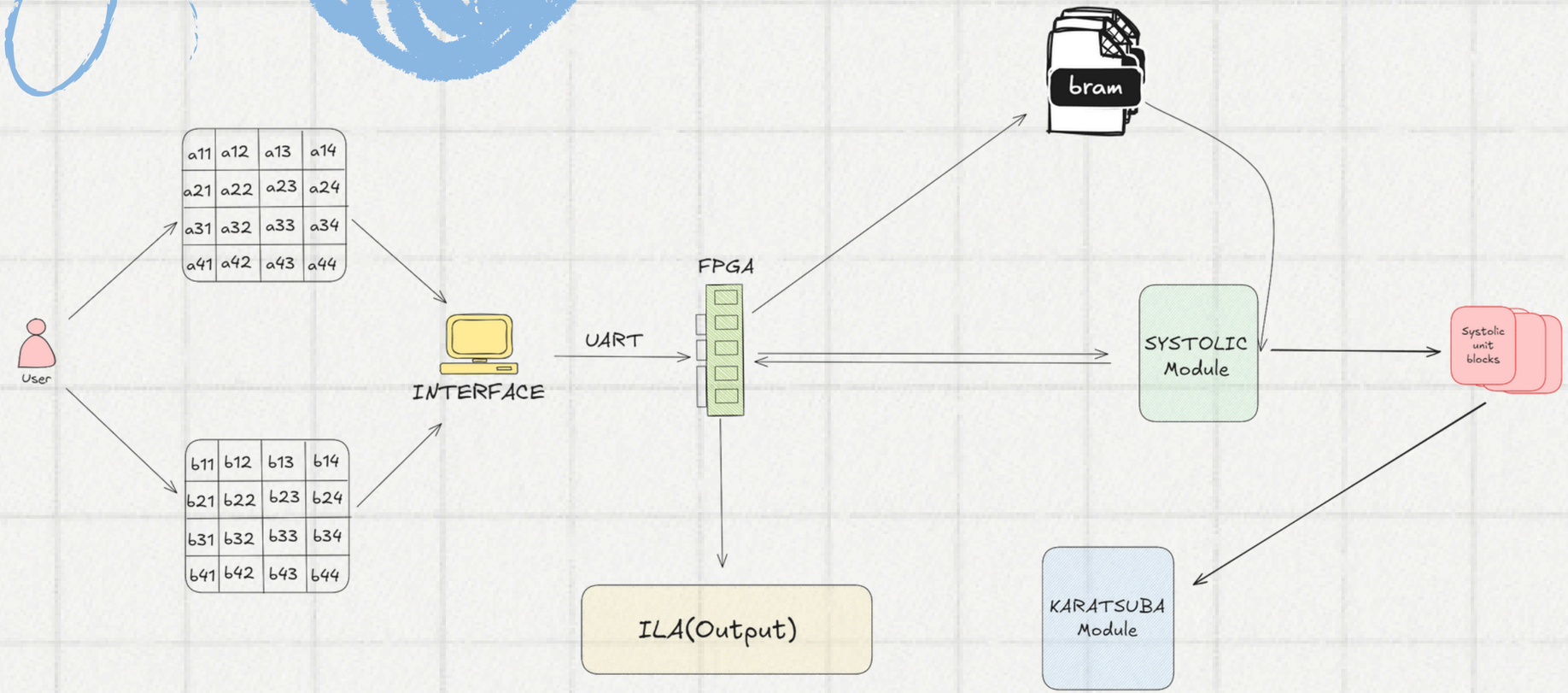
LDMMA

Large Dimension Matrix Multiplier accelerator

SOURCE CODE

DEMO VIDEO

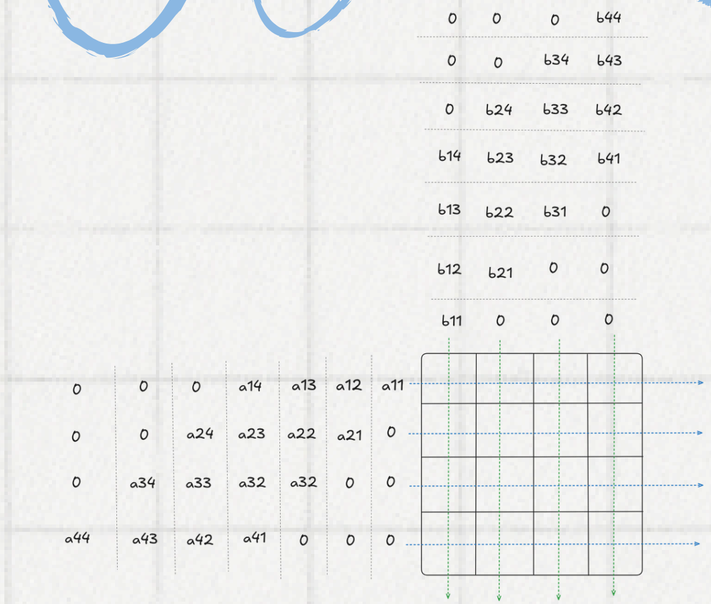


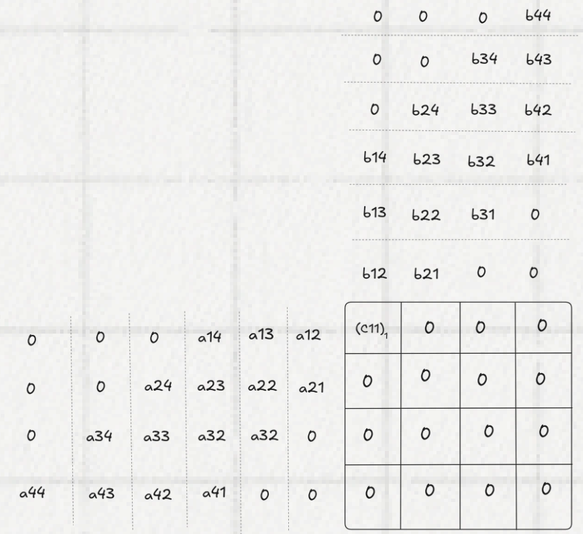
BRIEF DESCRIPTION

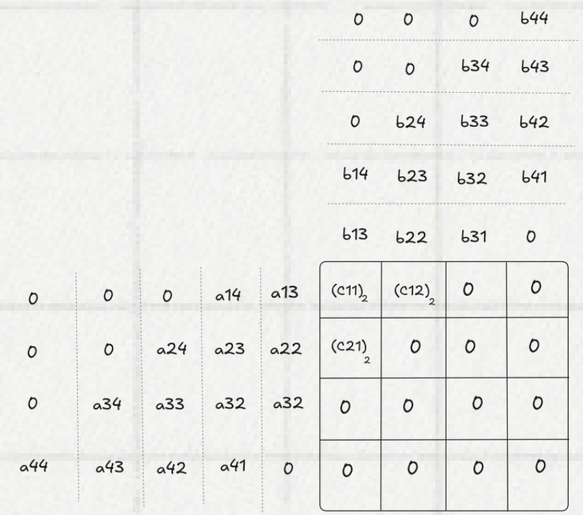
We implemented 4x4 matrix multiplication using a systolic architecture, with each matrix element represented by a 32-bit value. The matrix values are initially stored in block RAMs in a systolic pattern, and these values are fed into the systolic units, which serve as core blocks for MAC (Multiply-Accumulate) operations. For a 4x4 matrix, we have 16 systolic unit blocks.

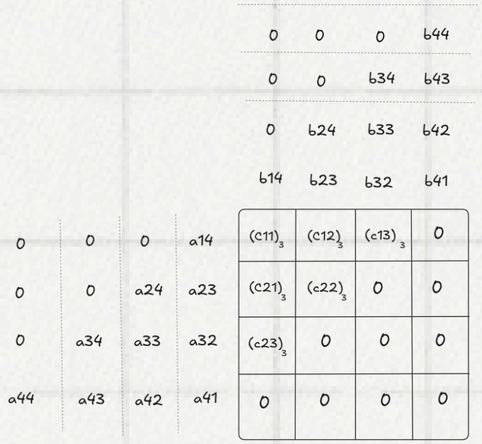
In the second part of the project, we replaced the standard multiplication in each systolic unit block with Karatsuba multiplication, enhancing the efficiency of the system. In the third part, we used UART to transfer data from a PC to the FPGA, enabling a Python script to send 4x4 matrices A and B. These matrices are stored in block RAM, after which systolic multiplication is performed.

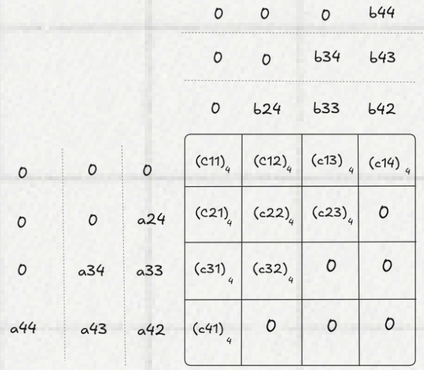
SYSTOLIC WORKING

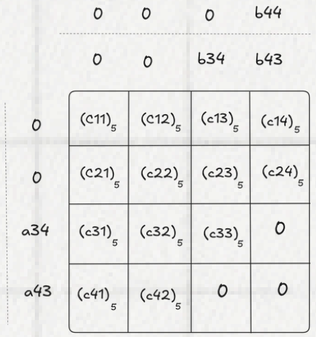


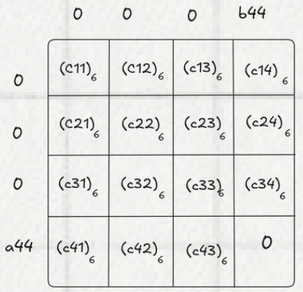


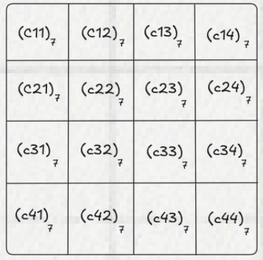












APPROACHES EXPLORED/TIMELINE

01

the systolic array was implemented using HLS, writing the code in C++ ,failed to achieve parallelism

02

Transiioned to implementing the systolic array using a state-based approach for better control over operations.

03

Attempted to extract repetitive inputs directly from Block RAM (BRAM). ,

04

Developed individual modules for each block of the final output, ensuring each input is taken only once

05

Tried implementing a generalized Karatsuba function using recursion, but Verilog's lack of recursion support presented challenges.

06

Tried implementing a generalized Karatsuba function using recursion, but Verilog's lack of recursion support presented challenges.

07

Following guidance, shifted to a 32-bit Karatsuba algorithm by dividing each number into two 16-bit parts.

08

Performed the multiplication using a 16-bit multiplier for each part and combined the results according to the Karatsuba algorithm.

09

Tried to send the matrix rows using uart without storing them in the BRAM's, leading to timing violation

10

finally,sending the matrix rows through the uart and storing them in the BRAM's,and eventually extracting them after few clock cycles

ISSUES FACED

When Sending the matrix rows through UART and extracting them directly without using any block-ram
,leading to negative timing slack.

(Check out SS-REPORT:PG-1)

When I load the first row of the matrix into the first row of the BRAM, the multiplication begins using values from the second row of the BRAM.

wrong test bench written,resulting in invalid outputs of the 2 cross 2 systolic output frame it properly

Wns error encountered on the block ram to systolic unit paths when single port RAM is used

(Check out SS-REPORT:PG-2)

Unable to Display All the Outputs On the Ila at Once

Missing /Invalid/Garbage values in the Outputs in The ILA

RESULTS

Max Clock frequency and Throughput

Max Clock Frequency=1/(T-Tslack)= 113MHZ

Max Throughput=
Max Clock Frequency/Latency (clock cycles) =12.55 MHZ

(Check out SS-REPORT:PG-3)

Resource Utilisation

(Check out SS-REPORT:PG-4)

Final Output

(Check out SS-REPORT:PG-5)

Power Consumption

(Check out SS-REPORT:PG-6)

Layout Diagram

(Check out SS-REPORT:PG-7&8)

Latency of computation

The end clock cycle minus start clock Cycle
=526-517=9Cycles

Latency(Clock Cycle)*(T-Tslack)
=79.9227(absolute latency)

(Check out SS-REPORT:PG-9)

COMPARE

Simple Matrix Multiplication	Ideally Takes 64 Clock cycles to Compute the Output,But Post implementation and after taking the inputs from the Block ram the number of clock Cycles Required Crosses 70
Systolic Matrix Multiplication	Post implementation the 4 * 4 matrix multiplication completes in approximately 9-10 clock cycles due to its reliance on parallelism and pipelining.
Systolic Matrix Multiplication with Karatsuba Multiplier	(Check out SS-REPORT:PG-10)
Systolic Matrix Multiplication with MAC	(Check out SS-REPORT:PG-11)

Links/References

- Github Repository (Click here)
- Systolic architecture (Click here)
- Karatsuba Algorithm (Click here)

Thank you