Matrix Block Multiplication

Nicholas Teffandi*
*School of Electrical Engineering and Informatics, Bandung Institute of Technology, Indonesia
Email: 13221084@mahasiswa.itb.ac.id

***Abstract*—Matrix block multiplication is a matrices multiplication method for a large matrix. The idea is to break a larger matrix into the smallest matrix chunk called block unit. Since processing a large matrix in one go is expensive in terms of both hardware and time. It is more applicable to break down the matrix into chunks and proccess it in term of smaller chunks. This paper provides a report regarding modeling and implementing matrix block multiplication architecture. The model is verified using python script, where the output of those script is compared to the RTL model, which resulted in absolute maximum error of 2.36. This error occured due to the samll propagation error accumulated during calculation (the python result is calculated with 32 bit machine while the RTL is 16 bit design). Furthermore, final implementation metrics can be found on tabel I and the architecture can be clocked into maximum 125MHz.**

***Index Terms*—Matrix Block Multiplication, architecture, implementation**

I. Introduction

This section is a summary of what is and why matrix block multiplication is used, along with the chosen specification for the base design. The purpose of this section is to briefly give the reader a context of this paper. The full code can be found on this github link

A. What is matrix block multiplication?

Matrix block multiplication is a matrices multiplication method for a large matrix. The idea is to break a larger matrix into the smallest matrix chunk called block unit. A block unit is simply the smallest matrix size that we define for the matrix multiplication processing

B. why Matrix block Multiplication?

Since processing a large matrix in one go is expensive in terms of both hardware and time. It is more applicable to break down the matrix into chunks and proccess it in term of smaller chunks

C. Specification of base design

The proposed architecture will be built upon the following specification

- Input matrix size : 512x64
- Output matrix size : 512x512
- Unit block size : 4x4
- 16 bit fixed point (Q8.8 format)

II. Proposed Design

A. How does it works?

The main idea of the architecture is to stream the input data from RAM in terms of matrix chunk and operate it in a systolic array multiplier. The output of the systolic multiplier will recursively add itself to the buffer until all necessary chunks are operated. refer to 1 for an illustration of how matrix block multiplication is done. The figure shows multiplication of two matrices with sizes of 512 by 64 and 64 by 512 which resulted in a 512 by 512 matrix, the matrix is broken down into chunks of 4 by 4 where it is numbered accordingly. The red number highlights affected chunks that will be operated to produce a single chunk of output, for this case, the row chunks of first matrix and first coloumn of second matrix need to be operated via systolic array and an accumulator to produce a single chunk of output. This process will be iterated through until the output is fully constructed. A more detailed design will be addressed in the next sub section

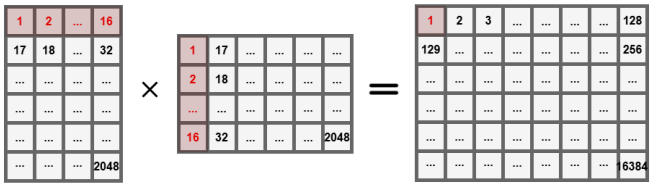

Fig. 1. matrix operation

B. Proposed Architecture

Figure 2 illustrates the proposed design. The design consists of the following item

- **RAM** : responsible for storing both input matrices and storing back the result.
- **Control Unit** : oversee and manage the dataflow for Data converter, systolic array, and adder buffer.
- **Systolic array** : multiply two matrix chunk
- **Data converter** : convert data from RAM into systolic array input by re-arranging its order and zero padding to synchronize the data
- **Adder buffer** : accumulator of matrix chunk. This block will iterate its function until it reaches 64 iterations before sending its output to RAM

The operation can be illustrated as the following step

1) RAM transfers 2 matrix chunks to data converter.
2) Data converter re-arrange the data and pads zero to the data accordingly to drive the systolic array

3) Systolic array is driven and produces the result which will be passed into the adder buffer
4) The adder buffer receives systolic output and then add it with its buffer content to update its buffer
5) Repeat the process from step 1 until it completed 64 iterations of matrix chunk operations to produce a single output chunk
6) the final result of the chunk will be sent to RAM, and the process will be repeated until all output chunk is produced

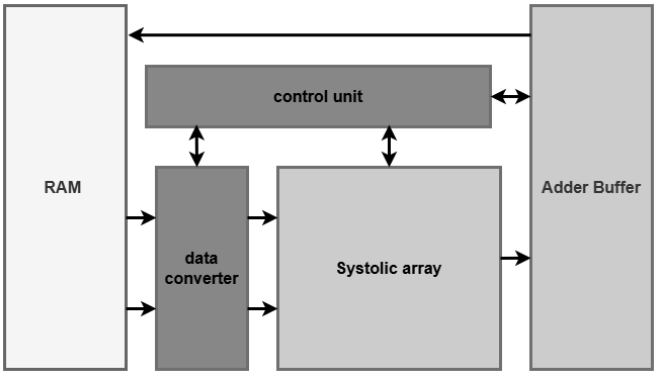

Fig. 2. Top Level Architecture

III. Implementation and discussion

this section will discuss the implementation of the proposed model in RTL, Verification, and synthesis result

A. RTL model

this subsections will discuss RTL model for each sub block of the architecture.

RAM Model

the RAM is modeled as 2 memory element with depth of 2048 and width of 256 bit each. The depth and width is chosen to simplify the data management, where a single chunk of 4 by 4 matrix is stored into 256 bit data with a total of 2048 chunks per input matrix. The width 256 bits came from the number of elements in a single chunk, which is 16, and since the data is represented in 16 bit, a single chunk can be coded into 256 bit of data. Figure 4 shows how the data from both input matrix chunk is coded for both matrix input. Since the chunk is coded that way, we can just simply iterate the corresponding stack of chunks downward, for example, one could simply iterate from top address until 16th address to produce a the first output chunk. Figure 3 is an example of the RAM content for the first 8th chunk of input

Data Converter

the data converter is modeled as 4 muxes with a cycling selector for **each input**, the architecture for single set of data conversion can be seen in figure 5. Note that the zero padding is done by holding muxes reset accordingly, which forced the output of the mux to be zero for a certain period. The detailed behavior can be shown in the RTL simulation in figure 5 for the mux control and for the final behavior in figure 7, the input is the following array (in Q8.8 Hex)

$$\begin{bmatrix} 0000 & 0100 & 0200 & 0300 \\ 0400 & 0500 & 0600 & 0700 \\ 0800 & 0900 & 0100 & 0100 \\ 0100 & 0100 & 0400 & 0500 \end{bmatrix}$$

Systolic Array

The systolic array used is a standard 4 by 4 array systolic with 16 Processing Elements, figure 8 shows the simulation result. Where the input is highlighted as blue and the output is highlighted as yellow, notice that it took about 9 cycle to complete the systolic operation.

Adder Buffer

The adder buffer is simply an accumulator where the signal for updating the accumulator is based on systolic array finish flag. The signal for sending the result to RAM however, are 16 cycles of systolic array operations (16 chunks operation) to produce a single chunk of output.

Control Unit

The control unit are responsible for data management, where it controls the RAM addressing and select appropriate chunks address while controlling other blocks to prevent data hazard. It will iterate for 16384 cycle of producing single chunks to complete the whole output matrix, it begin by iterating matrix 2 input coloumn wise then traverse matrix 1 row wise,hence producing output with order of coloumn wise then row wise

B. Verification

To verify the whole design, we will be using a python script to generate the input and output sample, where the input will be randomly generated for each run and constrained such that the number did not exceed Q8.8 representation. The input will be converted into Q8.8 then it will be dumped into the RTL via a text document. The output of the RTL will be extracted in form of text document and converted back to decimals, the final result then will be compared using spreadsheets to obtain its maximum error. The comparison result only contains the first 4000 chunk output, the result of the comparison is the maximum absolute difference between is approximately **2.36** . These difference arise from the difference between bit resolution, as the python is ran in 32 bit machines while the RTL only operates on 16 bit. These difference propagates and hence produced an error, future improvement will tweak its bit representation such that it will supress those error. (note : the author are unable to run all 16834 chunk simulation due to time and computation resource constrain),

```
1   008300E5008B00FA0076004600F200AC00B5003D006A00FF000600CD009B0028
2   00DE0050009E00700028007000 4F001E004A0060006900C7006C003E00A40040
3   002800A100F00011003F001B002700B900B20052008300B5007400E20054002F
4   00C400DD00CF003100F30053002A00450063009000690003F004900B600CC007B
5   0041004E0015005B00420024009D0074000C00AE00C000DB00750026001A00F7
6   006D00A500E2006700AB0017009D0086002B00180030005900980013008A00F9
7   0053009A008F008000BB007F0035009B00FB00110033009400AF00F900D900F0
8   00A4008600B30083009D00E4003D007200EA00B2001F00A500A800F40027008C
```

Fig. 3. RAM Content example

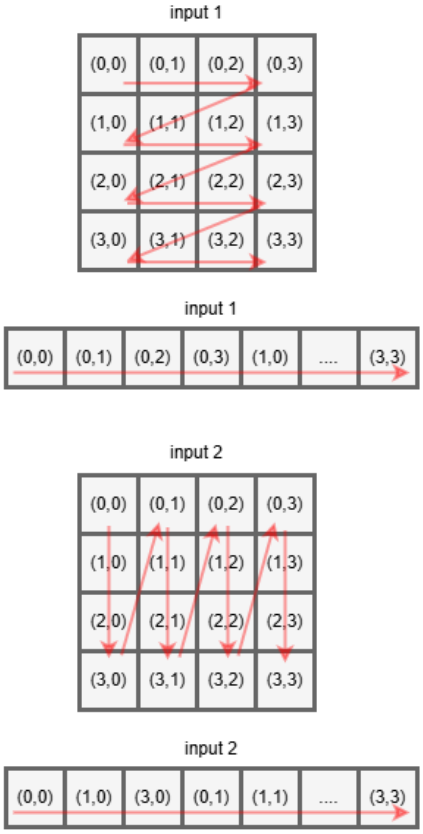

Fig. 4. RAM data structure

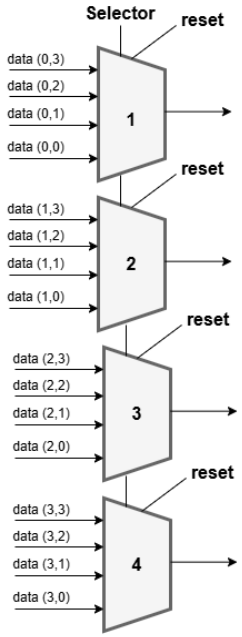

Fig. 5. data converter structure for one input

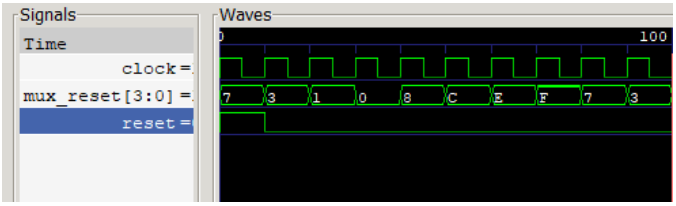

Fig. 6. Mux Reset Control

C. Synthesis Result

The RTL model (except RAM) is synthesized using Vivado with Pynq Z2 as the FPGA target.The RAM is not included because we are focusing on the core of the architecture (in the future the RAM will probably replaced with BRAM's), Table I is the utilization report generated from the synthesis. The slice can be approximated with $\#Slice \approx \frac{\#LUT}{8} + 51.2 \times \#DSPs$ for the 7 Series and $\#Slice \approx \frac{\#LUT}{8} + 51.2 \times \#DSPs$ for the UltraScale family of FPGA devices . Furthermore, the synthesis and implementation also generated a timing report, where the author set a constrain of **8ns** for the clock, with the result of worst negative slack of **0.516ns**, since the worst negative slack is positive and nearing zero, we can infer that all **timing is met** with 8ns clock, hence the **maximum frequency** for this implementation is around **125MHz**

IV. Optimization?

Due to the nature of systolic array and buffer which directly contributes to a faster clocking since the clock is constrained to the critical path between registers, the architecture is inherently optimized, since it has reached 8ns of clock. For context, a single adder and multiplier has approximately 3-4ns of path.

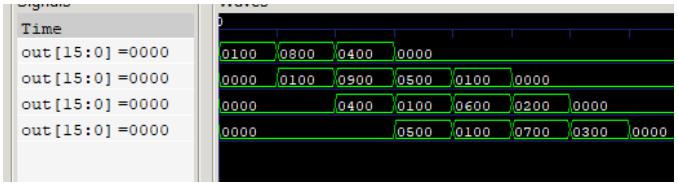

Fig. 7. data converter output (single output)

TABLE I
RESOURCE UTILIZATION REPORT

Resource	#Utilization	Utilization %
LUT	661	1.24%
FF	1317	1.24%
DSP	16	7.27%
Slice	902	-

V. CONCLUSION

This paper provides a report regarding modeling and implementing matrix block multiplication architecture. The model is verified using python script, where the output of those script is compared to the RTL model, which resulted in absolute maximum error of 2.36. This error occured due to the samll propagation error accumulated during calculation (the python result is calculated with 32 bit machine while the RTL is 16 bit design). Furthermore, final implementation metrics can be found on tabel I and the architecture can be clocked into maximum 125MHz.

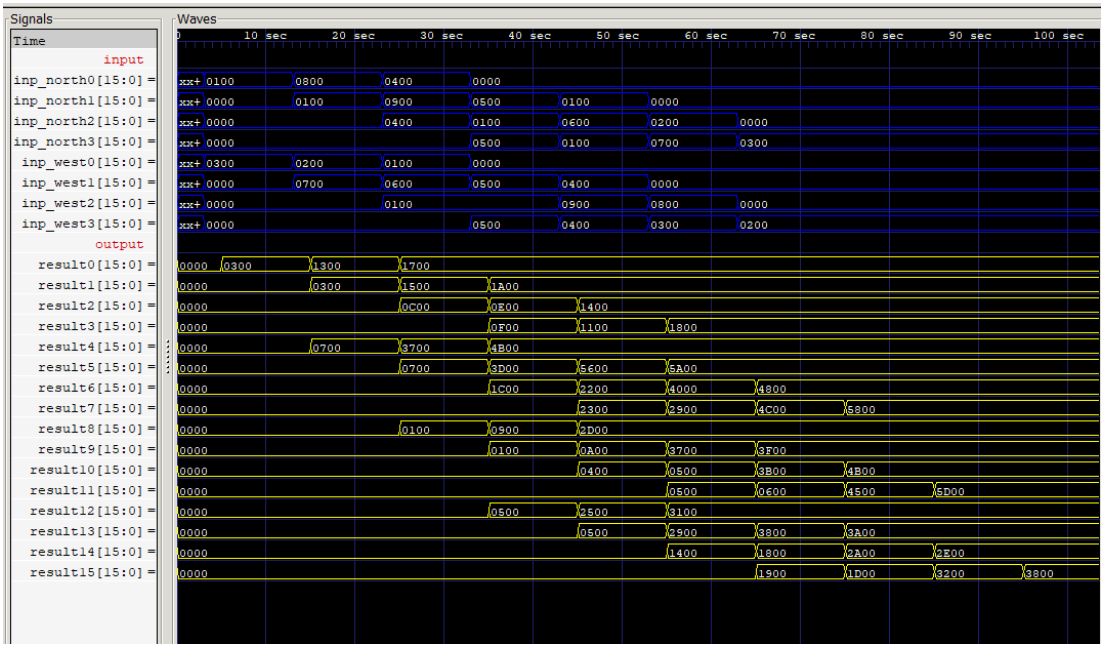

Fig. 8. Systolic RTL Simulation (blue is input and yellow is output)