Implementation and Resource Analysis of a Systolic Array in Verilog

Mathew Prabakar
Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, USA
mprabakar3@gatech.edu

***Abstract*— Machine learning and AI have seen tremendous progress in recent years, ushering the need for dedicated computing hardware to render them useful in solving real world problems. Fundamental functions such as matrix multiplication and convolution, are computationally intensive and require several cycles on modern processor and GPUs. This gives us the necessity to investigate methods to offload such intensive tasks to special purpose hardware that can perform them in both a computationally and energy efficient manner. Systolic arrays provide a methodology to map intensive computations to hardware structures. These structures are exceptionally good at convolution and matrix multiplication and provide high throughput as it can process them in parallel. This project aims to implement as Systolic array for convolution in Verilog, analyze its resource utilization and compare it against similar arrays implemented in Bluespec Verilog.**

Keywords—Systolic Array; Convolutional Neural Net; Verilog; Machine Learning; AI; Resource Analysis

I. INTRODUCTION

The recent popularity of deep learning, specifically deep convolutional neural networks (CNNs),is strongly attributed to its ability to exceed human level performance in tasks such as image recognition, speech recognition and strategy development in games . These state-of-the-art CNNs are orders of magnitude larger than those used in the 1990s, requiring up to hundreds of megabytes for filter weight storage and 30k-600k operations per input pixel. One of the most important reasons for up recent uprising of machine learning is due to the computing capabilities of modern computers. Machine learning training requires massive, repetitive computations on very large data sets which proved infeasible in the late 1900's and was a stumbling block to AI development.

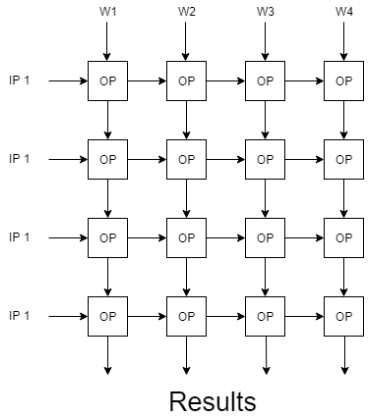

Fig.1 An Output Stationary Systolic Array

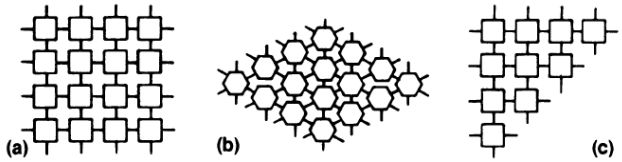

Fig.2 Types of Systolic Arrays

Modern day GPUs and high-performance processors have once again opened the doors for Machine learning research and are slowly enabling AI to solve real-world problems. Further progress in this direction requires dedicated hardware to accelerate learning and inference for real-time applications and this is where systolic arrays excel. Programming a systolic array to perform multiple functions is not a daunting task, but due to the restricted set of algorithms that can be implemented on them, systolic arrays are only used for special purpose applications.

II. BACKGROUND

A. Systolic Arrays

A systolic array is a grid of Processing elements (PE) that rhythmically process and pass data through the system. These are heavily pipelined machines capable of performing some simple operations and can speed up compute bound tasks in an inexpensive manner without increasing memory bandwidth requirements. Information in a systolic array flows between cells in a pipelined fashion and communication with the external world only happens at the boundaries of the array. The goal of the systolic array is to ensure that once data is brought in from memory, it is used effectively in multiple computations before leaving the array.

There are 3 types of systolic arrays as described in Fig.2, Type R, Type H and Type T. We will be focusing on the type R systolic array as it is the most favorable for output stationary convolution.

B. Convolutional Neural Network

A convolutional neural network (CNN) is constructed by stacking multiple computation layers as a directed acyclic graph. Through the computation of each layer, a higher-level abstraction of the input data, called a feature map (fmap), is extracted to preserve essential yet unique information at every

stage. Each stage is generally composed of an input feature map(ifmap) and several weight matrices. These matrices are convolved over the input feature map to extract the output feature map. In addition, a bias vector is added to the filtering results.

$$\mathbf{O}[z][u][x][y] = \mathbf{B}[u] + \sum_{k=0}^{C-1} \sum_{i=0}^{R-1} \sum_{j=0}^{R-1} \mathbf{I}[z][k][Ux+i][Uy+j] \times \mathbf{W}[u][k][i][j],$$
$$0 \le z < N, 0 \le u < M, 0 \le x,y < E, E = (H-R+U)/U. \quad (1)$$

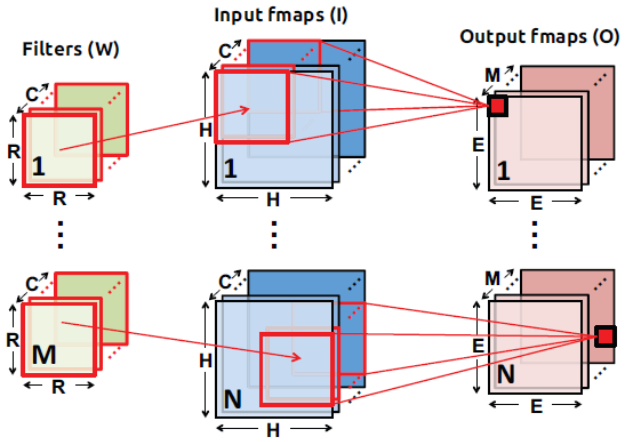

Fig.3 Operation of Convolution

O, **I**, **W** and **B** are the matrices of the ofmaps, ifmaps, filters and biases, respectively. U is a given stride size. Fig. 3 shows a visualization of this computation (ignoring biases). Modern CNNs can achieve superior performance by employing a very deep hierarchy of layers.

Although the MAC operations in Eq. (1) can run at high parallelism, which greatly benefits throughput, it also creates two issues. First, naively reading inputs for all MACs directly from DRAM requires high bandwidth and incurs high energy consumption. Second, a significant amount of intermediate data, i.e., partial sums (psums), are generated by the parallel MACs simultaneously, which poses storage pressure and consumes additional memory R/W energy if not processed, i.e., accumulated, immediately.

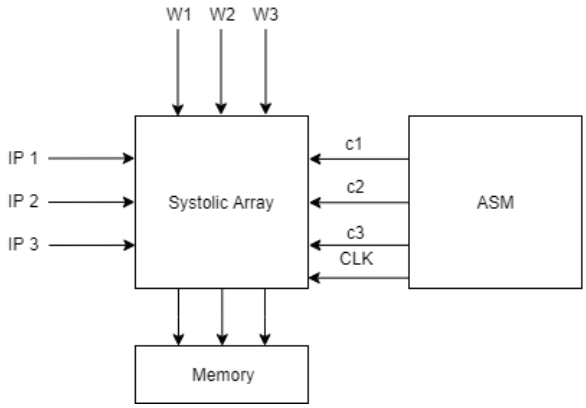

Fig.4 Block Diagram of the System

Fortunately, the first issue can be alleviated by exploiting different types of input data reuse:

- **convolutional reuse**: Due to the weight sharing property in convolution, a small amount of unique input data can be shared across many operations. Each filter weight is reused E2 times in the same ifmap plane, and each ifmap pixel, i.e., activation, is usually reused R2 times in the same filter plane.
- **filter reuse**: Each filter weight is further reused across the batch of N ifmaps.
- **ifmap reuse**: Each ifmap pixel is further reused across M filters (to generate the M output channels).

III. DESIGN

This section will elaborate on the design of the systolic array and its operational features. The size of the systolic array is parameterized and hence is decided at compile time. It can either be a square array, or any arbitrary sized array.

A. Processing Element

It has two inputs and two outputs. The values passed as the output can either be the result of the previous computation or the values at the input ports. c1 controls output mux 1 and c2 controls output mux 2.

At every clock, the data values present at the input are latched into the input registers after which they are multiplied and accumulated in the same cycle. Control signals and combinational logic decide between the input or the accumulator results. Bit width of each input and output is set to 16 bits.

B. ASM

The systolic array has a central state machine that asserts control signals which decide the output of each PE. It has counters to keep track of elapsed cycles and asserts the required control signals depending on its state.

It has two operational states:

- **Execute**:

  All the PEs multiply the inputs and accumulate the result. The values at the Inputs are passed on to the inputs of the next PE.

- **Transfer**:

  In the first clock, the value in the PE accumulator is pushed to the output port, in successive clocks, these values are sent to the boundary of the array through adjacent PEs and then eventually sent to global memory.

C. Calculataing Number of Cycles for Output Stationary

For a convolution of an M_x x M_y weight matrix over an XxY image matrix. The number of clock cycles taken by the systolic array can be computed as follows.

No of elements fed into First column of array = M_x*M_y
No of zeros inserted to meet correctness = M_y*(X-1)

No of zeros inserted because of pipeline delay = X-1
No of cycles to exit the systolic array due to pipelining = Y-1
Total Number of execution cycles =

$$M_x * M_y + (M_y+1)*(X-1) + Y-1$$

Where,
M_x = No of columns in Weight Matrix
M_y = No of rows in Weight Matrix
X = No of columns in Input Matrix
Y = No of rows in Input Matrix

There is some room for optimization as the initial and final few elements in our systolic array input columns are zeros due to zero padding.

An M*M square weight matrix will introduce (M-1) columns due to zero padding and as a result we will have M*(M-1) elements at the front and end of our inputs that we can ignore.
Optimized number of cycles = (M+1)*X + Y – 2

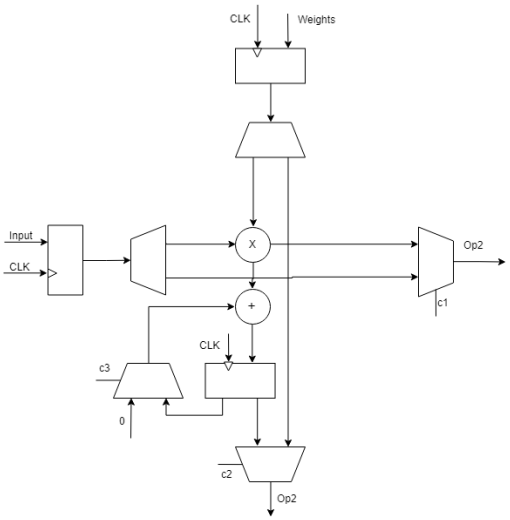

Fig.5 Component level diagram of a Processing Element for Output Stationary

D. Weight Stationary

The weight stationary implementation uses the same PE architecture, additional logic was added to the ASM to manage the loading and processing sequences. There were slight modifications to the overall arrangement of PEs. Each weight matrix would be unrolled and stored along the columns of the systolic array. Thus, for a single 3x3 filter, the systolic array would just be a single column with 9 rows.
Inputs are unrolled in such a way that the result of multiplication with the weights and the sum of the partial sums of each PE is on pixel of the output matrix. The weights are fed in along the column, 1 per cycle and are stored in the PEs. The inputs are fed in along the rows and pass through to adjacent columns if any.
The partial sums are added by a binary adder tree across every column. This makes the design somewhat like the MAERI architecture, where an augmented reduction tree is used to sum up all the partial sums in the neurons. Thus this implementation generates one output pixel result every cycle after the weights have been initialized.

The number of cycles for the computation of a weight stationary systolic array is straight forward. Initializing the PEs with weights takes $M_x * M_y$ cycles. Then it would take X*Y cycles to compute all the output pixels. The symbols have the same meanings as in the previous section.

E. FPGA Implementation

Several implementations have been tested on the FPGA and for ease of usability and for achieving the goals of this project, the following implementation has been adopted for evaluation.
Input image and weights are initialized into the RAM modules on the FPGA during compile time. A pushbutton is used to start execution for a given set of inputs and weights. Onboard switches are used to index into the RAM where the results are stored. The results are then read from the RAM and converted into BCD which is then displayed on a 7-segment display.

The following were the alternate implementations explored.

- Using a UART interface to stream data from the PC to the FPGA, perform computation and stream the data back. The data would be stored on the DRAM modules of the FPGA instead of building memory out of logic resources.
- Integrating the Systolic array with SCALE-Sim, which is a cycle accurate CNN architecture simulator, to provided memory address traces that can be used by the Systolic array to compute results.
- Storing the weights of a simple CNN along with the input data onto the BRAM of the FPGA and then computing the result and storing it back in RAM.

IV. Limitations and Future Scope

A. UART Interface

The UART interface used to transfer the data from the PC to the FPGA was limited by the fact that only 8 bits of ASCI data could be sent at a time. This meant multi digit numbers could not be sent as one 8-bit number but only as multiple 8-bit digits, which means a protocol needs to be implemented using a state machine to accept multiple ASCI digits from the PC, on receiving a terminating character, combine the BCD values into binary and then store into RAM.

B. Fixed point vs Floating Point computations

Generally, the input data of real world neural nets are floating point. The systolic array implemented has a 16-bit data path (parameterized to be easily varied) this means that input data and weights must be converted into fixed point notation before being processed on the FPGA. Additional software support

would be necessary to handle this. Currently the systolic array only handles integers.

C. BRAM vs DRAM

The DRAM on the Nexys 4 DDR is accessible via a Memory interface generator and requires additional logic to translate signals from a RAM interface to a DDR interface. BRAM on the other hand can easily be generated using the BRAM generator Ip and instantiation using code.

D. Interfacing with SCALESim

SCALE-Sim provides memory traces for the systolic array and handles folding of large arrays to fit the network on hardware with limited resources. It poses a few hurdles since it considers the data to be present in a contiguous address space. For integrating this with the FPGA, three possible approaches exist.

1. Load all the input and weight data into DRAM and treat it as Data memory and then store the memory addresses provided by SCALE-Sim into an Instruction memory and use a co-processor to read instructions and fetch the corresponding data and then feed it into a FIFO which then feeds the data into the systolic array. This has a large utilization footprint and requires many clock cycles for computation.

2. Perform the address to data translation on the PC and then transfer the data to the FPGA into DRAM. This would require additional logic to condition the data into FIFOS before feeding into the systolic array.

3. Perform the address to data translation on the PC and modify the data transfer logic to automatically store the data into DRAM banks attached to each input of the systolic array, this would circumvent any logic needed to condition the data from memory to computation and would give the maximum performance. This might not be area efficient.

V. EVALUATIONS

The proposed design has been synthesized in Verilog and implemented on the Digilent Nexys 4 DDR. The systolic array was generated with 3 configurations, 4x4, 8x8 and 16x16. The simulations were performed in ModelSim and the synthesis in Vivado for the Nexys 4 Artix-7 FPGA Trainer Board with the Xilinx Artix-7 FPGA XC7A100T-1CSG324C.

The Bluespec Verilog (BSV) Systolic array could only be synthesized but not implemented due to insufficient IO pins in the target FPGA even for the smallest 2x2 design. As a result, no FPGA power metrics are provided for the BSV version.

The 16x16 Verilog design could not be implemented but only synthesized due to insufficient DSP blocks on the FPGA. As a result, max frequency and power estimates have been provided for a 15x15 Verilog design.

A. Output Stationary

The evaluations for the output stationary design are plotted below. Figure demonstrates the results at the output lines.

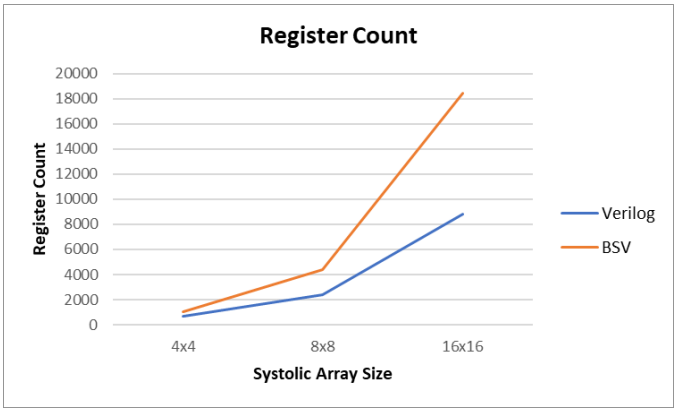

Fig.6 Register Utilization between Verilog and BSV implementation for different size of systolic arrays.

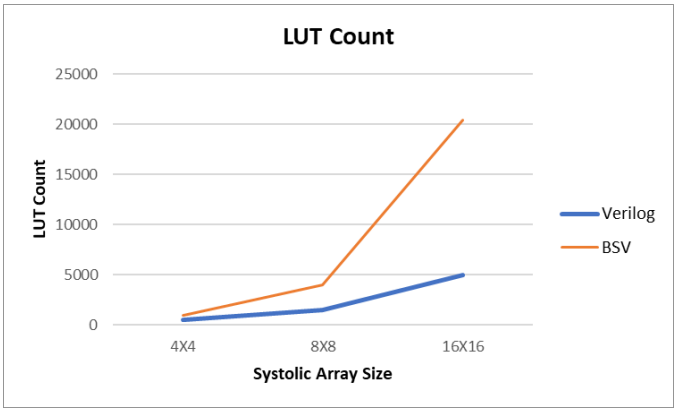

Fig.7 LUT utilization between Verilog and BSV implementation for different size of systolic arrays.

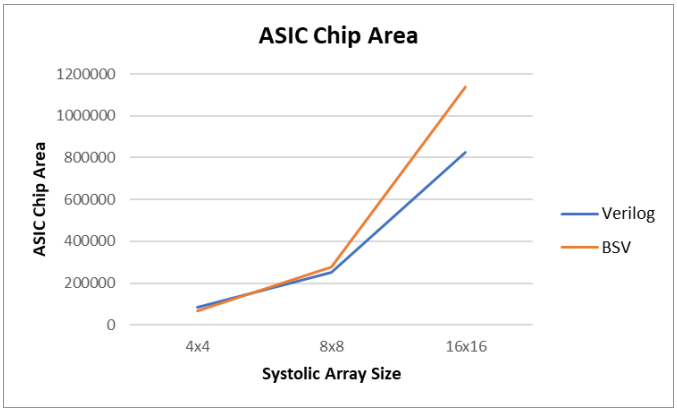

Fig.8 ASIC Chip area between Verilog and BSV implementation for different size of systolic arrays. Chip area is in units, where each unit is the area of the smallest buffer in the NanGate 15nm library.

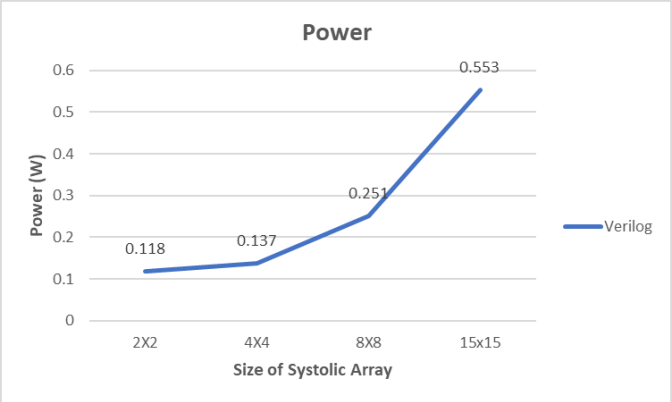

Fig.9 Power consumption of the design at maximum frequency for various systolic array sizes.

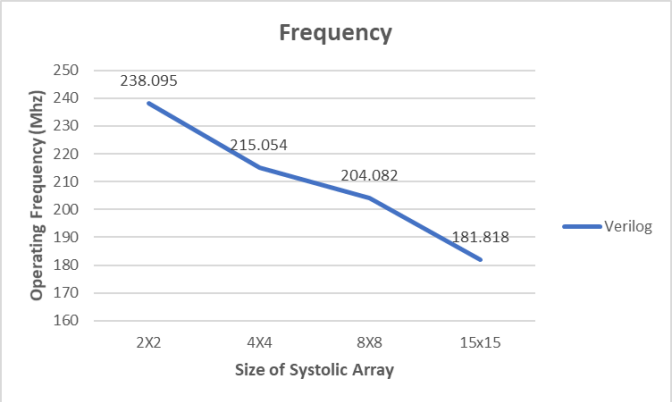

Fig.10 Maximum frequency of operation for various systolic array sizes.

B. Weight Stationary

The weight stationary designs were evaluated for 2 types of variations. First, by increasing filter count, thus adding columns to the array and the second is increasing filter size, by adding more rows to the array. The trade offs between the two variations are also analyzed.

The evaluations for the weight stationary designs are plotted below. Figure demonstrates the result at the output line.

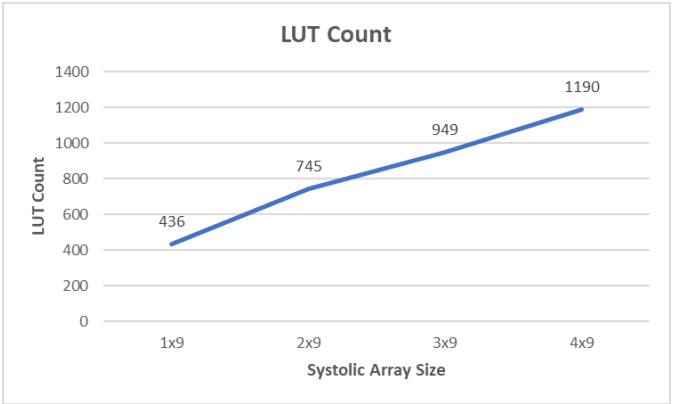

Fig.11 Variation in LUT count for increase in number of convolution filters.

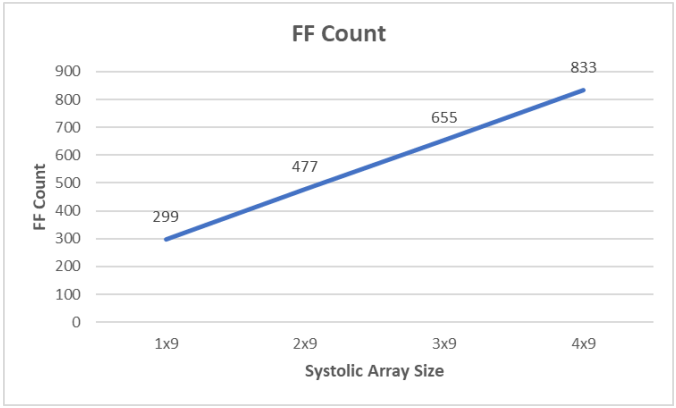

Fig.12 Variation in Flip Flop count for increase in number of convolution filters.

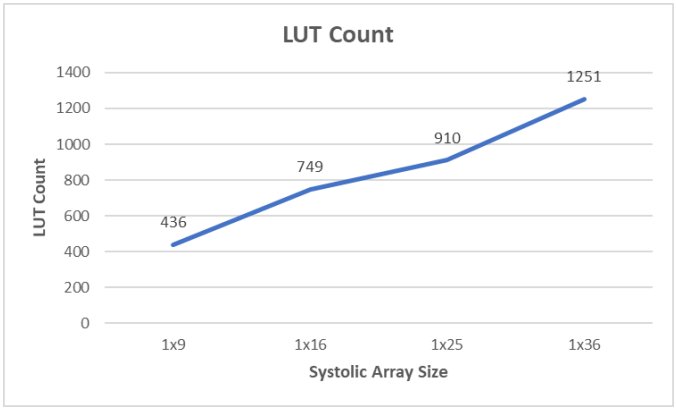

Fig.13 Variation in LUT count for increase in size of convolution filters

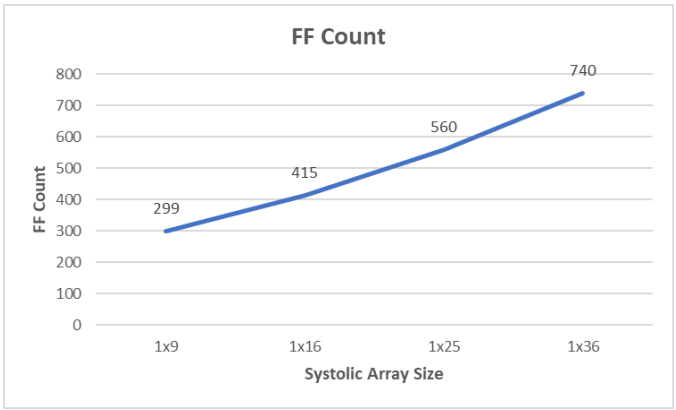

Fig.14 Variation in Flip Flop count for increase in number of convolution filters.

From the above evaluations, we observe that there is a tradeoff between area and performance. The output stationary implementation is more area efficient, takes several cycles to compute the result but operates at a higher frequency. The weight stationary implementation although requiring more area, generates a result every cycle owing to the binary adder tree, but operates at a slower clock. It is possible to bypass the adder tree and make the PEs generate the final sum by forwarding the partial sum to the adjacent PEs, but this would increase execution cycles, but it would also reduce the area

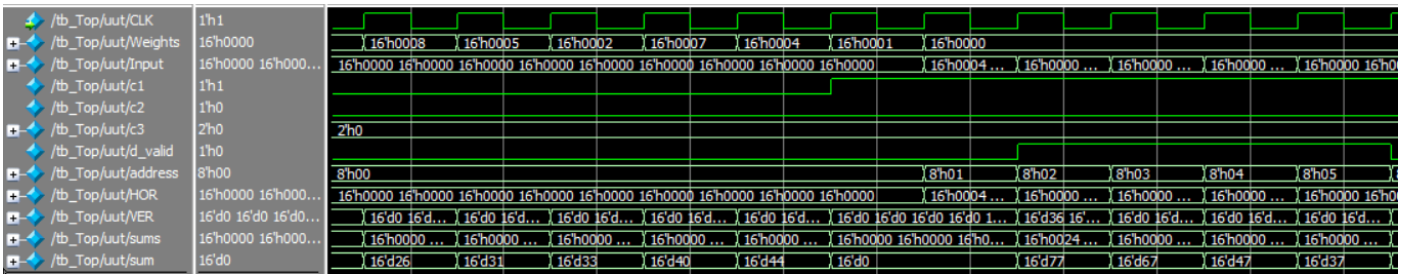

Fig.15 Waveform for functional simulation of a 1x9 weight stationary systolic array for a 2x2 input

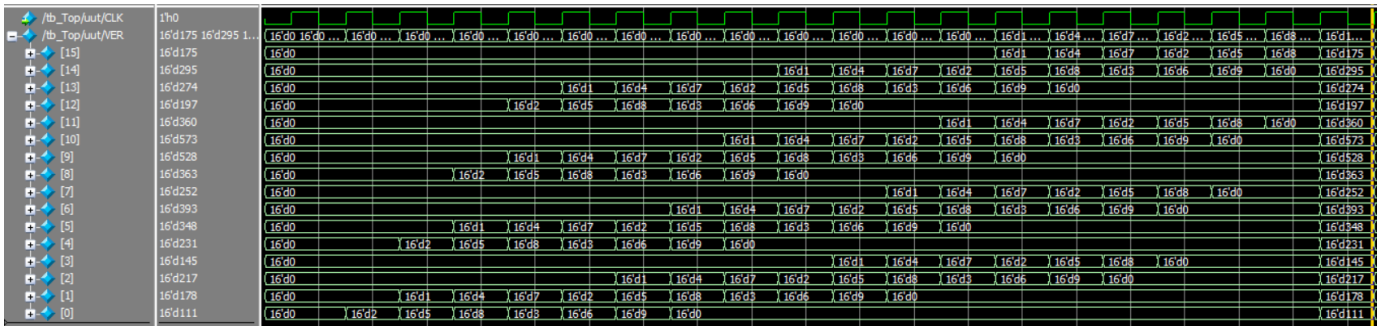

Fig.16 Waveform for functional simulation of a 4x4 output stationary systolic array

overhead of the binary adder tree. The observations for variations in parameter sizes are tabulated below.

Variation	Output Stationary	Weight Stationary
Increase in Input Size	Increase in size of array	Increase in Iteration count
Increase in Filter size	Increase in iteration count	Increase in length of array
Increase in Filter number	Increase in iteration count	Increase in width of array

Table 1. Effect of parameter variations on the systolic array resource utilization and performance.

VI. RELATED WORK

There are many interesting approaches to mapping DNN workloads to hardware accelerators for maximum performance and minimum area and power. One of which is MAERI, a DNN Accelerator architecture, made up of a collection of multiply and accumulation units connected by an augmented adder tree and a chubby distribution tree. Fundamentally, it is a more refined systolic array that is organized in the form of a tree, making it more flexible and efficient to map arbitrary data flows onto real hardware.

VII. CONCLUSION

This project implements a systolic array in two modes of operation and compares the area and power utilization with that of a systolic array generated in Bluespec Verilog for various sizes. The designs were also implemented on an FPGA to which weights and input could be sent through a PC over UART. This makes the designs flexible enough to be extended to process real world neural networks in the future.

REFERENCES

[1] Y. H. Chen, J. Emer and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), Seoul, 2016, pp. 367-379.

[2] W. Lu, G. Yan, J. Li, S. Gong, Y. Han and X. Li, "FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks," 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), Austin, TX, 2017, pp. 553-564.

[3] H. T. Kung "Systolic architectures, which permit multiple computations for each memory access, can speed execution of compute-bound problems without increasing I/O requirements."

[4] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. In Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18). ACM, New York, NY, USA, 461-475. DOI: https://doi.org/10.1145/3173162.3173176