resources, such as Look-Up Tables (LUTs), Flip-Flops (FFs), and Digital Signal Processing (DSP) blocks, becomes increasingly important as matrix sizes grow. Without careful management, resource usage can escalate, impacting the design's scalability.

- **Objective**: Optimize resource utilization by leveraging DSP blocks for arithmetic operations, minimizing the use of LUTs and FFs where possible.
- **Implementation**: Utilize DSP blocks for the MAC operations, which are central to the systolic array's functionality. DSP blocks are optimized for high-speed arithmetic, and their use helps reduce the load on LUTs and FFs, which would otherwise increase significantly with larger matrices.
- **File Adjustments**:
    - **Source File (systolic_array.cpp)**: Apply #pragma HLS RESOURCE to specify the use of DSP blocks for multiplication operations. This approach maximizes arithmetic efficiency and conserves LUT and FF resources, making the design more resource-efficient and scalable.

5. Testbench Adaptation for Scalability

Testing the design's functionality for multiple matrix sizes is crucial to ensure that the parameterized design performs accurately and consistently. A flexible testbench structure allows the design to be validated across 16x16, 32x32, and 64x64 configurations.

- **Objective**: Verify the design's functionality across multiple matrix sizes, ensuring correctness and scalability.
- **Implementation**: Modify the testbench to support different matrix sizes using template-based functions. This approach allows testing of each matrix size by instantiating the systolic array function with the appropriate size and comparing the output matrix CCC with expected values.
- **File Adjustments**:
    - **Testbench File (tb_systolic_array.cpp)**: Implement templated test functions to test each matrix size (16x16, 32x32, and 64x64) independently. The testbench initializes matrices AAA and BBB for each size, runs the systolic array multiplier, and verifies that the output matrix CCC matches the expected results.

Summary of Adjustments

The following table summarizes the key adjustments needed to expand the systolic array multiplier to 16x16, 32x32, and 64x64 matrix sizes:

Adjustment	Objective	Implementation
Matrix Size Parameterization	Enable flexible matrix sizing	Use a template parameter for matrix size in function declarations and definitions
Memory Partitioning	Optimize data access for larger matrices	Apply array partitioning pragmas to input/output matrices for parallel data access
Loop Pipelining and Unrolling	Maintain high throughput and low latency	Use pipelining and unrolling pragmas on inner MAC operation loops
Resource Management	Efficiently use FPGA resources	Specify DSP blocks for MAC operations to reduce LUT and FF usage
Scalable Testbench	Validate functionality across multiple matrix sizes	Implement templated test functions to run and verify matrix multiplication for 16x16, 32x32, and 64x64 cases

By implementing these adjustments, the systolic array multiplier can be effectively scaled to handle 16x16, 32x32, and 64x64 matrix multiplications while maintaining efficiency and

performance. These modifications ensure that the design remains adaptable, resource-efficient, and suitable for larger computations on FPGA platforms.

6. Developing the design for multiple size matrix multiplications

6.1. Modifying the .h File to Support Multiple Sizes

To create a scalable design, it's useful to parameterize the matrix size. By defining a configurable parameter for the matrix size, you can easily adjust between 16x16, 32x32, and 64x64.

systolic_array.h

Here's an updated version of the header file to support different matrix sizes:

```
#ifndef SYSTOLIC_ARRAY_H
#define SYSTOLIC_ARRAY_H

#include <ap_fixed.h>

// Define data type (e.g., 16-bit fixed-point with 8 bits integer, 8 bits fractional)
typedef ap_fixed<16, 8> data_t;

// Define matrix size as a template parameter for scalability
template<int SIZE>
void systolic_array(data_t A[SIZE][SIZE], data_t B[SIZE][SIZE], data_t C[SIZE][SIZE]);

#endif
```

With this setup, you can now instantiate systolic_array with any matrix size by specifying the template parameter (e.g., systolic_array<16> for a 16x16 matrix).

6.2. Adjusting the .cpp File to Use Template Parameter for Different Matrix Sizes

In the .cpp file, we'll modify the function to support the matrix size as a template parameter. This allows the same code to handle 4x4, 16x16, 32x32, and 64x64 configurations without hardcoding specific dimensions.

systolic_array.cpp

Here's an updated version of the .cpp file using a template parameter for scalability:

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

// Templated function to support multiple matrix sizes
template<int SIZE>
void systolic_array(data_t A[SIZE][SIZE], data_t B[SIZE][SIZE], data_t C[SIZE][SIZE]) {
    #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

    data_t local_A[SIZE][SIZE];
    data_t local_B[SIZE][SIZE];
    #pragma HLS array_partition variable=local_A complete dim=0
    #pragma HLS array_partition variable=local_B complete dim=0

    // Local buffers to store intermediate results
    data_t buffer[SIZE][SIZE];
    #pragma HLS array_partition variable=buffer complete dim=0
```

```
    // Initialize buffers and C matrix
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            buffer[i][j] = 0;
            local_A[i][j] = A[i][j];
            local_B[i][j] = B[i][j];
        }
    }

    // Perform the systolic array multiplication
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            for (int k = 0; k < SIZE; k++) {
                #pragma HLS PIPELINE II=1
                buffer[i][j] += local_A[i][k] * local_B[k][j];
            }
            C[i][j] = buffer[i][j];
        }
    }
}

// Explicit template instantiations for 16x16, 32x32, and 64x64 matrix sizes
template void systolic_array<16>(data_t A[16][16], data_t B[16][16], data_t C[16][16]);
template void systolic_array<32>(data_t A[32][32], data_t B[32][32], data_t C[32][32]);
template void systolic_array<64>(data_t A[64][64], data_t B[64][64], data_t C[64][64]);
```

6.3. Testbench Adjustments for Larger Sizes

To test the systolic array multiplier with different sizes, update the testbench to initialize matrices of different sizes and call the systolic_array function with the appropriate template parameter.

tb_systolic_array.cpp

Here's a sample testbench that tests the 16x16, 32x32, and 64x64 cases:

```cpp
#include "systolic_array.h"
#include <iostream>
#include <cmath>

template<int SIZE>
bool check_result(data_t C[SIZE][SIZE], data_t expected[SIZE][SIZE]) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            if (std::abs(C[i][j] - expected[i][j]) > 0.01) { // Allow small tolerance
                return false;
            }
        }
    }
    return true;
}

template<int SIZE>
void run_test() {
    data_t A[SIZE][SIZE];
    data_t B[SIZE][SIZE];
    data_t C[SIZE][SIZE] = {0};
    data_t expected[SIZE][SIZE]; // Fill this with expected results based on A and B
```

```
    // Initialize A and B with sample data
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            A[i][j] = i + j;
            B[i][j] = i - j;
            expected[i][j] = 0; // Populate with precomputed expected results if available
        }
    }

    // Call the systolic array function
    systolic_array<SIZE>(A, B, C);

    // Display the result matrix C
    std::cout << "Result matrix C (" << SIZE << "x" << SIZE << "):\n";
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            std::cout << C[i][j] << " ";
        }
        std::cout << "\n";
    }
    // Check the results
    if (check_result<SIZE>(C, expected)) {
        std::cout << "Test passed for " << SIZE << "x" << SIZE << " matrix.\n";
    } else {
        std::cout << "Test failed for " << SIZE << "x" << SIZE << " matrix.\n";
    }
}

int main() {
    std::cout << "Testing 16x16 matrix multiplication:\n";
    run_test<16>();

    std::cout << "\nTesting 32x32 matrix multiplication:\n";
    run_test<32>();

    std::cout << "\nTesting 64x64 matrix multiplication:\n";
    run_test<64>();

    return 0;
}
```

V. Comparison with Other Multiplier Architectures

The systolic array multiplier offers several advantages in terms of performance and parallelism, but it also presents unique challenges. Here, we compare the systolic array multiplier with other common architectures and discuss its limitations and areas for potential optimization.

1.1. Comparison with Other Multiplier Architectures

a. Systolic Array Multiplier vs. Serial Multiplier

- **Performance**: Serial multipliers process bits sequentially, making them slower for larger inputs, as they require more cycles to complete a multiplication. In contrast, the systolic array multiplier performs multiple multiply-accumulate (MAC) operations concurrently across its processing elements (PEs), significantly reducing overall latency.
- **Parallelism**: The systolic array's parallel structure enables simultaneous operations across PEs, making it highly suitable for high-throughput applications like matrix computations and signal processing.

b. Systolic Array Multiplier vs. Array Multiplier

- **Structure**: Traditional array multipliers utilize a 2D grid structure but lack pipelining, requiring each element to complete its operation before the next can start.
- **Pipelining**: The systolic array is fully pipelined, allowing data to flow rhythmically through the array with minimal delays. This pipelining capability enables higher clock rates and makes systolic arrays more suitable for FPGA implementations compared to non-pipelined array multipliers.

c. Systolic Array Multiplier vs. Booth Multiplier

- **Efficiency**: Booth multipliers are optimized for signed numbers, reducing the number of additions required, which makes them efficient for larger bit-width operations. However, Booth multipliers are more complex and may not be as resource-efficient in FPGA applications as systolic arrays.
- **Application Focus**: While Booth multipliers are often used in general-purpose CPU architectures, systolic arrays are specialized for repeated matrix multiplications and excel in machine learning and DSP applications.

1.2. Limitations of the Systolic Array Multiplier

a. Resource Utilization:

- Systolic arrays require a significant number of logic elements and flip-flops, particularly when scaled to larger sizes (e.g., 16x16).

- While pipelining enhances performance, it also increases resource usage to manage data flow between PEs, especially if DSP blocks are not utilized.

b. *Scalability for Large Matrices:*
   - For very large matrices (e.g., 16x16 or 32x32), latency can still be considerable as data must propagate through multiple PEs sequentially, which can create bottlenecks.
   - Scaling a systolic array to handle very large matrices requires additional optimizations to maintain performance without a linear increase in resource usage.

c. *Fixed Architecture:*
   - Systolic arrays are typically designed for fixed matrix sizes, making it challenging to adapt them dynamically for variable matrix sizes without additional complexity. This lack of flexibility can limit their use in applications requiring adaptable matrix sizes.

1.3. Recommendations for Optimization in Vitis HLS

To enhance the performance and resource efficiency of the systolic array multiplier, several optimization techniques can be implemented in Vitis HLS:

a. *Loop Pipelining and Unrolling:*
   - **Loop Pipelining**: Apply #pragma HLS PIPELINE to pipeline inner loops, especially for the MAC operations, with an initiation interval (II) of 1 if possible. This ensures maximum throughput by allowing each PE to initiate a new operation every cycle.
   - **Loop Unrolling**: Unroll inner loops using #pragma HLS UNROLL for smaller matrix sizes to further reduce latency. Fully unrolling the loops can minimize the overhead of looping and improve data processing speed if resources allow.

b. *Efficient Use of DSP Blocks:*
   - Utilize DSP blocks for MAC operations by applying #pragma HLS RESOURCE to specify DSPs for multiplication tasks, thereby reducing LUT and FF usage.
   - DSPs are optimized for high-speed arithmetic and can significantly enhance performance, particularly for larger matrix sizes.

c. *Array Partitioning:*
   - Apply #pragma HLS ARRAY_PARTITION on matrices AAA and BBB to allow parallel access to multiple elements, reducing memory access bottlenecks. Array partitioning enables multiple PEs to access data simultaneously, further reducing latency.

d. Fixed-Point Arithmetic:

- To reduce resource usage, use fixed-point arithmetic (ap_fixed data types) instead of floating-point. This allows for precise control over bit widths, balancing resource usage and precision for applications that can tolerate lower precision.

e. Optimize Data Flow Between PEs:

- **Stream Data:** Use hls::stream objects to enable direct streaming of data between PEs, avoiding intermediate storage in memory, which reduces memory overhead and increases throughput.
- **Reduce Buffering:** Minimize local buffers in each PE by optimizing data flow so that PEs can operate directly on inputs without excessive intermediate storage, thus saving resources.

1.4. Future Directions and Enhancements

b. Scalability for Larger Matrices:

- Consider hierarchical systolic array architectures that combine multiple smaller arrays to handle large matrices efficiently. This structure can help mitigate latency and improve throughput without a proportional increase in resource usage.

c. Dynamic Array Size Support:

- Implement a configurable systolic array to support dynamic matrix sizes. Such flexibility, though requiring additional control logic, would optimize resource usage and adaptability for various matrix sizes.

d. Hybrid Architectures:

- Explore hybrid designs that combine the strengths of systolic arrays with other multipliers (e.g., Booth encoding for handling signed numbers). This approach allows for flexibility in applications with mixed requirements, leveraging the advantages of each multiplication technique.

Summary

The systolic array multiplier offers high-performance, parallel matrix multiplication suitable for FPGA-based applications like DSP and machine learning. Compared to serial, array, and Booth multipliers, it provides faster, more predictable performance for matrix operations but can face scalability challenges and high resource usage for large matrices. Optimizations in Vitis HLS—such as pipelining, DSP utilization, array partitioning, and fixed-point arithmetic—can further enhance the systolic array's efficiency. These adjustments make the systolic array multiplier a viable choice for high-throughput applications, particularly when resources are carefully managed and tailored to application-specific requirements.

VI. Conclusion

In this project, we explored the design, synthesis, and implementation of a systolic array multiplier, beginning with a 4x4 array and scaling it to 16x16, 32x32, and 64x64 configurations. Systolic arrays, with their parallel architecture and efficient data flow between processing elements, proved highly effective for matrix multiplication, achieving substantial improvements in performance and resource utilization. The initial 4x4 design demonstrated the feasibility of systolic arrays for FPGA deployment, providing low latency and minimal resource consumption. Scaling the design to handle larger matrices, however, introduced new considerations, such as memory partitioning, loop pipelining, and optimal resource allocation. Through these adjustments, we maintained efficient performance across different matrix sizes while balancing power, performance, and area (PPA) trade-offs. Compared to traditional multiplier architectures, the systolic array multiplier delivers a higher degree of parallelism, making it suitable for applications in machine learning, signal processing, and scientific computations. While it presents challenges in scalability and resource usage for very large matrices, Vitis HLS optimizations—such as loop unrolling, DSP usage, and array partitioning—help to address these issues, enabling efficient execution on FPGA platforms. Overall, this work validates the systolic array architecture's potential for high-throughput matrix operations in hardware, providing a foundation for future enhancements and applications that demand scalable and efficient matrix computation.