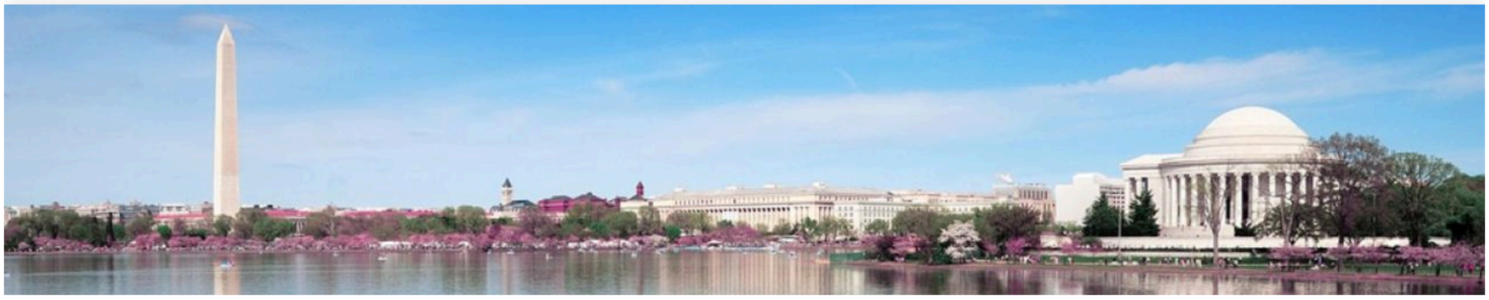


Understanding Matrix Multiplication on a Weight-Stationary Systolic Architecture

July 30, 2018 ankur6ue Computer Architecture, Machine Learning 10

If you follow the hardware for deep learning space, you may have heard of the term "systolic array". A 2D systolic array forms the heart of the Matrix Multiplier Unit (MXU) on the Google TPU and the new deep learning FPGAs from Xilinx. If you are a computer architecture expert, then you know what systolic arrays are and perhaps even implemented a convolution or matrix multiplication on a systolic array in grad school. If you are like me – generally well versed in tech, but not a computer architecture expert, then you are probably a bit lost. In this post, I'll explain what systolic arrays are and how a 1D correlation operation can be implemented using a systolic array. I'll then show an animation of how the multiplication of two 3×3 matrices is implemented on a systolic array, which will help you understand the trade-offs made in systolic architectures. The operation of the MXU on a TPU is

6.1. SEARCH

SEARCH ...

6.1. ABOUT THIS SITE

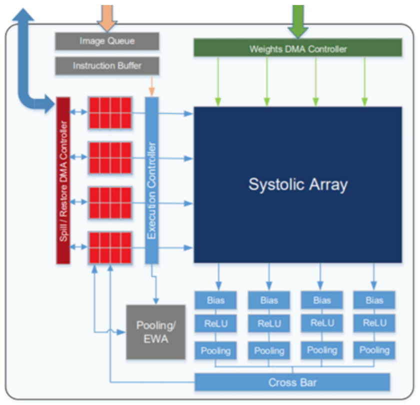
Welcome to my blog! My name is Ankur and I love to write about ML/AI algorithms and Cloud Computing technologies, as well as travel stories (see "Stories" section). I hope you find my blogs useful and enjoy reading them. Please leave a comment if you do!

6.1. CATEGORIES

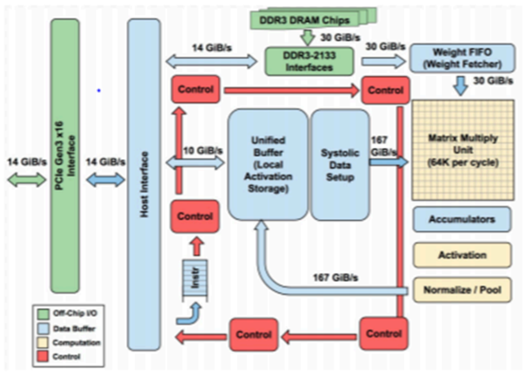
Airspace Management

AWS

identical to the data flow shown in the animation. To provide a concrete example of the ideas discussed here, I'll show relevant excerpts from the Google TPU Whitepaper.



Systolic Array on the Xilinx Deep Learning FPGA Accelerator



Systolic Array on the Google TPU

The term "systolic array" can sound like a new innovation in computer architecture. It isn't. Like most ideas in deep learning, the concept of systolic array is decades old. See this review paper by H.T. Kung for a great description. The main idea is that general purpose processors such as CPUs are not optimal for special purpose, high performance applications. By identifying the key mathematical operation involved in the application, and decomposing the mathematical operation into a sequence of simple calculations such as multiply and accumulate operations, specialized hardware can be designed that performs a large number of these simple calculations in parallel. In this post, we'll examine how this idea is applied to accelerate deep learning applications. We'll first identify what this key mathematical operation is for deep learning systems and then see how it can be efficiently implemented using systolic arrays.

Deep Learning involves a number of calculations – convolutions, matrix-matrix (M-M), matrix-vector (M-V), application of non-linearities, calculation of loss functions, weight updates, max-pooling and so on. As shown by Bill Dally in his 2015 NIPS tutorial, M-M, M-V and convolutions are by far the most expensive operations and thus the prime target for optimization. Fortunately, these operations consist of a number of repeated multiply-accumulate operations. As shown here, convolutions can

Cloud Tech
Commentary
Computer Architecture
Computer Vision
devops
Distributed Computing
ETL
Explainable AI
kubernetes
Machine Learning
Neural Machine Translation
NLP
object detection
Photography
Programming
Ray
Sensor Fusion
Sensors
seq-to-seq
Stories
Uncategorized

6.1. ABOUT THIS SITE

This may be a good place to introduce

be converted into matrix-multiplication so it suffices to focus on matrix multiplication. We'll come back to this point a bit later.

Product Z of two $n \times k$ and $k \times m$ matrices X and Y consists of nmk multiplications and $nm(k-1)$ additions

$$Z_{ij} = \sum_{i=1}^{k} X_{ik} Y_{kj} : k \text{ multiplications and } k-1 \text{ additions}$$

Applications can be compute bound or memory bound. Memory bound applications can't benefit from HW acceleration as they are bound by how quickly data can be accessed, not by how fast data can be processed. Thus, an application must be compute bound to benefit from special purpose HW such as systolic architectures. A related issue is Arithmetic Intensity (AI). As mentioned in this post about Roofline charts, Arithmetic Intensity is an algorithm specific metric that measures the number of operations per unit data. It is equivalent to "data reuse" i.e., the number of operations that can be performed before new data needs to be fetched. Algorithms with high AI are preferable as they can achieve higher Ops (Operations per second) with a lower memory bandwidth (BW) and are therefore more likely to be compute bound. Fortunately, the AI of matrix multiplication (if implemented properly) is high – (2M-1)/3 for the product of two $M \times M$ matrices and proportional to the shorter matrix dimension for product of non-square matrices. This assumes that all matrix data is available in high speed memory which can be accessed at zero latency. However, if the load/stores involved in storing and retrieving the intermediate results of the sub-computations in a matrix multiplication are not handled properly, then those can turn a compute bound calculation into a memory bound one. Memory access also consumes energy and repeated reads/writes from local storage can consume a lot of power, as noted in the Google TPU paper.

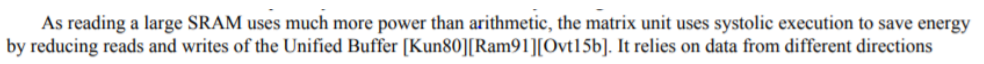

As reading a large SRAM uses much more power than arithmetic, the matrix unit uses systolic execution to save energy by reducing reads and writes of the Unified Buffer [Kun80][Ram91][Ovt15b]. It relies on data from different directions

Table of Contents

yourself and your site or include some credits.

6.1. SEARCH

SEARCH …

6.1. FIND US

Address
123 Main Street
New York, NY 10001

Hours
Monday—Friday:
9:00AM–5:00PM
Saturday & Sunday:
11:00AM–3:00PM

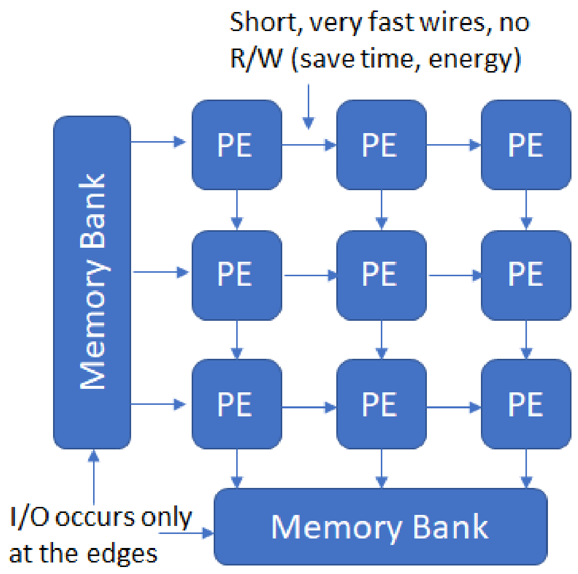
6.1. FIND US

Address
123 Main Street
New York, NY 10001

Hours
Monday—Friday:
9:00AM–5:00PM
Saturday & Sunday:
11:00AM–3:00PM

1. Systolic Architectures

A systolic system consists of a set of interconnected cells also called "processing elements" (PEs), each capable of performing some simple operation. Information can flow directly between cells in a pipelined fashion. This addresses the problem of storing/loading intermediate results that we mentioned earlier. Communications with the outside world occurs only at the boundary cells.


Short, very fast wires, no R/W (save time, energy)
Memory Bank
PE
I/O occurs only at the edges
Memory Bank

Systolic systems are similar to pipelined systems, but differ in a few ways. Systolic arrays can be 2D and data flow can be at multiple speeds in different directions. Both input and results can flow in systolic arrays, whereas only results flow in pipelined systems. The rhythmic data flow in systolic arrays keeps the control logic simple but also means that individual PEs cannot stall – if there is insufficient bandwidth, the entire array needs to stall.

To make these ideas more concrete, let's consider how correlation, a close counterpart of convolution can be implemented using a systolic array of three "processing elements" (PEs). The correlation operation can be expressed as:

$$y_i = w_1 x_i + w_2 x_{i+1} + w_3 x_{i+2}$$

$$y_1 = w_1 x_1 + w_2 x_2 + w_3 x_3$$

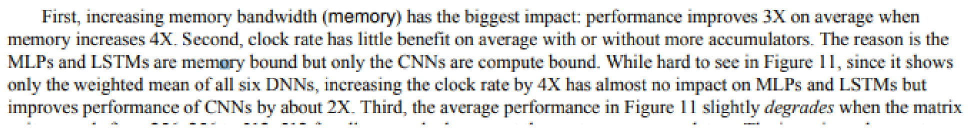
$$y_2 = w_1 x_2 + w_2 x_3 + w_3 x_4 \text{ etc}$$

So, the output can be calculated by the dot product of the weight vector with time-shifted input vector. Let's calculate the arithmetic intensity of this operation.

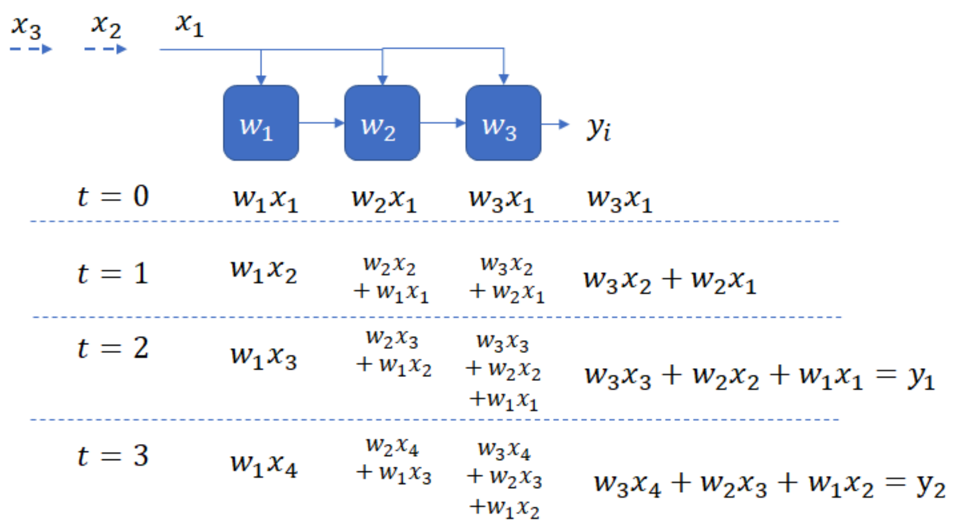
The number of operations: 3 multiplies and 2 adds

I/O: one input element is read and one output element is written back.

Thus, the arithmetic intensity is 5/2. For a k dimensional correlation, the arithmetic intensity is (2k-1)/2. This high arithmetic intensity is because of the high data reuse – each element of the output vector reuses two data points that have already been read. Thus, the correlation operation is a good candidate for a systolic array. This principle is true in general – high arithmetic intensity operations are good candidates for systolic architectures. Low AI operations are memory bound, and thus performing the calculations faster by adding parallelization or increasing the clock rate won't help. The Google TPU paper makes this point in section 7 – "Evaluation of Alternative TPU Designs"

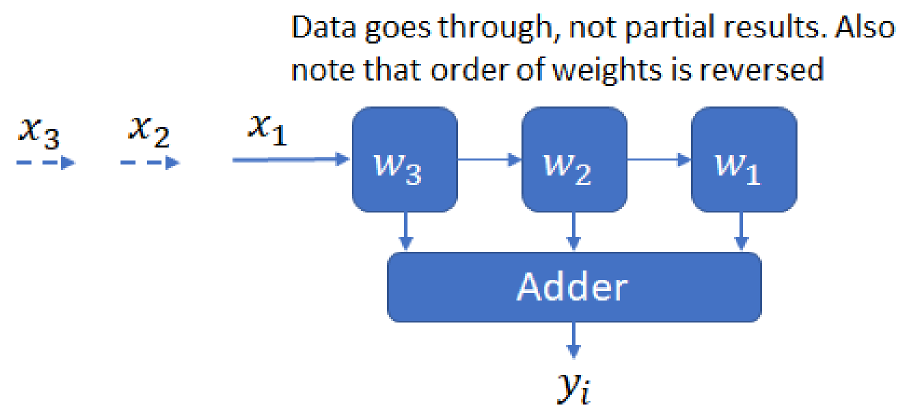

First, increasing memory bandwidth (memory) has the biggest impact: performance improves 3X on average when memory increases 4X. Second, clock rate has little benefit on average with or without more accumulators. The reason is the MLPs and LSTMs are memory bound but only the CNNs are compute bound. While hard to see in Figure 11, since it shows only the weighted mean of all six DNNs, increasing the clock rate by 4X has almost no impact on MLPs and LSTMs but improves performance of CNNs by about 2X. Third, the average performance in Figure 11 slightly *degrades* when the matrix

We can decompose the correlation operation into the individual multiply and add calculations and perform each calculation on a separate PE.


x_3
x_2
x_1
w_1
w_2
w_3
y_i
$t=0$ w_1x_1 w_2x_1 w_3x_1 w_3x_1
$t=1$ w_1x_2 $w_2x_2+w_1x_1$ $w_3x_2+w_2x_1$ $w_3x_2+w_2x_1$
$t=2$ w_1x_3 $w_2x_3+w_1x_2$ $w_3x_3+w_2x_2+w_1x_1$ $w_3x_3+w_2x_2+w_1x_1=y_1$
$t=3$ w_1x_4 $w_2x_4+w_1x_3$ $w_3x_4+w_2x_3+w_1x_2$ $w_3x_4+w_2x_3+w_1x_2=y_2$

Each element of the PE array (other than the leftmost) stores a weight value, multiplies it with its input and adds the result to the partial sum calculated by the previous element. After the second time step, one element of the output vector is computed every clock cycle, and one input element is read. We are obtaining maximum possible throughput at the minimum required bandwidth and taking advantage of all data reuse available. This design is an example of a "weight stationary" design, as the weights stay stationary and the inputs are streamed in.

In this design, each x_i is being broadcast to all PEs. Another alternative is to cycle the x_i sequentially through the PEs and and sum the partial sums using an adder.


Data goes through, not partial results. Also note that order of weights is reversed
x_3
x_2
x_1
w_3
w_2
w_1
Adder
y_i

$$t = 0 \qquad w_3x_1$$

$$t = 1 \qquad w_3x_2 + w_2x_1$$

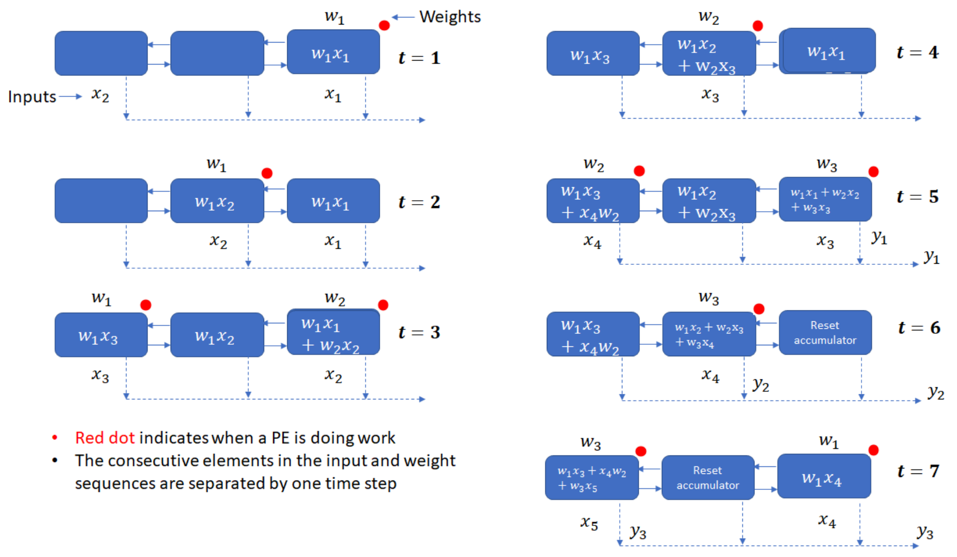
$$t = 2 \qquad w_3x_3 + w_2x_2 + w_1x_1 = y_1$$

$$t = 3 \qquad w_3x_4 + w_2x_3 + w_1x_2 = y_2$$

Each PE simply implements the multiplication operation. The add operation has been moved to the global accumulator.

A downside of both these designs is that either each element of the input must be broadcast to all PEs (in the first design) or the partial outputs must be collected and sent to the accumulator. Doing so requires the use of a bus, which must scale as the size of the correlation window increases.

An ingenious alternative is an “output stationary” architecture, where the results stay and the inputs and weights move in opposite directions. Each PE stores and accumulates the partial results. The $x_i's$ and $w_i's$ move systolically in opposite directions, such that when an x meets a w, they are multiplied and resulting product is added to the partial result at that PE. To ensure that each x_i is able to meeting every w_i, a time step is inserted between consecutive elements in the input and weight streams. The execution of this scheme is shown below.


Weights
Inputs
$t = 1$
$t = 2$
$t = 3$
$t = 4$
$t = 5$
$t = 6$
$t = 7$
Reset accumulator
Red dot indicates when a PE is doing work
The consecutive elements in the input and weight sequences are separated by one time step

This design doesn't require a bus or any other global network for collecting output from the PEs. A systolic output path, indicated by the broken arrows is sufficient.

While this design solves the problem of the global bus, it suffers from the drawback that each PE is performing useful work only one half of the time (as indicated by the red dots). Extra logic is also necessary to reset the accumulator in a PE once it completes calculation of a y_i. The paper by Kung offers a few other variations that make different design choices such as moving the weights and inputs at different speeds and adding extra storage in each PE. A summary of these choices is shown below.

Key Insight:
Trade-off complexity of PE unit with HW utilization/throughput/latency requirements

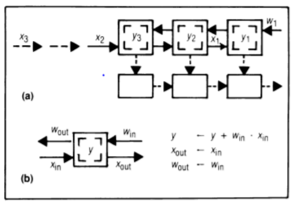


Figure 6. Design R1: systolic convolution array (a) and cell (b) where y_i's stay and x_i's and y_i's move in opposite directions systolically.

Advantage: No bus or global network is required
Disadvantage: only one-half the cells are doing useful work at one time

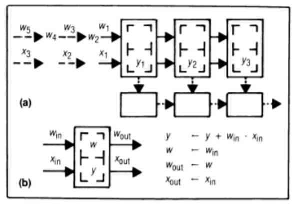


Figure 7. Design R2: systolic convolution array (a) and cell (b) where y_i's stay and x_i's and w_i's both move in the same direction but at different speeds.

Advantage: All cells work all the time when performing a single convolution
Disadvantage: Requires an extra register in each cell to hold a w value

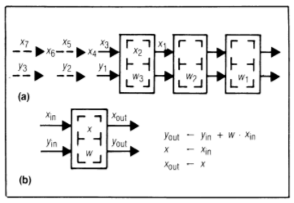


Figure 9. Design W2: systolic convolution array (a) and cell (b) where w_i's stay and x_i's and y_i's both move systolically in the same direction but at different speeds.

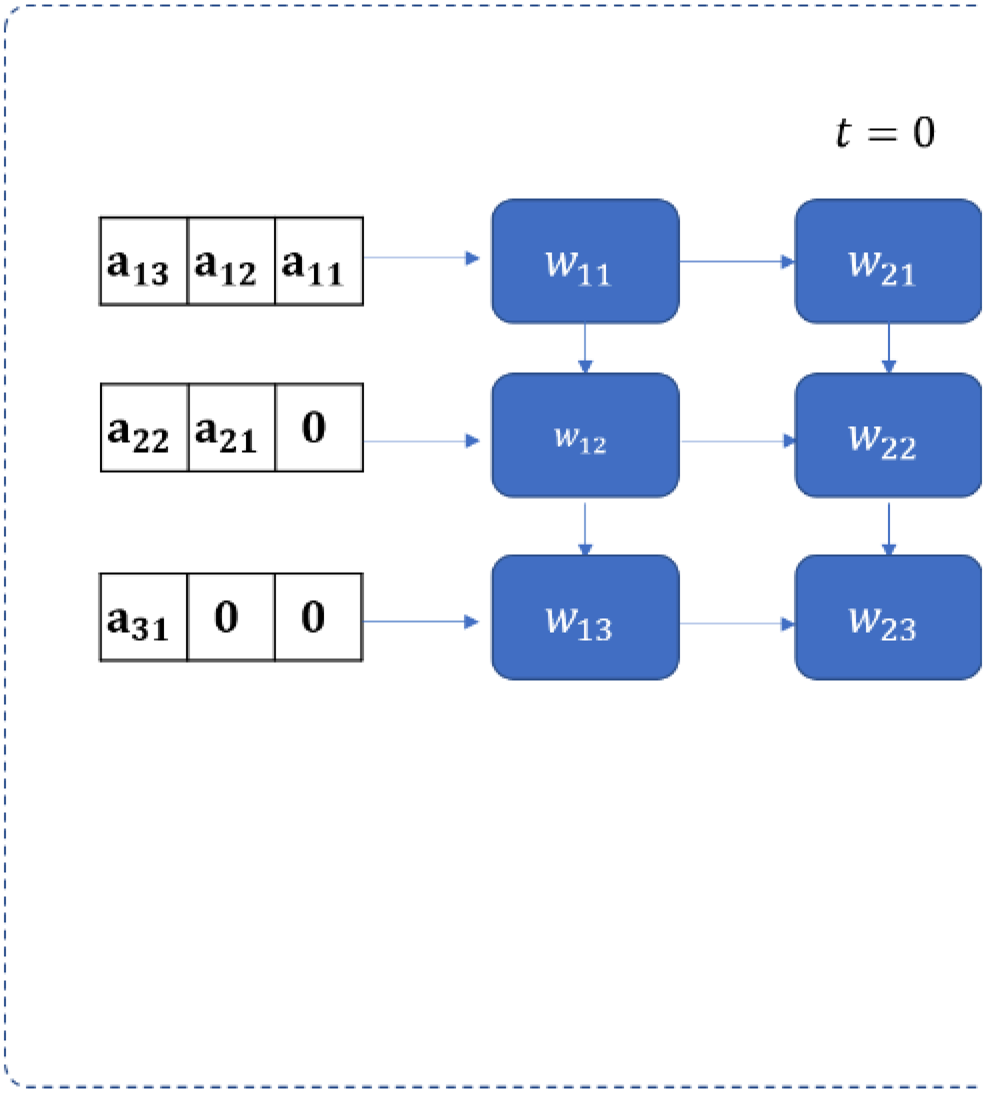
Advantage: All cells work all the time when performing a single convolution
Disadvantage: Loses constant response time, introduces latency (output y_i now available k cycles after the last of its input enters the systolic array

The key point to understand is that various trade-offs between HW utilization (how often is a PE doing useful work), throughput and latency can be obtained by adjusting the complexity of a PE and the surrounding HW (buses, timing hardware etc.)

2. Matrix Multiplication on a Weight Stationary 2D Systolic Array (MXU on a Google TPU)

The heart of the TPU is the systolic array consisting of a $N \times N$ (N=256) grid of Multiply-Accumulate (MAC) units. The TPU uses a weight-stationary architecture where the weights are pre-loaded into the MAC array and the activations are marched in from the activation storage buffer. The activations move horizontally from left to right and the partial sums move vertically from top to bottom. The results of the matrix product are fed to the activation unit which provides hardware support for common activation functions.

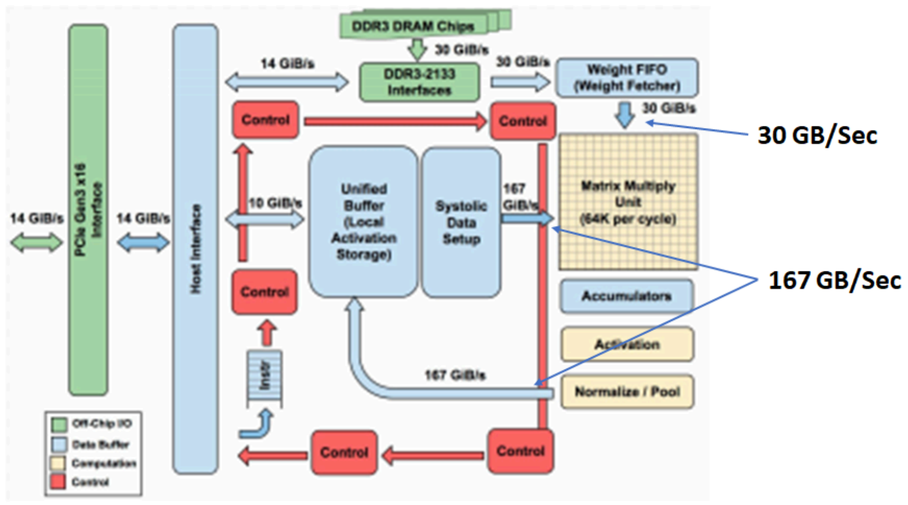
The animation below shows the flow of activations and partial sums for two 3×3 matrices. The weight matrix W is preloaded into the MAC units and the input matrices A, B, C are marched in from the left. The rows of each input matrix are offset in time. The product of the weight and input matrices is represented by $Y's$ where $Y^a = WA, Y^b = WB$ etc.



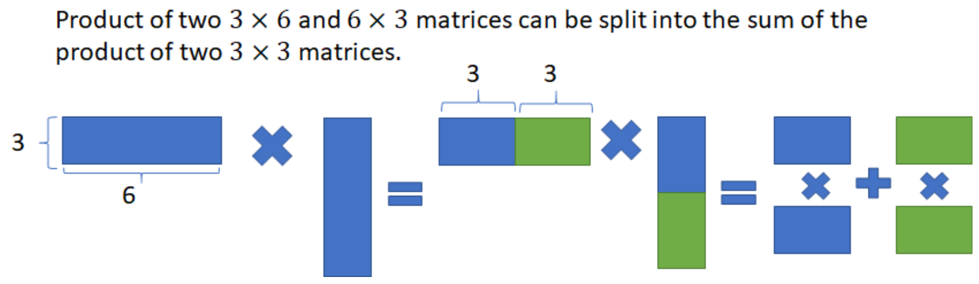
Incidentally, making this animation and getting it to work in a WordPress post was quite interesting in itself, specially since I'm not a web expert. To make the individual frames of the animation, I drew the shapes and text in PowerPoint and saved the slides as bitmaps. I then used the EaselJS 2D animation library to play the frames as an animation. It is possible to execute custom Javascript code from a WordPress post, thus I could connect the click events generated when the user clicks on the Play/Stop buttons to the corresponding onPlay, onStop handlers in the Javascript code.

Coming back to our discussion, from this animation, it should be clear that it takes 2N-1 cycles to read all elements of the product of two $N \times N$ matrices.

Note that since the weights are read much less frequently than the activations, the bandwidth into and out-of the activation storage buffer is much higher than the bandwidth of the weight buffer.



If the size of matrix-matrix multiplication is larger than the systolic array, the operation is performed in several blocks. The result of each block is stored in accumulators and is added to the result of the next block.


Product of two 3×6 and 6×3 matrices can be split into the sum of the product of two 3×3 matrices.

3. Advantages and Disadvantages of a TPU

3.1. Advantages

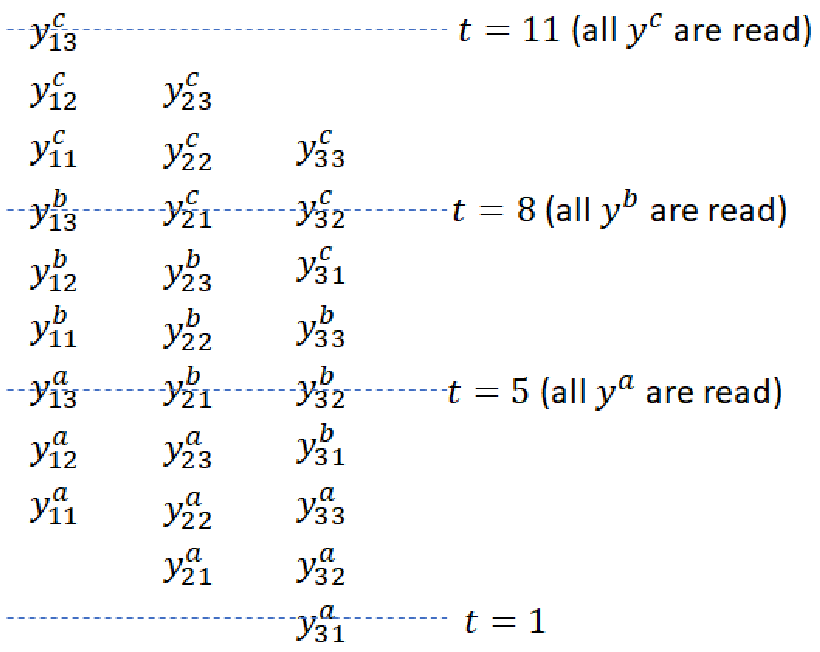
The key advantage of the TPU and systolic architectures in general is their simplicity. The TPU is a domain specific processor designed to do one thing well – matrix multiplication. It doesn't implement general purpose features such as caches, branch prediction, out-of-order execution, multiprocessing, context switching etc., which keeps the design simple and power consumption low. This simple design means that control logic takes a very small portion (2%) of the overall chip space. The matrix multiply unit and different memory types consume the bulk of the space. This helps reduce chip manufacturing costs and overall energy consumption.

A key advantage of the TPU over a GPU is good hardware utilization for latency sensitive inference applications. A GPU needs batching to take advantage of all available hardware resources. Batching however introduces latency, which is not desirable for inference applications that may have a strict latency upper bound.

3.2. Disadvantages

Latency Depends on Matrix Dimensions

We saw earlier that it takes 2N-1 clock cycles to complete a product of two $N \times N$ matrices. For a batch of size B, the latency is N(B+1) -1 (see below). This implies that the latency of the matrix product depends upon the matrix dimension. This is not desirable for constant latency applications although it isn't much of an issue for neural network inference as the size of the matrix dimensions are known in advance and the batch size can simply be adjusted to meet the latency bound.

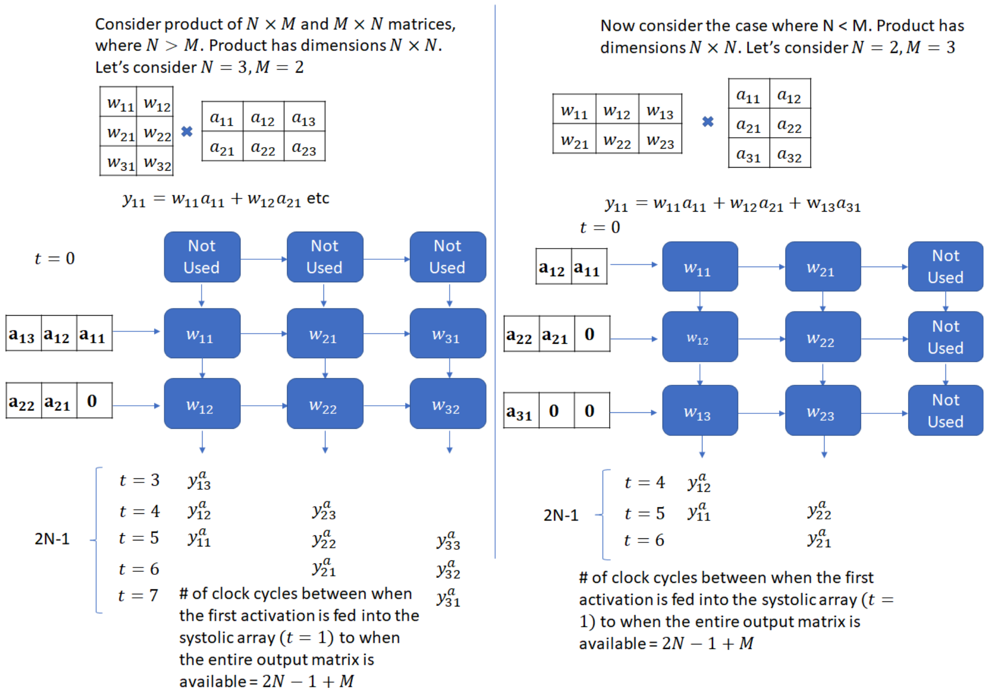


$t = 11$ (all y^c are read)
$t = 8$ (all y^b are read)
$t = 5$ (all y^a are read)
$t = 1$
y^c_{13} y^c_{12} y^c_{23} y^c_{11} y^c_{22} y^c_{33}
y^b_{13} y^c_{21} y^c_{32} y^b_{12} y^b_{23} y^c_{31} y^b_{11} y^b_{22} y^b_{33}
y^a_{13} y^b_{21} y^b_{32} y^a_{12} y^a_{23} y^b_{31} y^a_{11} y^a_{22} y^a_{33}
y^a_{21} y^a_{32} y^a_{31}

Thus for a batch of B matrices of size $N \times N$, the number of clock cycles required to read the products = $N(B-1) + 2N - 1 = N(B+1) - 1$

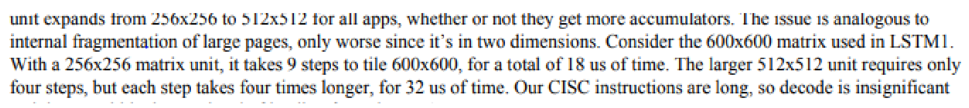
Poor MXU Utilization and Wasted Memory BW for Non-Standard Matrix Dimensions

A bigger issue arises when the matrix dimensions are not a multiple of the MXU tile size. When the weight matrix is larger than the MXU, it will need to be tiled and for the last row and column of the tile, all of available MAC units will not be used. A simple example of systolic matrix multiplication with a partially occupied tile is shown below.


Consider product of $N \times M$ and $M \times N$ matrices, where $N > M$. Product has dimensions $N \times N$. Let's consider $N = 3, M = 2$
$y_{11} = w_{11}a_{11} + w_{12}a_{21}$ etc
$t = 0$
Not Used
w_{11} w_{21} w_{31}
w_{12} w_{22} w_{32}
2N-1
$t = 3$ y^a_{13}
$t = 4$ y^a_{12} y^a_{23}
$t = 5$ y^a_{11} y^a_{22} y^a_{33}
$t = 6$ y^a_{21} y^a_{32}
$t = 7$ y^a_{31}
# of clock cycles between when the first activation is fed into the systolic array ($t = 1$) to when the entire output matrix is available = $2N - 1 + M$
Now consider the case where N < M. Product has dimensions $N \times N$. Let's consider $N = 2, M = 3$
$y_{11} = w_{11}a_{11} + w_{12}a_{21} + w_{13}a_{31}$
$t = 0$
w_{11} w_{21} Not Used
w_{12} w_{22} Not Used
w_{13} w_{23} Not Used
2N-1
$t = 4$ y^a_{12}
$t = 5$ y^a_{11} y^a_{22}
$t = 6$ y^a_{21}
# of clock cycles between when the first activation is fed into the systolic array ($t = 1$) to when the entire output matrix is available = $2N - 1 + M$

A partially occupied MXU tile also wastes memory bandwidth

The TPU paper says this:


unit expands from 256x256 to 512x512 for all apps, whether or not they get more accumulators. The issue is analogous to internal fragmentation of large pages, only worse since it's in two dimensions. Consider the 600x600 matrix used in LSTM1. With a 256x256 matrix unit, it takes 9 steps to tile 600x600, for a total of 18 us of time. The larger 512x512 unit requires only four steps, but each step takes four times longer, for 32 us of time. Our CISC instructions are long, so decode is insignificant

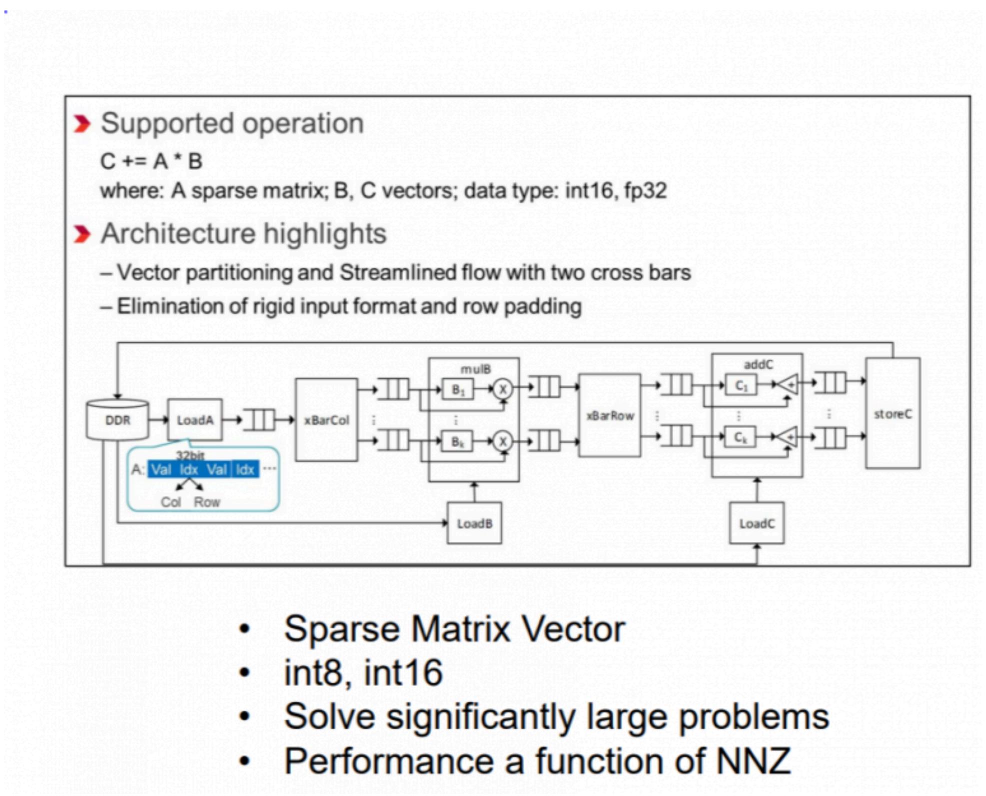
This implies that it takes the same time to load a fully occupied tile as a partially occupied one. Thus a partially occupied tile will waste memory bandwidth.

Implementing Convolutions as Matrix-Multiplications may not be Optimal

TPUs don't have direct support for convolutions. Convolutions are implemented by converting them into matrix-multiplications. This may not be optimal (particularly for large convolution kernel sizes) as convolutions have specific data flow patterns that can be leveraged to achieve better efficiency. A great example is the Eyeriss accelerator that is specially designed for the convolution operation.

No Direct Support for Sparsity

It turns out that many of the connections in a deep neural network can be pruned without significantly impacting the inference accuracy. In this approach, less important weights are progressively removed and the network retrained for a few iterations until the desired performance characteristic – for example classification accuracy or total network size is achieved. More than 90% of the weights can be pruned without a significant impact on performance. Weight pruning can dramatically reduce the size of the network, number of operations and energy consumption. Sparsity is not currently supported on the TPU. This is as much of an opportunity as a drawback, because with hardware support for sparsity, significantly better performance should be achievable. Support for sparsity will add complexity to the TPU design though as the sparse weights will no longer be located contiguously in memory. For a great example of a specialized accelerator that performs customized sparse matrix vector multiplication and handles weight sharing with no loss of efficiency, see the "Efficient Inference Engine" paper. Also, the DL FPGA from Xilinx appears to support sparse matrix-vector computations.



- Supported operation
  C += A * B
  where: A sparse matrix; B, C vectors; data type: int16, fp32
- Architecture highlights
  – Vector partitioning and Streamlined flow with two cross bars
  – Elimination of rigid input format and row padding

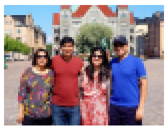
- Sparse Matrix Vector
- int8, int16
- Solve significantly large problems
- Performance a function of NNZ

4. Pipelining of Weight Reads

It takes time to read the weights from the host memory to the AI accelerator on-chip memory and to transfer the weights to the matrix-multiply unit. Therefore, it makes sense to use pipelining to hide the corresponding latencies. The basic idea is to load the weights for the next network layer while processing the computations for the current layer. The TPU implements a 4 tile weight FIFO to transfer the weights from the off-chip DRAM to the on-chip unified buffer. The matrix unit is double-buffered so it can hold the weights for the next layer while processing the current layer. Without double buffering, it will cost 256 cycles to transfer the weights for the next layer from the unified buffer to the MXU. This is similar to "page flipping" in computer graphics. The Xilinx deep learning accelerators also implement pipelined systolic arrays.

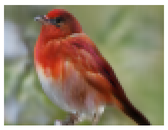
5. Conclusion

The point of this article was to help you understand the main thrust of the innovations in deep learning hardware architecture which is to design custom hardware to speed up matrix computations, particularly matrix multiplication. Systolic architectures are a well-understood technique to implement high throughput matrix multiplication by using an array of interconnected multiply-accumulate units. Understanding how matrix multiplication actually works in a 2D systolic array helps to develop intuition about the pros and cons of systolic architectures. Since systolic architectures are being used in deep learning inference accelerators from many important market participants, this intuition should be helpful in making design choices about inference HW for your specific application.


« PREVIOUS

Europe Trip (7/9/2018 – 7/23/2018)

NEXT »

Wrapping a Python Application into a Web Service using mod_wsgi and gunicorn


5.1. 10 COMMENTS

Bob McClellan

OCTOBER 4, 2018 AT 2:40 AM

Excellent read.

REPLY

ankur6ue ☆

OCTOBER 4, 2018 AT 12:46 PM

Thanks bob!

REPLY

Sazzad

OCTOBER 27, 2019 AT 2:03 AM

Very nicely written. Thank you for sharing!
I am wondering if you did any analysis on what other type of hardware compute units are proposed so far or being used other than the “Systolic array” for AI accelerators.

Thank you for your time.

REPLY

ankur6ue ☆

NOVEMBER 21, 2019 AT 4:27 AM

Glad you liked the post. As examples of other compute units – search for Eyeriss, an accelerator for convolutional NNs and EIE (Efficient Inference Engine) for an accelerator for sparse matrix calculations. The references in those papers should provide many other examples.

REPLY

emad

DECEMBER 11, 2019 AT 6:30 PM

Thank you very much for sharing the knowledge and this great post.

REPLY

Richard0618

JUNE 13, 2022 AT 9:39 AM

Thank you very much. I am working on accelerators currently and I find your post is very helpful for me to understand systolic arrays

REPLY

---

nando

OCTOBER 20, 2023 AT 9:28 PM

Hello, really nice blog! By the way, where do you know that TPUs have no direct support for convolutions? Is there a particular source?

REPLY

---

nando

OCTOBER 20, 2023 AT 9:33 PM

Hello, really nice analysis! Is there a source where it says that the TPU cannot support direct convolution? Or you just assumed it from the architecture?

REPLY

TK

MAY 8, 2024 AT 9:32 AM

This article is amazing! Finally, I understood what a systolic array is.
Thank you so much.

REPLY

TK

MAY 8, 2024 AT 9:33 AM

This is an amazing article! I finally understood what a systolic array is.
Thank you so much.

REPLY

6. Leave a Reply

Your email address will not be published.

Comment

Name *

Email *

Website

☐ Save my name, email, and website in this browser for the next time I comment.

POST COMMENT

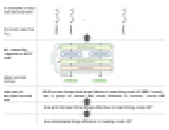
OTHER MACHINE LEARNING POSTS

Deploying Ray on a local kubernetes cluster

November 8, 2020 0

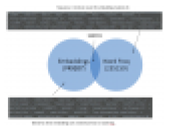
Kernel SHAP

September 17, 2020 0



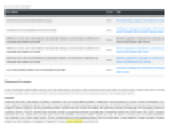
Building a Information Retrieval system based on the Covid-19 research challenge dataset: Part 3

June 16, 2020 0



Building a Information Retrieval system based on the Covid-19 research challenge dataset: Part 2

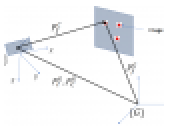
June 10, 2020 2



Building a Information Retrieval system based on the Covid-19 research challenge dataset: Part 1

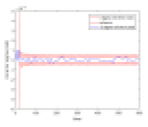
June 10, 2020 1

OTHER SENSOR FUSION RELATED POSTS



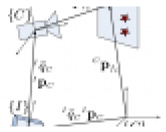
Sensor Fusion: Part 4

May 7, 2017 0



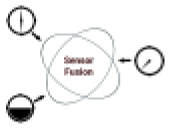
Sensor Fusion – Part 3: Implementation of Gyro-Accel Sensor Fusion

May 2, 2017 6



Sensor Fusion: Part 2 (combining Gyro-Accel data)

April 30, 2017 0



Sensor Fusion: Part 1

April 27, 2017 0