Cycle-level Simulation of Tensor Processing Unit (TPU)

Anchal Sinha
Department of Electrical and Computer Engineering,
University of California, Los Angeles
Los Angeles, USA
anchalsinha@ucla.edu

Jacqueline Lam
Department of Electrical and Computer Engineering,
University of California, Los Angeles
Los Angeles, USA
jacqueline.lam902@gmail.com

***Abstract*— With the stagnation of Moore's Law, it has become increasingly more pertinent to use domain specific architectures such as the Tensor Processing Unit (TPU) that Google created for accelerating the computations involved with Deep Neural Networks (DNNs). The most important computation in a DNN involves matrix multiplication due to the ability to convert to it from convolution. Matrix multiplication is implemented via systolic arrays which are 2D structures composed of multiply accumulators (MACs) which process inputs and weights and pass that information to other MACs. The two most popular systolic array structures that perform matrix multiplication include the Non-Stationary Weight Systolic Array (NSSA) and the TPU-style Stationary Weight Systolic Array (TSSA). The architecture of both are simulated and their performance is compared, to which we see the TSSA significantly outperforms the NSSA.**

Keywords—Systolic array, matrix multiplication, TPU

I. INTRODUCTION

Moore's Law

Moore's Law states that as transistors in a dense integrated circuit (IC) doubles about every two years, processing power theoretically should also double. This principle has guided CPU architects over the decades, but limits to Moore's Law are being approached since there could only be so many transistors packed in a single chip. Mechanical, thermal limits, and power constraints such as electromigration, also are preventing the scaling of transistors even further [1]. Significant performance improvements must then occur in domain specific hardware accelerators (DSA) in order to compensate for the constraints. This was the motivation for Google to make a custom DSA called a Tensor Processing Unit (TPU) for deep neural networks (DNNs) in order to improve cost-performance over CPUs. Because the TPU is a DSA, it can drop features required by CPUs and GPUs that DNNs don't use. Such omissions make the TPU cheaper, save energy, and allow transistors to be repurposed for domain-specific optimizations.

Matrix Multiplication Optimization Using Systolic Arrays

One of the key components to the TPU's hardware architecture is their Matrix Multiply Unit (MMU) since it generalizes its hardware into a 2D structure in order to take advantage of the fact that DNNs perform numerous matrix multiplication operations and thus increase the number of operations per instruction significantly compared to a CPU which will is limited to perform one operation per instruction to make sure it is general purpose. Energy consumption also is decreased significantly by reusing fetched memory and register values [1]. The MMU consists of 256 x 256 chained Multiply Accumulators (MACs) that each perform partial sums of products at each cycle. There are two different systolic array implementations that we will explore to perform matrix multiplication which are the non-stationary (both operands of the matrices move through the array) and stationary architectures (one of the operands is stationary while the other passes through it). The TPU uses the latter and the implementations are explained in the next section.

II. BACKGROUND

For both systolic array architectures, there is a set number of MACs arranged in a 2D structure that each process inputs, but the difference lies in how the inputs are arranged and how each MAC processes the inputs.

Non-Stationary Systolic Array (NSSA)

In Fig. 1 [2], we can see that input matrices A (input/activation values) and B (weight input) are padded with zeros so that they can both enter and start computing in the systolic array at the same time. Matrix A is triangle padded with zeros on the right side, while matrix B is triangle padded on the bottom. After each cycle, the weights are shifted down once, and the inputs are shifted to the right.

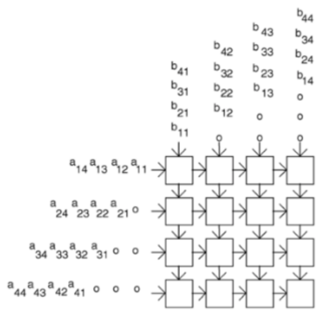
b_{41} b_{31} b_{21} b_{11}
b_{42} b_{32} b_{22} b_{12} o
b_{43} b_{33} b_{23} b_{13} o o
b_{44} b_{34} b_{24} b_{14} o o o
a_{14} a_{13} a_{12} a_{11}
a_{24} a_{23} a_{22} a_{21} o
a_{34} a_{33} a_{32} a_{31} o o
a_{44} a_{43} a_{42} a_{41} o o o

Fig. 1: Non-Stationary Systolic Array [2]

On the first cycle of execution, a11 and b11 will then be multiplied together and their partial sum is stored in the actual MAC itself at (1,1) and does not move. Then a11 is passed from the MAC at (1,1) to the MAC at (1,2), whereas the weight b11 is passed to the MAC at (2,1). On the next cycle, the MAC will add the previously stored partial sum to the product of b21 and a12. The MAC at (1,2) will then multiply the passed input a11 and b12 and store that as its partial sum. The MAC at (2,1) will multiple a21 with the passed weight b11 and store that. The process continues until all the elements from matrix A and B have entered the systolic array. When all the MACs are done processing A and B, the result matrix C is stored within the actual MACs themselves and need to be sent out of the systolic array to process the next inputs.

TPU-style Stationary Systolic Array (TSSA)

One of the main differences between the TSSA and the NSA is that the weight input matrix B is not triangle padded since the weights are stationary (i.e. they stay in the same MAC). As shown in Fig. 2 [2], the input matrix A still moves to the right, and the partial sum between the input and the weight is passed downwards to the next MAC. For the first cycle a11 will be multiplied by b11 inside of the MAC at (1,1), the partial sum is then sent down to the MAC at (2,1), and the input a11 is sent to the MAC on the right at (1,2). In the next cycle, the MAC at (2,1) will sum the partial sum it got from the MAC at (1,1,) with the product of a12 and b21. The MAC at (1,2) will multiply b12 and a11, and the MAC at (1,1) will multiple a21 and b11. Once all the inputs are processed by the MACs, the outputs are then sent out from the last row of MACs in a diagonal fashion.

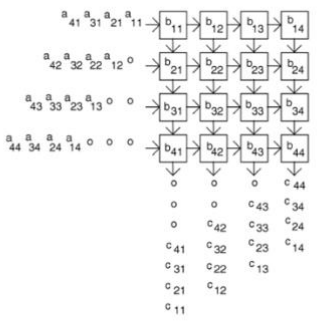

Fig. 2: Stationary Weight Systolic Array [2]

III. TPU Hardware Architecture

Our goal is to simulate the data movement between the modules of the TPU, which includes the TSSA. This does not include off-chip I/O such as interacting with the PCIe interface or implementing the ISA for TPUs. The main memory components we will be focusing on are the Weight FIFO (Weight Fetcher) which is an off-chip 8GB DRAM weight memory and is preloaded into Matrix Multiply Unit before inputs arrive and is shown in Fig. 3[1]; this can be thought of as Matrix B from the previous section. The Unified Buffer stores intermediate results that can serve as inputs to the Multiply Matrix Unit (MMU) and will go through the Systolic Array Setup for the data to be structured correctly and in case the input is larger than the actual MMU; this can be thought of as input Matrix A.

The output of the MMU is fed into the Accumulators which buffer the outputs before it can be fed into the Activation and Normalizing/Pooling unit.

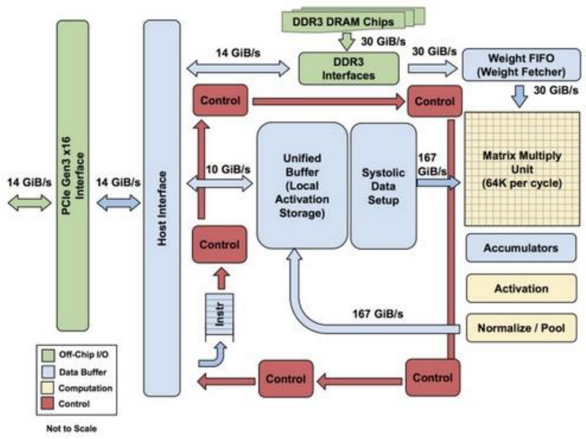

Fig. 3: TPU System Architecture [1]

IV. Simulation components

The TPU HW that was simulated include the Unified Buffer, MACs within the MMU, the MMU itself, the Weight FIFO, and the Accumulators.

MAC: To initialize the MAC, it needs the following:

- x, y: coordinates of the MAC within the MMU
- mac_left: reference to MAC located at (x-1, y) [activation]
- mac_up: reference to MAC located at (x, y-1) [partial sum]
- input_buffer: reference to unified buffer for MACs in first column to retrieve inputs

Each MAC can then retrieve the input_weight, input_partial_sum from mac_up and in input_activation from mac_left. After one cycle, the MAC will then compute the MAC's result_partial_sum given as: result_partial_sum = input_partial_sum + (input_weight * input_activation). The result_partial_sum can then be passed down to the next MAC to use.

MMU: To initialize the MMU, the number of rows and columns are required so that it could initialize the systolic array of MACs. The TPU has 256 x 256 MACs, so to simulate the TPU, we would use this as the input. The partial sums of the last row of the MMU are outputted to the accumulator and the inputs for the MMU are loaded from the unified buffer.

Accumulator: The accumulator aligns and stores the partial sum of the last row of the systolic array MMU. It can also perform add operations by resetting the 'cap' for each accumulator, which resets the index to start overlapping and adding successive values. To initialize the accumulator, we need to specify n_accumulators (the number of accumulators, equal to number of columns of the systolic array) and acc_size (the length of each accumulator).

The Unified Buffer: represents all inherent SRAM and systolic array setup elements. The Static RAM stores inputs and outputs, and performs the necessary conversions to set up the data into a format to feed into the systolic array MMU such as transposing and zero padding the input matrix in order to convert a convolution into matrix multiplication. The number of rows of the unified buffer also cannot be greater than the number of rows in the MMU as well.

*Storing new inputs***:** First, a region of SRAM is allocated for the MMU output to be stored and is particularly important when the output is tiled and needs to be combined after performing submatrix multiplications. While the SRAM stores the inputs unmodified, the systolic array buffer will store a list of queues with the proper input padding and transposed input. The offset is the padding of zeros between the current input and previous and is used to manage situations when the current input is larger than the previous input.

Storing outputs: Extract a region of the accumulators and store it in the correct region of the specified pre-allocated SRAM region.

Input that is smaller or the same as the size of the MMU will have to be transposed and padded before entering the MMU. As an example, we have a 3x3 input entering a 3x3 MMU in Fig 4.

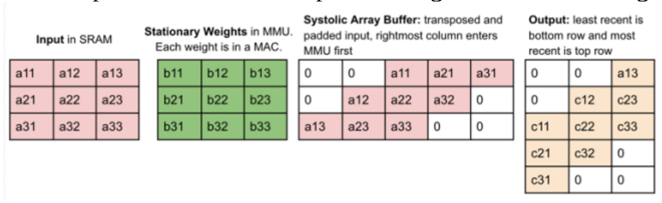
Input in SRAM
a11 a12 a13
a21 a22 a23
a31 a32 a33
Stationary Weights in MMU. Each weight is in a MAC.
b11 b12 b13
b21 b22 b23
b31 b32 b33
Systolic Array Buffer: transposed and padded input, rightmost column enters MMU first
0 0 a11 a21 a31
0 a12 a22 a32 0
a13 a23 a33 0 0
Output: least recent is bottom row and most recent is top row
0 0 a13
0 c12 c23
c11 c22 c33
c21 c32 0
c31 0 0

Fig 4. A 3x3 Input Transformation for Systolic Array Buffer

Weight FIFO: In order to decrease the latency of filling the MMU with stationary weights as the next input arrives, the weights are diagonally flooded in preparation. When we load weights, we start at the top left diagonal and load each diagonal (/) each cycle until it reaches the bottom right. While that is happening, inputs are also loaded in from the left and partial sums are computed. In cycle 4, since the MAC at (1,1) does not process any more input from Matrix A, it will start loading the new weights from Matrix Y and the new input from Matrix X.

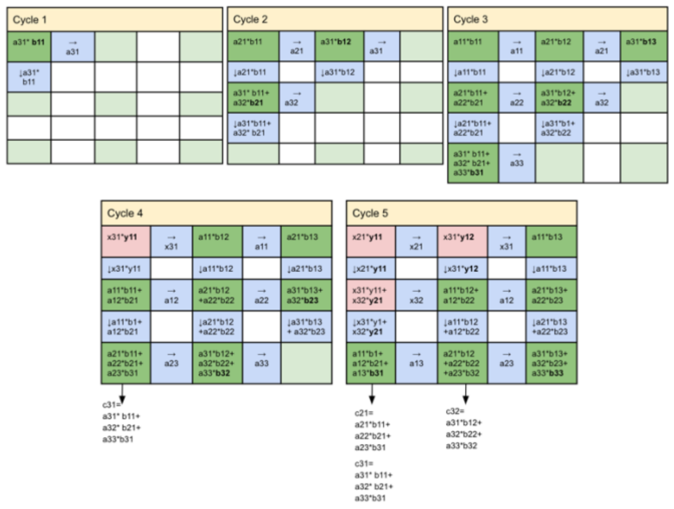

Fig. 5: Weight Flooding by Cycle

Fig. 5 shows the weight flooding mechanism by cycle. The light green boxes represent the MACs in the MMU that are not being utilized since there is no weight or input value. The dark green boxes indicate that the weight for that MAC has been filled with values from matrix B, and as the number of cycles increases, the more MACs are filled with weights in a diagonal fashion. The light red boxes indicate the next set of weights of matrix Y replacing B to be used for the next input matrix X. Contents of the MAC are the current partial sum result. The blue boxes represent the data that will be passed on to the next MAC. Right arrows pass the input from the current MAC to the MAC on the right and down arrows represent the partial sums that the MAC below the current one will use to compute its new partial sum.

V. Pipelining and Characterization

Systolic arrays are heavily pipelined, both at the individual MAC level and at the input level. At peak utilization, all MACs are performing an operation in parallel that will produce a corresponding output, and each cycle will allow for the movement and loading of data to continue the pipeline. However, the inputs can be pipelined as well, with the next input being fed to the systolic array in immediate succession to the first input. This requires careful coordination in the timing of weights loading and control of the accumulators. The TPU implements a weight-stationary systolic array, where the weights of each MAC are double-buffered [1] to prevent unnecessary cycles used to load weights without any computation. Since data in a systolic array follows a diagonal wavefront originating from the upper left MAC, we can pipeline the inputs and weights in a “flooding” pattern. Each MAC has no dependency on its source MACs once the previous cycle’s results are read, allowing those source MACs to perform computations independent of other computations. Inputs and weights follow that diagonal wavefront and “flood” in from the starting MAC in the upper left once the previous input row is available to receive input.

Pipelining allows for overlapped computation between successive matrix multiplications. The TPU performs matrix multiplication in the format:

$$X_{nxm} + Y_{mxp} = Z_{nxp} \tag{1}$$

with systolic array dimensions RxC. The number of cycles it takes for the TPU to fully compute the result and output the result to the accumulator is:

$$T_{TPU} = n + m + p - 1 + T_{offset} \tag{2}$$

There are two situations when this cycle count will encounter an offset. First, when the matrix dimensions are smaller than the systolic array dimensions, additional cycles are needed to propagate the results of the matrix multiplication to the accumulators, defined by:

$$T_{offset} = R - n - 1 \tag{3}$$

to account for the number of rows of the systolic array that must be traversed to reach the final row. Second, when inputs are pipelined, cycles from the current matrix multiplication perform computation on the next matrix multiplication, meaning the number of cycles for the next multiplication must account for those overlapped cycles.

$$-T_{offset} = n_{prev} + m_{prev} - 1 - T_{pad} \tag{4}$$

An additional padding offset is necessary to account for extra zero-padding to the matrix if the next matrix is larger. The padding prevents any data corruption from the larger matrix as the current smaller matrix is still being propogated through the systolic array, and is equivalent to the difference between the two inputs along the same dimension.

$$T_{pad} = m_{prev} - m \tag{5}$$

We can see that successive matrix multiplications require significantly fewer cycles to achieve the desired outputs due to the overlapped computations of the pipelined systolic array.

We can apply the same design methodology to the weight non-stationary systolic array (NSSA), but due to the inherent nature of its architecture, successive inputs are padded with zeros of the same size as the current input. The number of cycles to fully accumulate the output is:

$$T_{NSSA} = 2n + m + p - 2 + T_{offset} \tag{6}$$

Similar to the TPU, when the inputs and weights are pipelined, there are two situations where we must account for the offset. First, similar to the TPU, an offset is added to account for the propagation of results through unused rows of the systolic array, defined as:

$$T_{offset} = (R - n) * 2 \tag{7}$$

In addition, to account for overlapping computations due to pipelined inputs and weights, the offset is defined as:

$$-T_{offset} = m_{prev} + n_{prev} - 3 \tag{8}$$

VI. Testing and Evaulation

As previously mentioned, we simulated and validated the architecture at the cycle level. To provide the correct environment and context for the simulation, timing and control information needed to be manually performed, which would be done in the compiler in the general use case. Without a compiler and corresponding ISA implemented to process inputs to the systolic array, extract outputs from the accumulators, and …

To validate the TPU Simulator, the outputs of the MMU were compared against a ground truth matrix multiplication calculated using the numpy matmul() function. Weights and input matrices have randomly generated values as well. An assertion statement at the end of each test ensures that the ground truth and the output of the TPU simulator which is aligned at the end is equivalent. The following test cases are listed and shown in Fig. 6.

1. Testing Square Matrices of Various Sizes as Single Inputs: The simplest test case uses square input and weight matrices with the same size as the MMU. Another is with a square input matrix that is smaller than the MMU. One of the downsides of having a smaller input is that the partial sums will need to pass through unused MACs to reach the last row of MACs whose output is collected by the accumulator.

2. Testing Rectangular Matrices of Various Sizes as Single Inputs: The input might not be a square matrix and instead have a rectangular structure instead. An input matrix of n x m dimension, where m > n (columns > rows) can be fed into the MMU. Similarly, an input matrix where m < n can is also tested.

3. Testing Successive Pipelined Inputs and Weights: The first of the tests was 2 different and successive inputs using the same weight matrix. The next one was 2 different inputs and 2 different weights. The rest of the tests then test whether the current input was followed by a larger or smaller input and different weights corresponding to the input dimensions.

4. Tiling an Array Larger than the MMU: It is also possible for the input to be larger than the MMU which is 256x256, so this test makes sure that after tiling the array to be fed into the MMU with the proper structure, the output is still the same due to the accumulator.

This form of optimization would take place on the compiler side, allowing us to break up a single large matrix multiplication into a sequence of smaller matrix multiplications and additions using the divide-and-conquer technique.

A matrix multiplication follows the pattern of Fig. 5.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} aw+by & ax+bz \\ cw+dy & cx+dz \end{bmatrix}$$

Fig. 5: Matrix Multiplication

However, we can treat each individual value in the matrix as its own matrix, allowing us to divide an NxN matrix into four N/2 x N/2 matrices. This increases the computational cost, requiring 8 matrix multiplications and additions among the accumulators, but it allows us to perform seemingly large matrix multiplication on the fixed systolic array size.

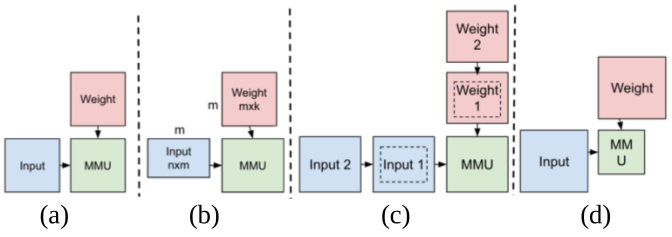

Fig. 6: Testing Various Inputs and Weights.

VII. Comparison between Non-Stationary and TPU(Stationary Weights) Systolic Array

The NSA was also implemented in order to compare the performances between the NSA and TPU. For an increasing number of pipelined matrix multiplications, the total cycle count was measured for each systolic array architecture. As shown in Fig. 7, the weight-stationary systolic array as implemented on the TPU performs much better than the non-stationary weights variant.

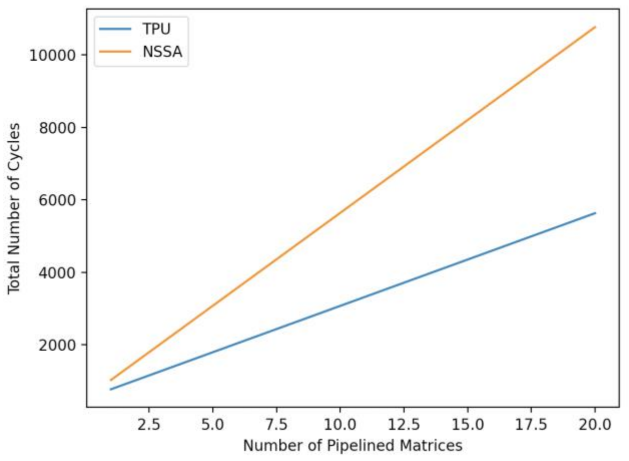

Fig. 7: Comparison of TPU and NSSA in Number of Pipelined Matrices vs Number of Cycles

VIII. Conclusion

As Moore's law becomes increasingly more difficult to uphold due to physical constraints, the need for domain specific architectures has become more popular such as the TPU that Google created for optimizing DNNs. The TPU aims to parallelize matrix multiplication through a stationary weight systolic array structure and by decreasing the amount of reads/ writes to memory per instruction. There are 2 popular systolic array architectures that exist, which is the non-stationary weight systolic array and the stationary weight systolic array (TPU implements this). Thus, we were able to simulate both by looking at the respective architectures. For the TPU, the MMU, Unified Buffer, Weight FIFO, and Accumulator were what we focused on implementing, where the Weight FIFO used a diagonal "flooding" mechanism. There were instances where behavior of the compiler had to be manually simulated such as when an input matrix was larger than the 2D systolic array and had to be tiled and attached together in the end or when inputs/weights were pipelined after each other of varying sizes in order to capture each possible scenario that the MMU could encounter. After simulating both structures and validating that the output of the MMU was correct, we could compare the performances between them. It was revealed that the TPU did much better in terms of cycles per pipelined matrices.

References

[1] N. P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), 2017, pp. 1-12

[2] Lim, Hyesook & Piuri, Vincenzo & Jr, Earl, "A serial-parallel architecture for two-dimensional Discrete Cosine and Inverse Discrete Cosine Transforms," IEEE Transactions on Computers, 2001, pp. 1297 – 1309