12/6/2023

Systolic Array Multiplier with VLS

Author: Jayanta Chowdhury
Tutor: Prof. Salim Ullah
Course: EHSD 2023

Table of Contents

List of Figures

List of Tables

1. Introduction

Recent industry requires multipliers which are efficient and fast for innumerable applications. There are various approaches to design a multiplier. One potential design is Systolic array-based multipliers. They have a very good property of data reuse and are faster than general purpose array multipliers. In this report we will discuss about the design, challenges as well as the performance of it on the hardware. Array multipliers could be floating point as well as integer type. Here we are focusing on the design aspect of integer multiplier. The code base for both Vitis HLS and Vivado can be found in the repo: https://github.com/jayanta1996/Systolic-array-multiplication

2. Architectural Design

The design of the multiplier is done in Vitis HLS IDE from Xilinx. Vitis HLS provides high level language support to design as well as to verify modules. It also supports directives called pragma which instructs the compiler how to synthesize a kernel source code to an RTL code. Export RTL feature helps to port the RTL code to an IP block which can be imported to Vivado IDE library to do further designs. The hardware used for the experiment is a FPGA called Xilinx Ultra 96v2 onboarding a chip part no: xczu3eg-sbva484-1-i. The IP block imported is then further connected to the Zynq ARM core to send and receive data easily. The interface chosen for the communication with the ARM core is AXI-Stream. For testing, we chose a simple Python script which will run on Jupiter notebook directly on the ARM core of the FPGA chip.

Here we explored different design options for the multiplier using pragmas to find the best way to implement the IP onto the FPGA board.

2.1. Systolic Array Organization

A systolic array works by flowing data from memory through an array of connected compute units, therefore reusing data read from memory. In our case, the matrix multiply operation, compute units are made of MAC units. By reusing data read from memory, this organization is much faster and efficient. *Figure 1* is the illustration of general purpose of matrix multiplication.

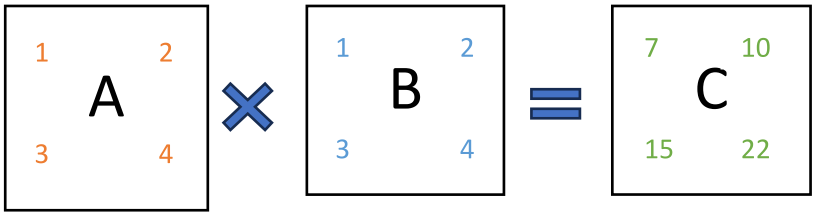


Figure 1 Matrix Multiplication

A first analysis of the systolic architecture shown on *Figure 2* shows us four compute units interconnected in a systolic array manner. Each compute unit takes two values, one from top and one from left and multiply them, storing the result in a built-in accumulator register. Notice that each compute unit will operate with the input values when they are both available. Then the data fed to the unit is passed onto the next unit.

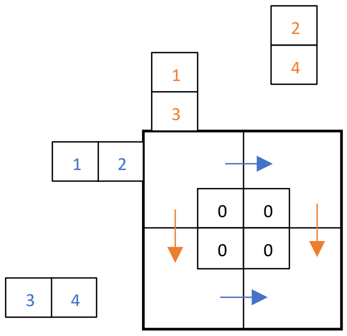


Figure 2 Systolic array initial scenario

The data flow is indicated by the colored arrows. Orange data is multiplied to the blue data on each unit and flows downwards, while the blue data will flow rightwards. The result of the performed multiplications is accumulated into each unit's memory as shown *Figure 3* . This process will be repeated with all data fed into the matrix and in a symmetrical process. This "symmetry" means that the same process is occurring on every compute unit in an identical manner, making conceptually the scaling of this units extremely simple. Larger size arrays mean further reusing data and reducing access to memory.

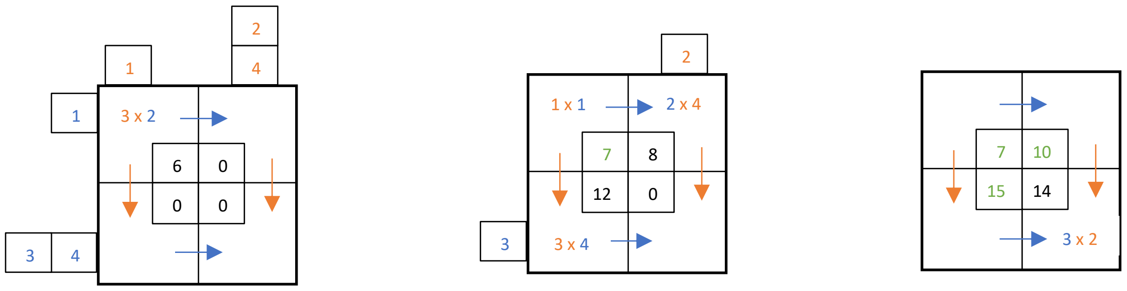


Figure 3 Systolic array steps

Finally, the data flowing outside the array could be fed on to another array or can be simply discarded.

2.2. Proposed Design

The main compute unit of the architecture is the MAC unit. Our MAC unit takes three inputs rather than traditional two input MACs named as "last", "aval" and "bval". "aval" and "bval" are the inputs fed from the systolic architecture orange and blue values as shown above whereas "last" gets the previously stored accumulated value. It is really important to instantiate the registers with "0" to avoid any garbage. It produces the output which is stored again in the local registers. This unit is instantiated multiple times to perform the complete array multiplication. Vitix HLS automatically create required Flipflops to control the data flow as described earlier.

The inputs and outputs are provided to the IP through AXI stream protocol with the help of inbuilt Zynq MPSoC. The use of inbuilt AXI template generates the required buffers for communication. The stream input is first stored in local matrices from where the data is fetched and provided to the systolic architecture. Similarly, the output is stored into a local matrix which is then converted to stream protocol.

Various pragmas are tested and different solutions are generated for comparison. Performance analysis is discussed in the next heading.

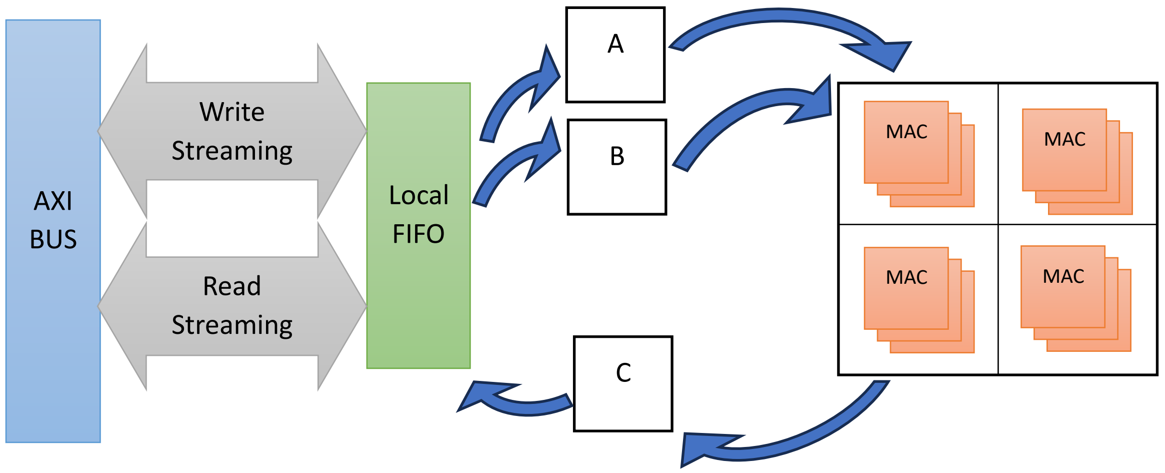


Figure 4 Complete design block diagram

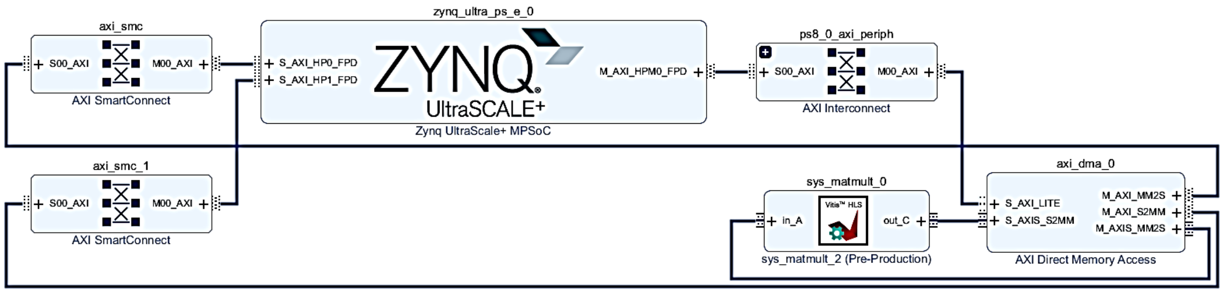


Figure 5 Interface view

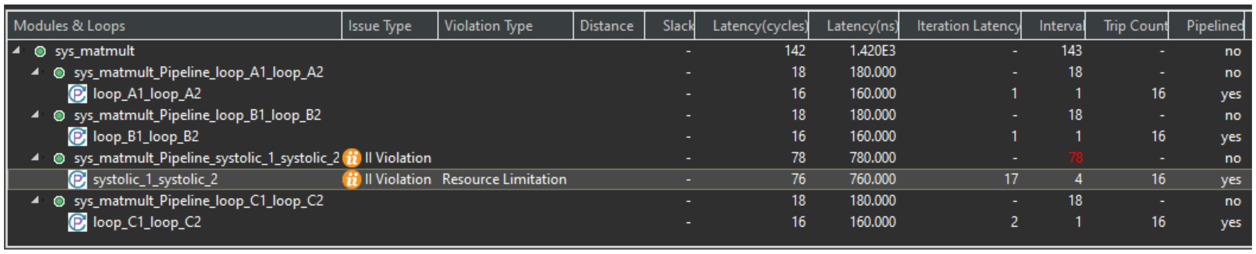
Figure 4 and *Figure 5* shows illustrations of the design from complete block diagram and Interface view. A, B and C are the matrices explained above. DMA is used here to relieve the Zynq MPSoC memory handling. Local memory access as well as local FIFO are managed by DMA. AXI SMC are automatically added in the design to avoid any packet loss or clock gating issues during data transfer.

3. Performance Evaluation

In this section we will discuss the performance of our systolic array IP module. For most of the analysis we used Vitis HLS platform and its inbuilt analysis tool. We tried to play with different pragmas to create multiple solutions in terms of resource utilization, latency and max frequency obtained. This allows us to choose the best solution and export it finally to an IP so that the functionality can be further verified through Vivado IDE.

3.1. Solution 1

In this variant no pragmas are used. It can be observed that input matrices are instantiated within the local multiport ROM memory and the output matrix is within a local dual port RAM. The hotspots of the code are understood and they are exactly the loops and the systolic architecture that need to be optimized to get the best performance.



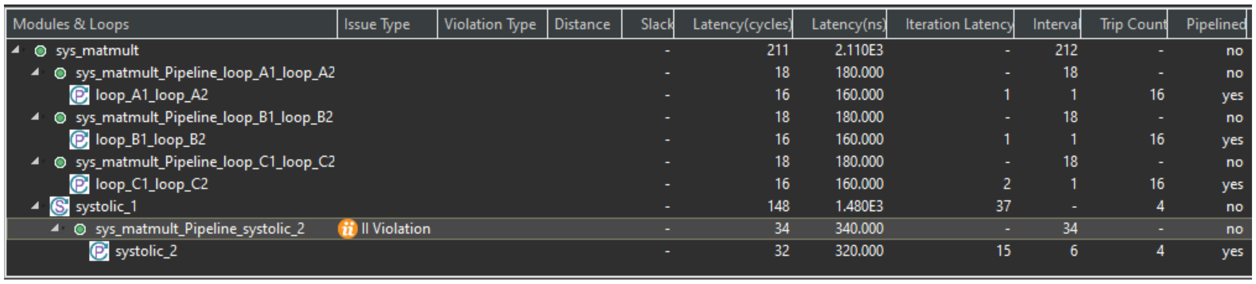
Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined
sys_matmult				-	142	1.420E3	-	143	-	no
sys_matmult_Pipeline_loop_A1_loop_A2				-	18	180.000	-	18	-	no
loop_A1_loop_A2				-	16	160.000	1	1	16	yes
sys_matmult_Pipeline_loop_B1_loop_B2				-	18	180.000	-	18	-	no
loop_B1_loop_B2				-	16	160.000	1	1	16	yes
sys_matmult_Pipeline_systolic_1_systolic_2	II Violation			-	78	780.000	-	78	-	no
systolic_1_systolic_2	II Violation	Resource Limitation		-	76	760.000	17	4	16	yes
sys_matmult_Pipeline_loop_C1_loop_C2				-	18	180.000	-	18	-	no
loop_C1_loop_C2				-	16	160.000	2	1	16	yes

Figure 6 Solution_1 loop performance

As a result, it can be observed that all input, output and systolic loops are flattened and pipelined but systolic loop received a violation of resource limitation. After further analysis the reason found to be the memory read dependency due to unavailability of enough ports. The details of the performance criteria are discussed on section Results.

3.2. Solution 2

In this variant we tried to solve the shortcomings from solution 1 and to improve the design. Here pragma *HLS ARRAY PARTITION* is used for all the input and output matrices. Input matrices A and B are partitioned in dimension 1 or row wise and dimension 2 or column wise respectively whereas Output matrix C is partitioned into individual elements with dimension 0. Hence there is no requirements of storing the data in the local memory but to directly fetch from the stream and send back as it is. We have added a pipeline pragma *HLS PIPELINE* to force pipeline under constraint to identify any improvements.



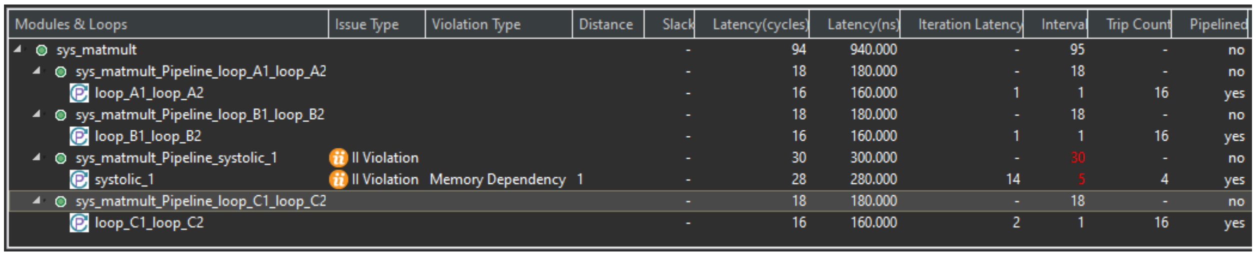
Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined
sys_matmult				-	211	2.110E3	-	212	-	no
sys_matmult_Pipeline_loop_A1_loop_A2				-	18	180.000	-	18	-	no
loop_A1_loop_A2				-	16	160.000	1	1	16	yes
sys_matmult_Pipeline_loop_B1_loop_B2				-	18	180.000	-	18	-	no
loop_B1_loop_B2				-	16	160.000	1	1	16	yes
sys_matmult_Pipeline_loop_C1_loop_C2				-	18	180.000	-	18	-	no
loop_C1_loop_C2				-	16	160.000	2	1	16	yes
systolic_1				-	148	1.480E3	37	-	4	no
sys_matmult_Pipeline_systolic_2	II Violation			-	34	340.000	-	34	-	no
systolic_2				-	32	320.000	15	6	4	yes

Figure 7 Solution_2 loop performance

We observe a violation of pipeline initiation interval that it cannot achieve provided constraint. The initiation interval is the number of cycles that must elapse between issuing two operations of a given type. The initiation interval required is 34 instead of 1 which definitely creates a scope for improvement.

3.3. Solution 3

This variant adds pragma *HLS LOOP TRIPCOUNT* in every loop which simply reports the latency and the loop iterations without impacting anything during synthesis. Pragma *HLS UNROLL* is used within the systolic architecture loops which creates multiple copies of the loop to perform parallel iterations if possible. Previous pragmas are removed except array partition.



Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined
sys_matmult				-	94	940.000	-	95	-	no
sys_matmult_Pipeline_loop_A1_loop_A2				-	18	180.000	-	18	-	no
loop_A1_loop_A2				-	16	160.000	1	1	16	yes
sys_matmult_Pipeline_loop_B1_loop_B2				-	18	180.000	-	18	-	no
loop_B1_loop_B2				-	16	160.000	1	1	16	yes
sys_matmult_Pipeline_systolic_1	II Violation			-	30	300.000	-	30	-	no
systolic_1	II Violation	Memory Dependency	1	-	28	280.000	14	5	4	yes
sys_matmult_Pipeline_loop_C1_loop_C2				-	18	180.000	-	18	-	no
loop_C1_loop_C2				-	16	160.000	2	1	16	yes

Figure 8 Solution_3 loop performance

It is observed that all the input and output matrix loops are flattened and pipelined. The violation still remains but the initiation interval reduced to 5 which is a significant improvement.

4. Results

In this section we will discuss the performance results under some criteria like latency, estimated timing achieved and resource utilization through the below given table

Solution Name	**Latency (cycles)**	**Estimated timing achieved (ns)**	**Resource utilization (%)**
Solution 1	142	6.919	2.2
Solution 2	211	7.207	5.1
Solution 3	94	7.207	7.1

Table 1 Criteria performance evaluation

Solution Name	**BRAM av. 432**	**DSP av. 360**	**FF av. 141120**	**LUT av. 70560**
Solution 1	8	14	2231	2421
Solution 2	0	14	6178	4667
Solution 3	0	56	8769	6264

Table 2 Detailed Resource utilization

Table 1 and *Table 2* indicates the IP performance with respect to the decided criteria and detailed utilization of the HW resources. Solution 1 seems better but it uses BRAM within the chip which is not usually a good practice also it achieves less timing due to its high latency. Solution 2 and Solution 3 achieves same timing but they differ in the resource utilization and latency. Solution 3 achieves less latency by sacrificing DSP element which is again a costly module within a chip. Hence the best solution can be decided based on the use cases. All solutions are exported as an IP and tested with the Zynq MPSoC and the result obtained is accurate.

5. Conclusions

In this project we target the domain of matrix multiplication with a Systolic array architecture as it is very famous for wide variety of applications. The design is vendor oriented since the acceleration of the design is mostly performed through the usage of DSPs which are vendor dependent. Communication protocol AXI Stream is chosen which is commonly used protocol to receive requests from the main processor and write data into memory. The design is configurable to any row and column of the input matrix but as the design is in Vitix HLS we need to import the RTL to Vivado for application.

Even though the current design can be further improved we successfully achieved our goal of keeping the granularity to the PE level. MAC unit is working as the PE of the complete design. The design can be further exploited to work with tiling methodology which are currently a hot topic within many researchers as it opens the door of large matrix multiplication with very less resources.

APPENDIX

- FPGA: Field Programmable Gate Array
- ARM: Advanced Risc Machine
- IP: Intellectual Property
- MPSoC: Multi Processor System on Chip
- AXI: Advanced Extensible Interface
- DMA: Direct Access Memory
- SMC: Smart Connect
- RTL: Register Transfer Level
- HDL: Hardware Description Language
- RAM: Random Access Memory
- ROM: Read only Memory
- FIFO: First in First out
- MAC: Multiply Accumulate
- HLS: High Level Synthesis
- DSP: Digital Signal Processor
- PE: Processing Engine
- HW: Hardware
- IDE: Integrated Development Environment