A Timing Simulator Implementation for Google Tensor Processing Unit

Geng Hexiang

Department of Electrical and Computer Engineering, Seoul National University

Abstraction

The increasing popularity of neural network researches and applications has induced a number of hardware accelerators. Among them, Google's Tensor Processing Unit (TPU) led a trend of using systolic array to accelerate matrix operations. This article describes a timing simulator of Google TPU (v1). The test results are shown in the result section and compared to those of TPU.

Introduction

In the past few years, the popularity of deep learning has massively increased, which can be seen in the word popularity of Google search, see **Figure 1**. It has been widely used in pattern recognition and many other areas.

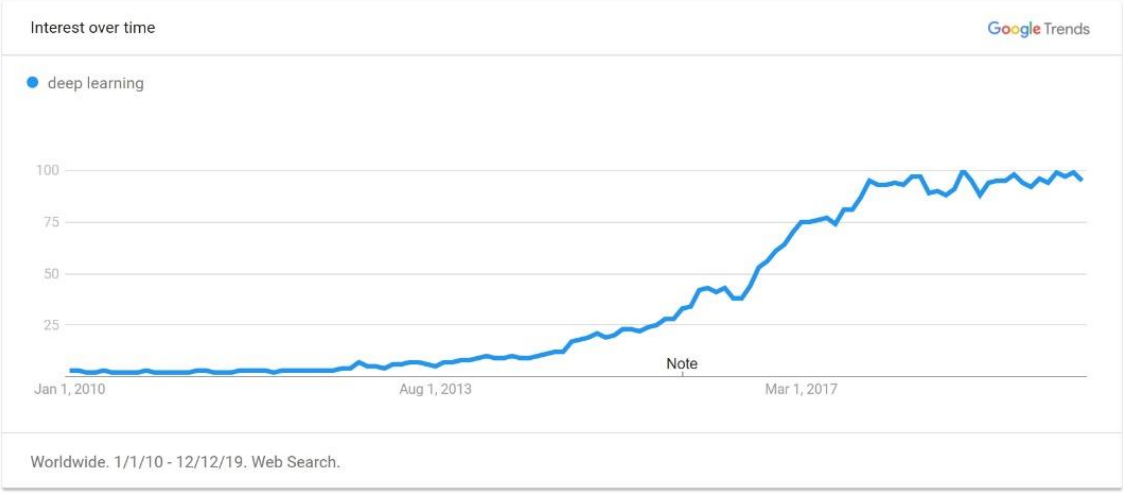

Figure 1. The search trend of the word "deep learning" on Google search engine. The "Note" marks 1/1/2016, when a system improvement was applied.[1]

Most of the implementations of deep learning technology are related to neural networks. Neural networks are inspired by the structure of a real brain. Multiple simple neurons are organized as networks and divided as layers. The input information will be abstracted while being passed through the layers. The details of the processing defer among design choices, but most popular ones during the running time require calculation of matrix multiplication and matrix convolution.

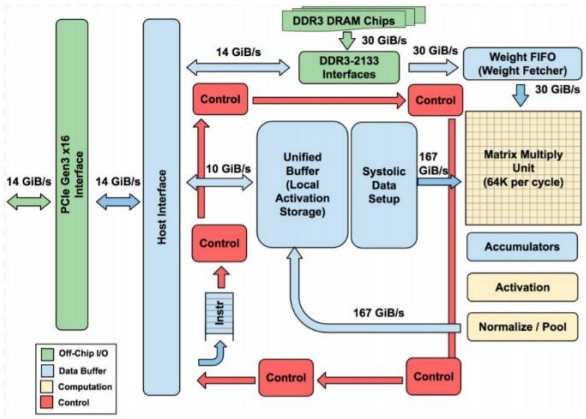

Figure 2. TPU Block Diagram. The main computation part is the yellow Matrix Multiply unit int the upper right corner.[2]

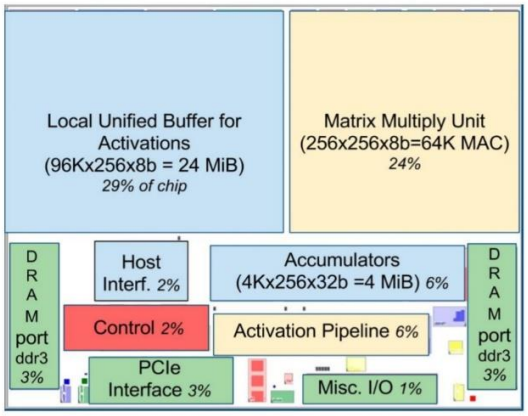

Figure 3. Floor Plan of TPU die. The shading follows Figure 2. Note that the control unit is just 2%, while is much larger in a CPU and GPU.[2]

Under this situation, there are many hardware accelerators designed for deep neural networks, one of the most popular one is Google Tensor Processing Unit (TPU), which is based on systolic array. A systolic array is a combination of simple cells connected to each other, and input data flow inside the array among cells, to reuse the input data and reduce the cost of data transferring, as a "pure" systolic array design will only has connection to other components on the sides[3]. The TPU's structure sees **Figure 2** and **Figure 3**. One of the design choices is that the cells in the matrix multiply unit can access a random position in the accumulators, to accumulate the temporary results.

In this article, a simple timing simulator for TPU will be introduced. The cycles required for computing certain sized matrix multiplication are tested and some different design choices will be compared.

Implementation

The implementation of the timing simulator referred to the block diagram of TPU and simplified some design details. It contains five main parts, as the systolic array, accumulators, unified buffer, control unit as well as simulation related parts.

The simulator uses GPU acceleration. Most of the memory used is allocated from the graphics card, and the computation are done on GPU using cuda cores.

Systolic array

The core of the simulator, same as that of TPU, is the systolic array. It has in total 256×256 cells. A cell's design sees **Figure 4**. Each cell in a cycle, will first take in the 8-bit data output by the upper cell and the left cell and multiply them together. Each cell contains a 16-bit storage to hold the product of two 8-bit input data. Then, the temporary result will be passed to the corresponding accumulator. The upper and left input data are latter passed to the lower and right cell, waiting to be used in the next cycle.

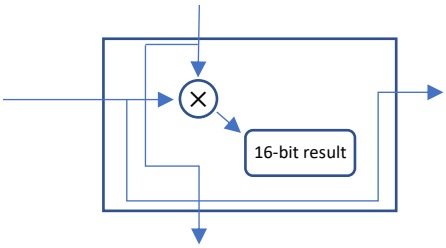

Figure 4. The simplified design diagram of a cell in the systolic array. The 16-bit product will be sent to a corresponding position in the accumulators.

Each cell is connected to four neighbors, except the ones on the sides. In this timing simulator, the cells on the upper side and the left side of the array are connected to the unified buffer. In each cycle, the cells on the two sides will get data from the unified data.

Unified buffer

The unified buffer holds the two input matrices and the computation result. It's implemented as a char array allocated on the graphics card. When loading matrix, the data will be stored in the array, and the dimensions are also stored. When doing matrix multiplication or convolution, before data are loaded into the systolic array, a result matrix will be opened in the array, dimensions are computed by both the matrices dimensions and the operation type.

While the systolic array is doing the computation, the data that need to be fed into the upper side and the left side will be offered by the buffer. In the block diagram of TPU, there are two dedicated components for offering data, while in the simulator, data are directly passed from the array. When a cell asks for input data, based on the position as well as the current cycle number, corresponding matrix data or 0 will be given, so that there will not be unnecessary sparse matrix stored in the buffer taking up spaces.

Accumulators

The accumulator unit in the simulator is rather simple, as its function is to simply add a 16-bit integer to a 32-bit one. An array of 256×256 32-bit integer is allocated on the graphics card, and when a cell need to accumulate the partial result, the result and the position of the cell will be given. The partial result will be stored until the computation is finished.

The TPU diagram gives a dedicated component for result activation and support user defined activation function. In the simulator, the ReLU activation function is supported and is handled by the accumulator unit.

When the result is to be read from the accumulator unit, the unified buffer will build result_rowsize× result_colsize blocks and each block will read a 32-bit integer to the position in the allocated result matrix.

Control unit

In the floor plan of a TPU die, the control unit takes up only 2% space. However, the control logic is mostly handled by this unit.

The matrix multiply function first reset the cycle count and flushes the systolic array and the accumulator unit. After that, the systolic array will run for certain cycles. The cycle number should be computed by the equation below. Note that data_rowsize is the same as weight_colsize.

$$run\_cycle = data\_colsize + data\_rowsize + weight\_rowsize - 1$$

Matrix convolution is done by transferring a convolution operation to a multiplication operation[4]. First, the data matrix will be transformed to a single-column vector, as the weight matrix for the multiplication. Then, a sparse matrix is made by convoluting the kernel matrix on the data matrix. A temp matrix is made by putting the kernel matrix on the data matrix, with the other positions filled by 0. The temp matrix is transformed to a single row. The process is done from the upper left corner, and the kernel matrix is moved to the left by each row and start from the next row if moved to the rightest position already. An example with a 3×3 data matrix and a 2×2 kernel is as below. Afterwards, the operation is treated as a normal matrix multiplication, with the result size configured.

$$conv\left(\begin{pmatrix} x1 & x2 & x3 \\ x4 & x5 & x6 \\ x7 & x8 & x9 \end{pmatrix}, \begin{pmatrix} k1 & k2 \\ k3 & k4 \end{pmatrix}\right) = \begin{pmatrix} k1 & k2 & 0 & k3 & k4 & 0 & 0 & 0 & 0 \\ 0 & k1 & k2 & 0 & k3 & k4 & 0 & 0 & 0 \\ 0 & 0 & 0 & k1 & k2 & 0 & k3 & k4 & 0 \\ 0 & 0 & 0 & 0 & k1 & k2 & 0 & k3 & k4 \end{pmatrix} * \begin{pmatrix} x1 \\ x2 \\ x3 \\ x4 \\ x5 \\ x6 \\ x7 \\ x8 \\ x9 \end{pmatrix}$$

Other functions such as to read input matrix and to write the result back to the host memory are simply to copy data between the host memory and the unified buffer on graphics card.

Simulation unit

The main function of this unit is to keep track of the cycles that the simulated TPU runs. In each of the systolic array cycle, the simulation unit will update a cycle counter. Also, when reading and writing matrix between the host and the device, the size of bytes being transferred will be sent to the simulation unit, and a data transfer counter will be updated.

As the official configuration of TPU, the frequency of the matrix multiply unit is 700MHz and the bandwidth between the unified buffer and the host is 10GB/s.

Due to the limitation of software simulation, the cycles consumed on data transformation, as well as those on accessing accumulators from the multiply unit are not counted.

Test Method and Result

In the previous TPU performance analysis paper, the performance was measured by cycles, and each limiting factor are given as percentages of cycles. However, different from the hardware components being designed to work as a pipeline, it's hard to measure the cycles for waiting other components in the software-based simulator. Thus, the performance is measured by the estimating time spend on each part of the computation process.

Theoretically, with a n×m data matrix and a m×k weight matrix, the array active cycles is $cycle_{active} = n + m + k - 1$. In each cycle, 256 bytes are needed to be read from the unified buffer, and 256 bytes are needed from DDR3 unit for weight reading. In each computation, (n×m + m×k + n×k×4) bytes will be transferred between host and device.

Thus, the time spent on each part is as follow.

$$T_{array} = (n + m + k - 1)/Frequency_{array}$$

$$T_{weight\_load} = (m \times k + (n + m + k - 1) \times Size_{array})/Bandwidth_{DDR3}$$

$$T_{data} = (n \times m + (n + m + k - 1) \times Size_{array} + n \times k \times 4)/Bandwidth_{Buffer}$$

$$T_{PCIe} = (n \times m + m \times k + n \times k)/Bandwidth_{PCIe}$$

The simulator is tested by first giving two matrix dimensions, then random 8-bit integer numbers are loaded in the two matrices. After that, 100 complete loading, multiplying and result writing processes are done, and the simulation results are printed. For time-saving reasons, frequency and all of the bandwidths are measured by MHz and MB/s. It is easy to know that dropping a 1024×1024 common factor from each denominator will not affect the result time ratio.

The dimensions selected for testing is 1×20, 20×10 data and weight matrices, and 1×21, 21×8, to make 200 byte and 168 byte weights.

	Total estimated time	Matrix multiply time	Weight load time	Buffer access time	PCIe transfer time
(1×20)*(20*10)	36.709	4.286	25.651	4.526	2.246
(1×21)*(21*8)	35.152	4.143	24.713	4.372	1.924

Table 1. The estimated running time in the test scenario.

	Matrix multiply time ratio	Weight load time ratio	Buffer access time ratio	PCIe transfer time ratio
(1×20)*(20*10)	11.68%	69.88%	12.33%	6.12%
(1×21)*(21*8)	11.79%	70.30%	12.43%	5.47%

Table 2. The estimated running time ratio in the test scenario.

Compared to the running data from **Table 3** in the TPU performance analysis paper, noting that MLP0 corresponds to the 200-byte weight and MLP1 corresponds to the 168-byte weight matrix multiplication, the result is rather close. Note that in **Table 3**, weight cycles are divided to weight stall cycles and weight shift cycles, yet in **Table 2**, they add up as weight load time.

Application	MLP0	MLP1
Array active cycles	12.7%	10.6%
Weight stall cycles	53.9%	44.2%
Weight shift cycles	15.9%	13.4%
Non-matrix cycles	17.5%	31.9%

Table 3. Factors limiting TPU performance of the NN workload based on hardware performance counters.[2]

Some differences are potentially due to the following reasons.

1. Non-pipelined software simulation approach may not accurately reflect the stall cycles.
2. Cycles/Time spent on the processing in the control unit, accumulator unit as well as the unified buffer are not counted.
3. The highly simplified simulator design does not fully imitate the behavior of all components
4. Other errors in running tests on real hardware.

The performance limitation of matrix convolution was not tested due to the implementation of the convolution operation is done by transferring to multiplication, thus, the sizes of the input data and kernel are limited. During testing, a correct result with small input matrix sizes can be generated, but the computational cost as well as the size limitation are massive. Considering the fact that TPU has been widely used for accelerating different types of neural networks, including CNN, it would be impractical to assume TPU implemented convolution using the same approach.

In addition, the cost for rearranging the input matrices when conducting convolution is not only high, but also difficult to measure in the simulation environment, as it's related to the processing capability of the control unit/unified buffer controller.

Discussion

During the implementation of the simulator, some other design choices as well as expendabilities were tested.

In the design of TPU, each cell can communicate directly to the accumulator unit, instead of only having connection along the sides. This design can potentially increase the complexity of the circuit design. During the implementation, some other designs were tied, attempting to not have an accumulator unit connected to all of the cells. All of the designs store the results in the cells and need to shift the results out after computation.

The first one was a more traditional design, with each cell having a 32-bit result storage within, sees **Figure 5**. While computation, the behavior of each cell is identical to the implemented one, except storing the partial sum in the cell. When the results need to be read our, the cells turn to a "result-shift mode" and constantly move the result downward. The cells on the upper side will read in 0, and an array of result collector in the bottom will read in the results from the lowest side cells. In this way, the results in the systolic array are read out into the unified buffer and will be handed to the host.

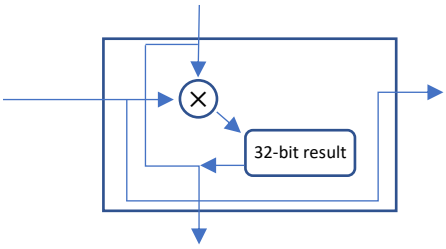

Figure 5. The simplified design diagram of a cell. Note that in "result-shift" mode, the 32-bit result will be passed to the lower cell, and a row of results will be read from the bottom of the systolic array.

Comparing to TPU's cell design, this approach eliminated the need of a random-access accumulator reachable from all the cells. However, the performance may be affected since there will be additional cycles needed for shifting results out. The number of cycles required to get the results out is

$$cycle_{result} = Size_{array}$$

Another approach attempted is to read the result from the top of the array. The structure of a single cell sees **Figure 6**. Another vertical data path is added, and when put in result-shift mode, the result will be passed vertically up, and thus read from the top edge.

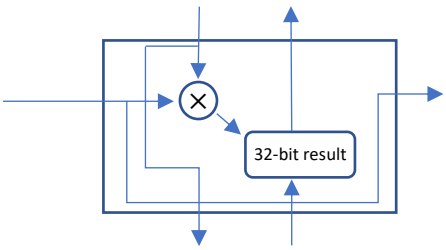

Figure 6. The simplified design diagram of a cell, in which when in "result-shift mode", the result will be passed upward and be read out from the top of the array.

In this way, the number of cycles needed for getting the result out will be reduced to the column size of the result matrix. The result collector will also need to be put in the same side as the weight fetcher. Although this approach may reduce the cycles needed for reading result, it may add the complexity to not only the array, but also the layout for the overall component layout. The result will highly depend on the implementation.

References

1. Data source: Google Trends (*https://www.google.com/trends*)

2. Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., ... & Boyle, R. (2017, June). In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (pp. 1-12). IEEE.

3. Kung, H. T. (1982). Why systolic architectures?. *IEEE computer*, *15*(1), 37-46.

4. Burrus, C. S., & Parks, T. W. (1985). *Convolution Algorithms*. John Wiley and Sons, New York,