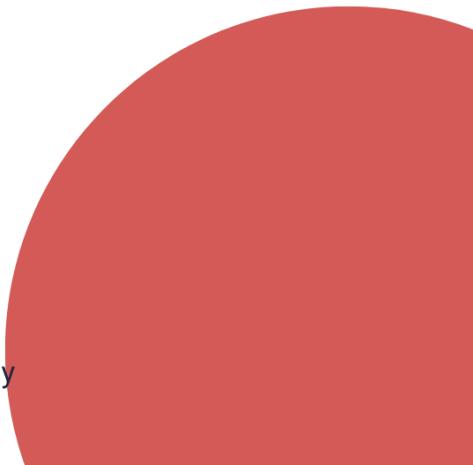




# TECHNICAL REPORT ON **TPUs**

Prepared By :  
SUMIT YADAV  
PUL076BCT088

Prepared For:  
Prof. Dr. Subarna Shakya  
Director, Information Technology  
Innovation Center, Tribhuvan University



[Sumit Yadav](#) (PUL076BCT088)

[Prof. Dr. Subarna Shakya](#)

Director, Information Technology Innovation Center, Tribhuvan University

Computer Organization and Architecture

09 June 2022

## TPUs (Tensor Processing Unit)

Tensor Processing Units (TPUs) are custom-designed application-specific integrated circuits (ASICs) developed by Google to accelerate AI and neural network works. Formally, it was begun in 2015 and first made available publicly in 2018 for both cloud computing and small business. Tensor processing unit was first introduced in 2016 at Google I/O, when the company said the TPU had been already used in the company for over a year. This chip was quite specific to machine learning framework like TensorFlow, PyTorch and JAX. TPU is another kind of processing unit like a CPU or a GPU. There are, however, some big differences between those - the biggest difference being that TPU is an ASIC. And you probably know CPU and GPU are not, as they are not optimized to do one specific kind of application.

### CPU, GPU and TPU

Now let's compare the difference between CPU, GPU and TPU. (For the case of multiple addition)

1. A CPU performs the multiply-add operation by reading each input and weight from memory, multiplying them with its ALU (Arithmetic Logic Unit), writing them back to memory and finally adding up all the multiplies values. Modern CPUs are strengthened by massive cache, branch prediction and high clock rate on each of its cores. Which all contribute to a lower latency of the CPU.
2. A GPU does however not use the fancy features which lower the latency. It also needs to orchestrate its thousands of ALUs which further decreases the latency. In short, GPUs drastically increase its throughput by parallelizing its computation in exchange for an increase in its latency.
3. A TPU on the other hand operates very differently. Its ALUs are directly connected to each other without using the memory. They can directly give pass information, which will drastically decrease latency.

Model	Die										Benchmarked Servers				
	<i>mm</i> <sup>2</sup>	<i>nm</i>	MHz	TDP	Measured		TOPS/s		GB/s	On-Chip Memory	Dies	DRAM Size	TDP	Measured	
					Idle	Busy	8b	FP						Idle	Busy
Haswell E5-2699 v3	662	22	2300	145W	41W	145W	2.6	1.3	51	51 MiB	2	256 GiB	504W	159W	455W
NVIDIA K80 (2 dies/card)	561	28	560	150W	25W	98W	--	2.8	160	8 MiB	8	256 GiB (host) + 12 GiB x 8	1838W	357W	991W
TPU	NA*	28	700	75W	28W	40W	92	--	34	28 MiB	4	256 GiB (host) + 8 GiB x 4	861W	290W	384W

Table 1. Benchmarked servers use Haswell CPUs, K80 GPUs, and TPUs. Haswell has 18 cores, and the K80 has 13 SMX processors.

Now on the basics of workload (While training AI models):-

- CPUs
  - Quick prototyping that requires maximum flexibility
  - Simple models that do not take long to train
  - Models that are dominated by custom TensorFlow operation written in C++
  - Models that are limited by available I/O or the networking bandwidth of the host machine.
- GPUs
  - Models for which source does not exist or is too onerous to change
  - Models with a significant number of custom TensorFlow operations that must run at least partially on CPUs
  - Models with TensorFlow ops that are not available on Cloud TPU
  - Medium-to-large models with larger effective batch sizes
- TPUs
  - Models dominated by matrix computation
  - Models with no custom TensorFlow operations inside the main training loop
  - Models that train for weeks or months
  - Larger and very large models with very larger effective batch sizes

### **TPU Origin, Architecture, and Implementation**

Starting as early as 2006, we discussed deploying GPUs, FPGAs, or custom ASICs in our datacenters. We concluded that the few applications that could run on special hardware could be done virtually for free using the excess capacity of our large datacenters, and it's hard to improve on free. The conversation changed in 2013 when a projection where people use voice search for 3 minutes a day using speech recognition DNNs would require our datacenters to double to meet computation demands, which would be very expensive to satisfy with conventional CPUs. Thus, we started a high-priority project to quickly produce a custom ASIC for inference (and bought off-the-shelf GPUs for training). The goal was to improve cost-performance by 10X over GPUs. Given this mandate, the TPU was designed, verified, built, and deployed in datacenters in just 15 months.

Rather than be tightly integrated with a CPU, to reduce the chances of delaying deployment, the TPU was designed to be a coprocessor on the PCIe I/O bus, allowing it to plug into existing servers just as a GPU does. Moreover, to simplify hardware design and debugging, the host server sends TPU instructions for it to execute rather than fetching them itself. Hence, the TPU is closer in spirit to an FPU (floating-point unit) coprocessor than it is to a GPU.

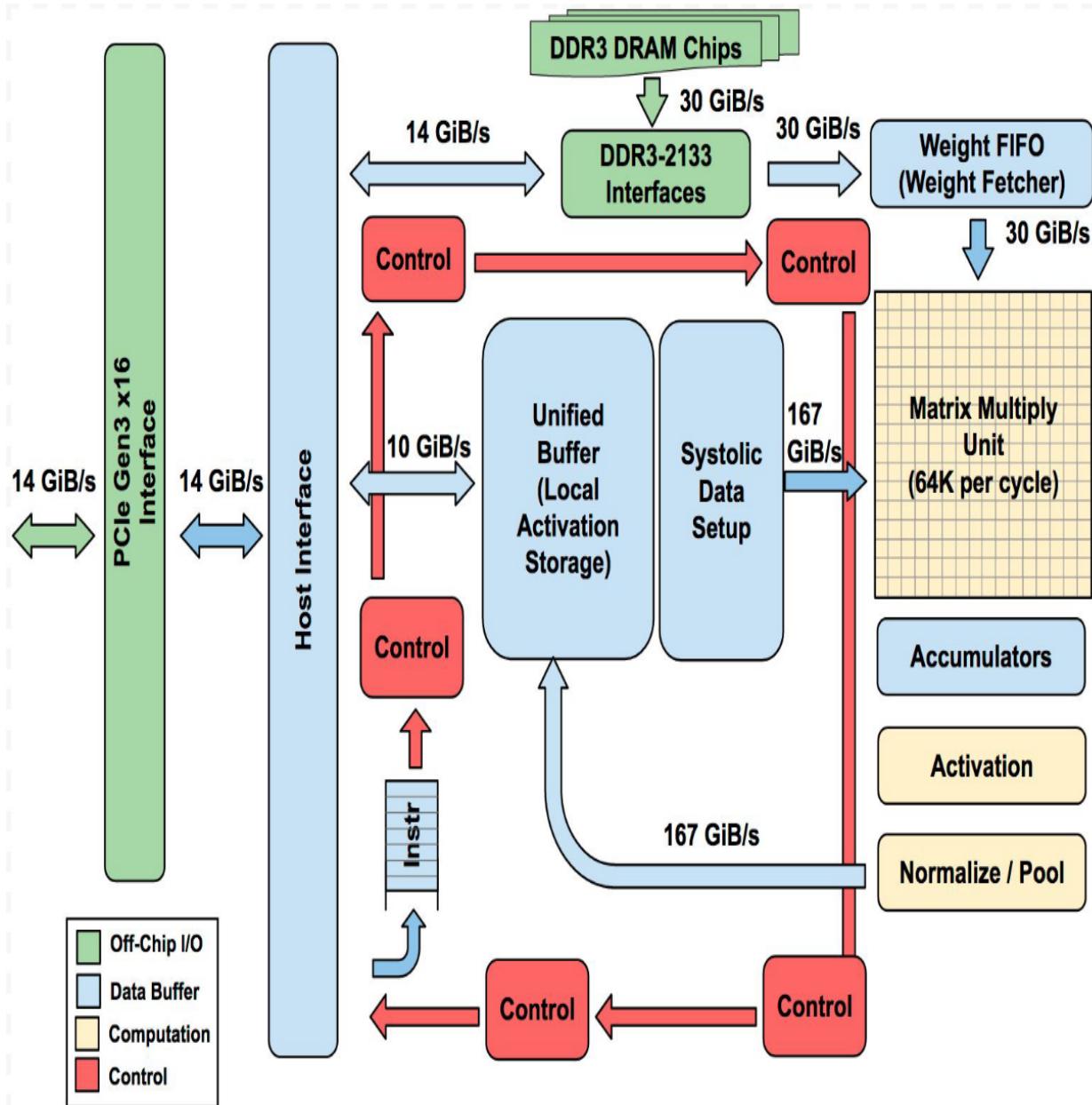


Fig 1. TPU Block Diagram. The main computation part is the yellow Matrix Multiply unit in the upper right-hand corner. Its inputs are the blue Weight FIFO and the blue Unified Buffer(UB) and its output is the blue Accumulators(Acc). The yellow Activation Unit performs the nonlinear functions on the Acc, Which go to the UB.

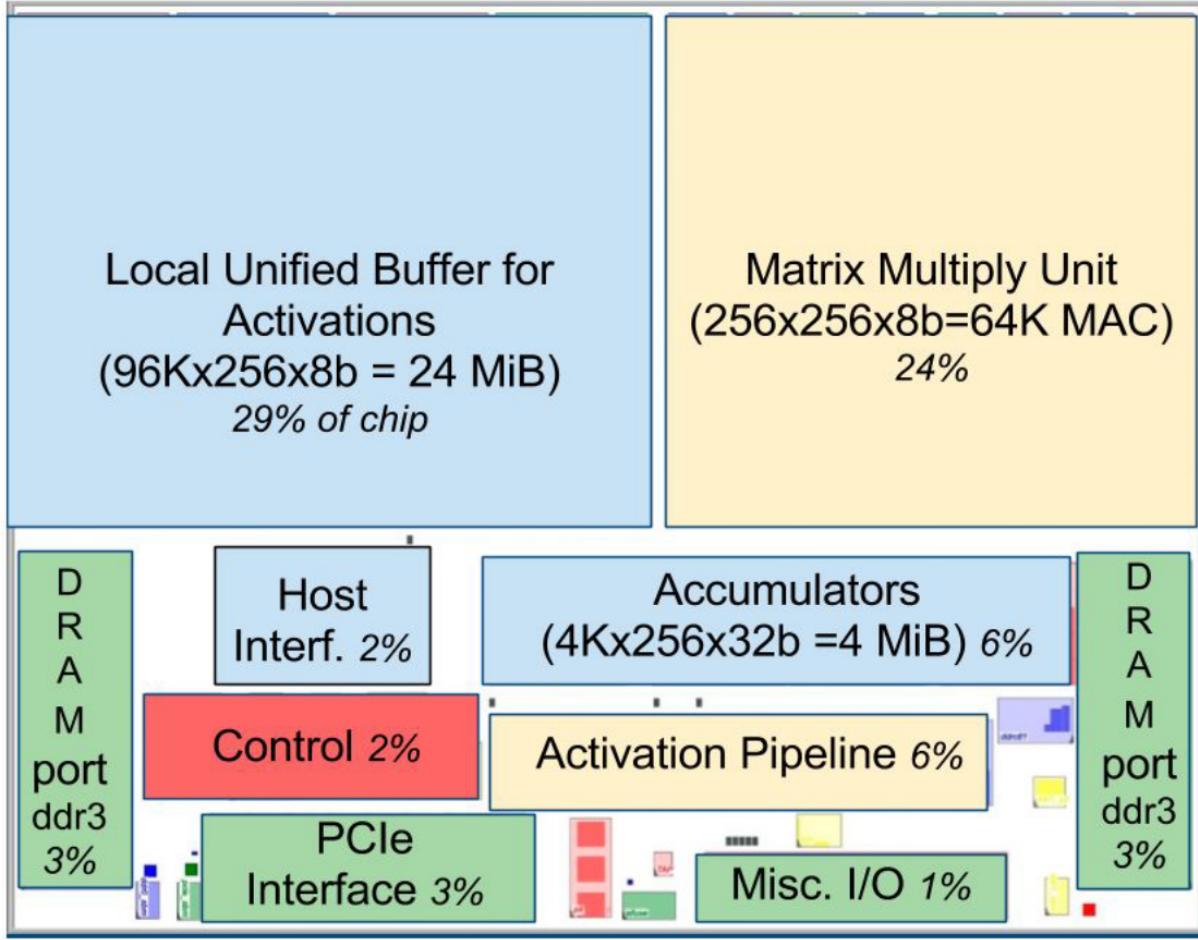


Fig 2. Floor Plan of TPU die. The shading follow Figure 1. The light(blue) data buffers are 37% of the die, the light (yellow) compute is 30%, the medium (green) I/O is 10%, and the dark (red) control is just 2%. Control is much larger (and much more difficult to design) in CPU or GPU.

The goal was to run whole inference models in the TPU to reduce interactions with the host CPU and to be flexible enough to match the NN needs of 2015 and beyond, instead of just what was required for 2013 NNs. Figure 1 shows the block diagram of the TPU. The TPU instructions are sent from the host over the PCIe Gen3 x16 bus into an instruction buffer. The internal blocks are typically connected together by 256-byte-wide paths. Starting in the upper-right corner, the **Matrix Multiply Unit** is the heart of the TPU. It contains 256x256 MACs that can perform 8-bit multiply-and-adds on signed or unsigned integers. The 16-bit products are collected in the 4 MiB of 32-bit **Accumulators** below the matrix unit. The 4MiB represents 4096, 256-element, 32-bit accumulators. The matrix unit produces one 256-element partial sum per clock cycle. We picked 4096 by first noting that the operations per byte need to reach peak performance (roofline knee in Section 4) is ~1350, so we rounded that up to 2048 and then duplicated it so that the compiler could use double buffering while running at peak performance.

When using a mix of 8-bit weights and 16-bit activations (or vice versa), the Matrix Unit computes at half-speed, and it computes at a quarter-speed when both are 16 bits. It reads

and writes 256 values per clock cycle and can perform either a matrix multiply or a convolution. The matrix unit holds one 64 KiB tile of weights plus one for double-buffering (to hide the 256 cycles it takes to shift a tile in). This unit is designed for dense matrices. Sparse architectural support was omitted for time-to-deploy reasons. Sparsity will have high priority in future designs. The weights for the matrix unit are staged through an on-chip **Weight FIFO** that reads from an off-chip 8 GiB DRAM called **Weight Memory** (for inference, weights are read-only; 8 GiB supports many simultaneously active models). The weight FIFO is four tiles deep. The intermediate results are held in the 24 MiB on-chip **Unified Buffer**, which can serve as inputs to the Matrix Unit. A programmable DMA controller transfers data to or from CPU Host memory and the Unified Buffer. Figure 2 shows the floor plan of the TPU die. The 24 MiB Unified Buffer is almost a third of the die and the Matrix Multiply Unit is a quarter, so the datapath is nearly two-thirds of the die. The 24 MiB size was picked in part to match the pitch of the Matrix Unit on the die and, given the short development schedule, in part to simplify the compiler. Control is just 2%. Figure 3 shows the TPU on its printed circuit card, which inserts into existing servers like an SATA disk.

As instructions are sent over the relatively slow PCIe bus, TPU instructions follow the CISC tradition, including a repeat field. The average clock cycles per instruction (CPI) of these CISC instructions is typically 10 to 20. It has about a dozen instructions overall, but these five are the key ones:

1. `Read_Host_Memory` reads data from the CPU host memory into the Unified Buffer (UB).
2. `Read_Weights` reads weights from Weight Memory into the Weight FIFO as input to the Matrix Unit.
3. `MatrixMultiply/Convolve` causes the Matrix Unit to perform a matrix multiply or a convolution from the Unified Buffer into the Accumulators. A matrix operation takes a variable-sized  $B \times 256$  input, multiplies it by a  $256 \times 256$  constant weight input, and produces a  $B \times 256$  output, taking  $B$  pipelined cycles to complete.
4. `Activate` performs the nonlinear function of the artificial neuron, with options for ReLU, Sigmoid, and so on. Its inputs are the Accumulators, and its output is the Unified Buffer. It can also perform the pooling operations needed for convolutions using the dedicated hardware on the die, as it is connected to nonlinear function logic.
5. `Write_Host_Memory` writes data from the Unified Buffer into the CPU host memory.

The other instructions are alternate host memory read/write, set configuration, two

versions of synchronization, interrupt host, debug-tag, nop, and halt. The CISC MatrixMultiply instruction is 12 bytes, of which 3 are Unified Buffer address; 2 are accumulator address; 4 are length (sometimes 2 dimensions for convolutions); and the rest are opcode and flags.

The philosophy of the TPU microarchitecture is to keep the matrix unit busy. It uses a 4-stage pipeline for these CISC instructions, where each instruction executes in a separate stage. The plan was to hide the execution of the other instructions by overlapping their execution with the `MatrixMultiply` instruction. Toward that end, the `Read_Weights` instruction follows the decoupled-access/execute philosophy [Smi82], in that it can complete after sending its address but before the weight is fetched from Weight Memory. The matrix unit will stall if the input activation or weight data is not ready.

We don't have clean pipeline overlap diagrams, because our CISC instructions can occupy a station for thousands of clock cycles, unlike the traditional RISC pipeline with one clock cycle per stage. Interesting cases occur when the activations for one network layer must complete before the matrix multiplications of the next layer can begin; we see a "delay slot," where the matrix unit waits for explicit synchronization before safely reading from the Unified Buffer.

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. It relies on data from different directions arriving at cells in an array at regular intervals where they are combined. Figure 4 shows that data flows in from the left, and the weights are loaded from the top. A given 256-element multiply-accumulate operation moves through the matrix as a diagonal waveform. The weights are preloaded, and take effect with the advancing wave alongside the first data of a new block. Control and data are pipelined to give the illusion that the 256 inputs are read at once, and that they instantly update one location of each of 256 accumulators. From a correctness perspective, software is unaware of the systolic nature of the matrix unit, but for performance, it does worry about the latency of the unit.

The TPU software stack had to be compatible with those developed for CPUs and GPUs so that applications could be ported quickly to the TPU. The portion of the application run on the TPU is typically written in TensorFlow and is compiled into an API that can run on GPUs or TPUs [Lar16]. Like GPUs, the TPU stack is split into a User Space Driver and a Kernel Driver. The Kernel Driver is lightweight and handles only memory management and interrupts. It is designed for long-term stability. The User Space driver changes frequently. It sets up and controls TPU execution, reformats data into TPU order,

translates API calls into TPU instructions, and turns them into an application binary. The User Space driver compiles a model the first time it is evaluated, caching the program image and writing the weight image into the TPU's weight memory; the second and following evaluations run at full speed. The TPU runs most models completely from inputs to outputs, maximizing the ratio of TPU compute time to I/O time. Computation is often done one layer at a time, with overlapped execution allowing the matrix multiply unit to hide most non-critical-path operations.

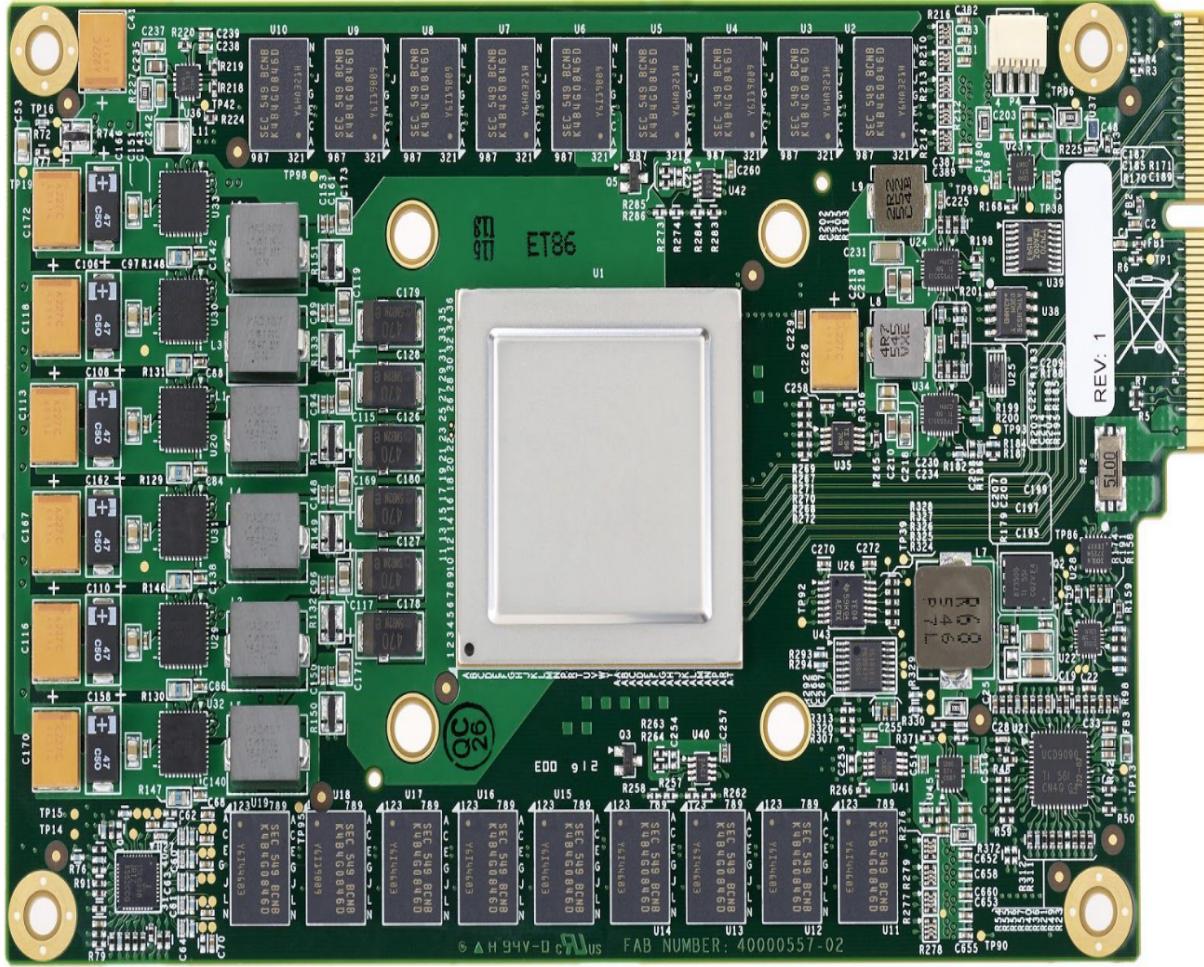


Fig 3. TPU Printed Circuit Board. It can be inserted in the slot for an SATA disk in server, but the card uses PCIe Gen3 x16.

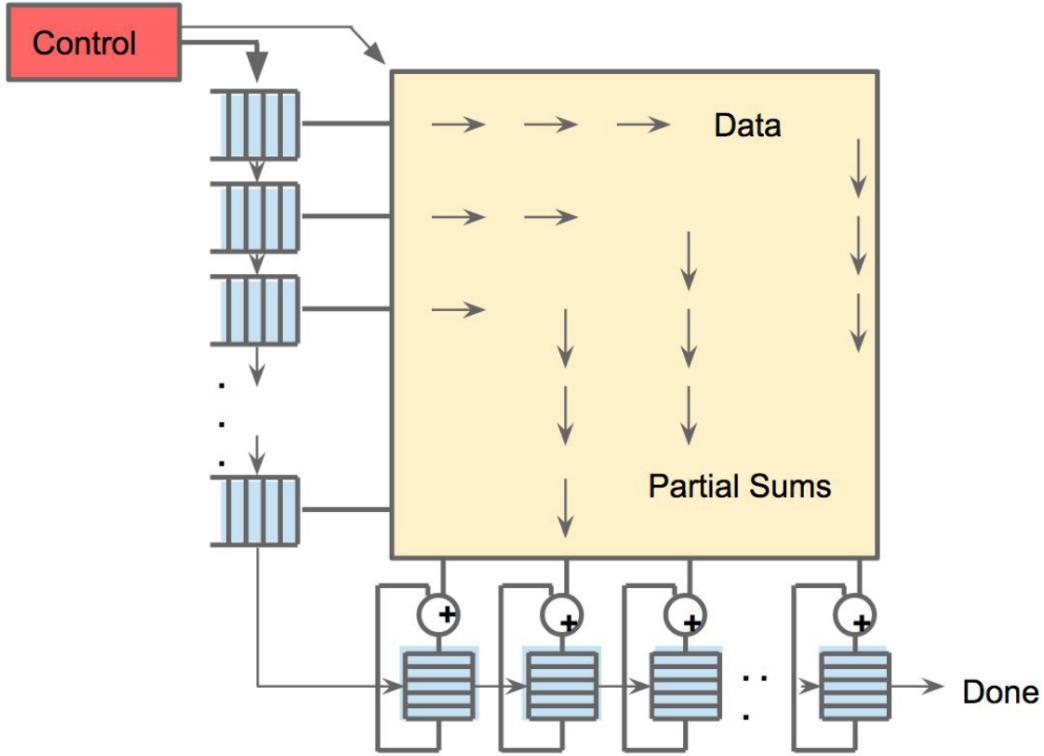


Fig 4. Systolic data flow of the Matrix Multiply Unit. Software has the illusion that each 256B input is read at once, and they instantly update one location of each of 256 accumulators RAMs.

## UNDERSTANDING MORE

To understand why TPUs were designed the way, let's look at the calculations involved in running a simple neural network.

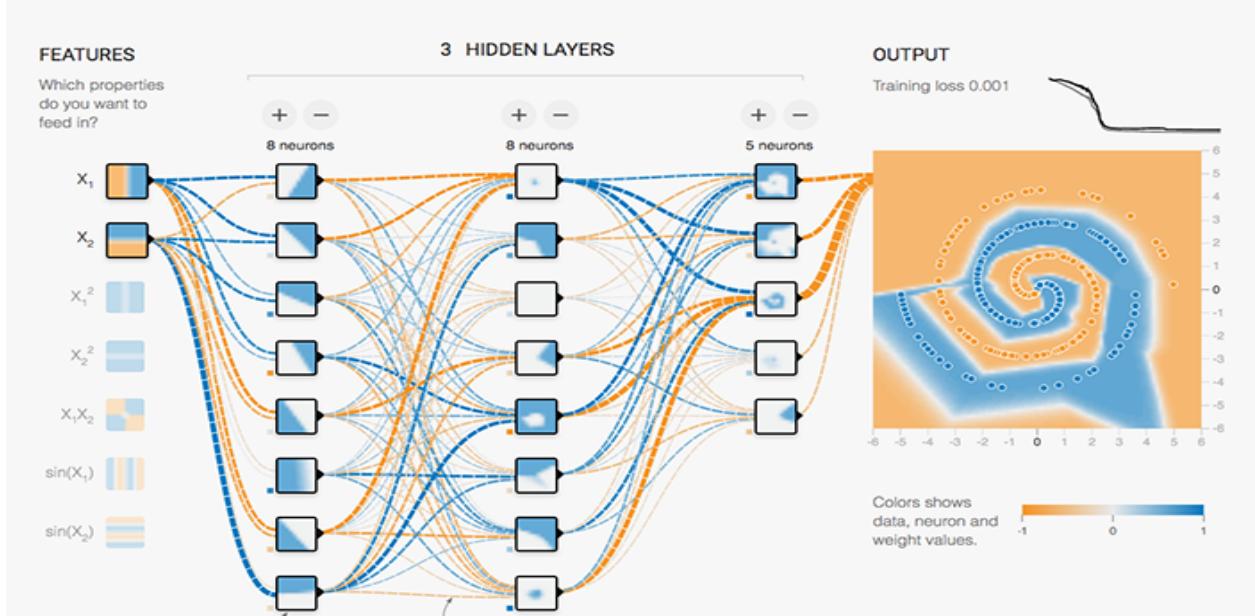
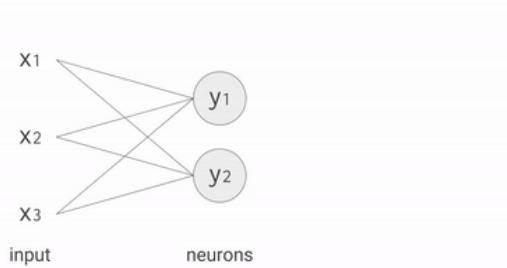


Fig5. Double spiral problem on TensorFlow Playground ([click here](#) to try it)

This example on the TensorFlow Playground trains a neural network to classify a data point as blue or orange based on a training dataset. The process of running a trained neural network to classify data with labels or estimate some missing or future values is called **inference**. For inference, each neuron in a neural network does the following calculations:

- **Multiply** the input data( $x$ ) with weights ( $w$ ) to represent the signal strength
- **Add** the results to aggregate the neuron's state into a single value
- Apply an **activation** function ( $f$ ) to modulate the artificial neuron's activity.



For example, if you have three inputs and two neurons with a fully connected single-layer neural network, you have to execute six multiplications between the weights and inputs and add up the multiplications in two groups of three. This sequence of multiplications and additions can be written as a **matrix multiplication**. The outputs of this matrix multiplication are then processed further by an activation function. Even when working with much more complex neural network model architectures, multiplying matrices is often the most computationally intensive part of running a trained model.

How many multiplication operations would you need at production scale? In July 2016, Google surveyed six representative neural network applications across Google's production services and summed up the total number of weights in each neural network architecture. You can see the results in the table below.

Type of network	# of network layers	# of weights	% of deployed
MLP0	5	20M	61%
MLP1	4	5M	
LSTM0	58	52M	29%

LSTM1	56	34M	
CNN0	16	8M	5%
CNN1	89	100M	

As you can see in the table, the number of weights in each neural network varies from 5 million to 100 million. Every single prediction requires many steps of multiplying processed input data by a weight matrix and applying an activation function.

In total, this is a massive amount of computation. As a first optimization, rather than executing all of these mathematical operations with ordinary 32-bit or 16-bit floating point operations on CPUs or GPUs, we apply a technique called quantization that allows us to work with integer operations instead. This enables us to reduce the total amount of memory and computing resources required to make useful predictions with our neural network models.

### PARALLEL PROCESSING ON THE MATRIX MULTIPLIER UNIT

Typical RISC processors provide instructions for simple calculations, such as multiplying or adding numbers. These are so-called **scalar processors**, as they process a single operation (= scalar operation) with each instruction.

Even though CPUs run at clock speeds in the gigahertz range, it can still take a long time to execute large matrix operations via a sequence of scalar operations. One effective and well-known way to improve the performance of such large matrix operations is through **vector processing**, where the same operation is performed concurrently across numerous data elements at the same time. CPUs incorporate instruction set extensions such as [SSE](#) and [AVX](#) that express such vector operations. The streaming multiprocessors (SMs) of GPUs are effectively vector processors, with many such SMs on a single GPU die. Machines with vector processing support can process hundreds to thousands of operations in a single clock cycle.

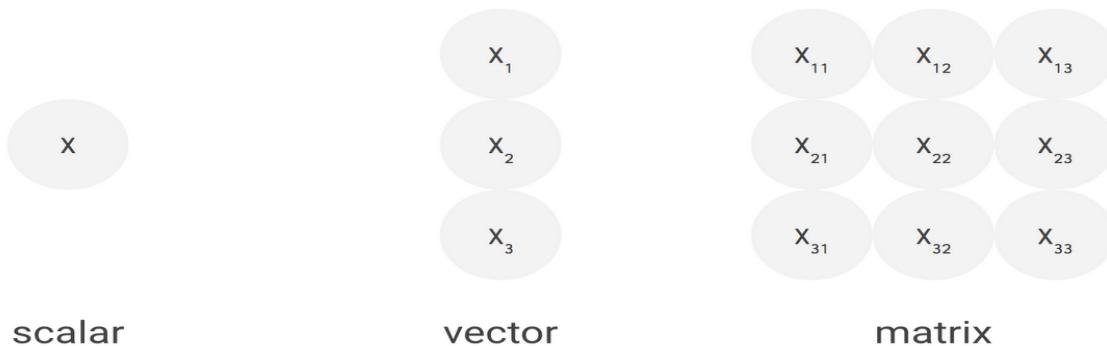


Fig6. Scalar, vector and matrix

In the case of the TPU, Google designed its MXU as a **matrix processor** that processes **hundreds of thousands of operations (= matrix operation)** in a single clock cycle. Think of it like printing documents one character at a time, one line at a time and a page at a time.

### The heart of the TPU: A systolic array

To implement such a large-scale matrix processor, the MXU features a drastically different architecture than typical CPUs and GPUs, called a [systolic array](#). CPUs are designed to run almost any calculation; they're general-purpose computers. To implement this generality, CPUs store values in registers, and a program tells the Arithmetic Logic Units (ALUs) which registers to read, the operation to perform (such as an addition, multiplication or logical AND) and the register into which to put the result. A program consists of a sequence of these read/operate/write operations. All of these features that support generality (registers, ALUs and programmed control) have costs in terms of power and chip area.

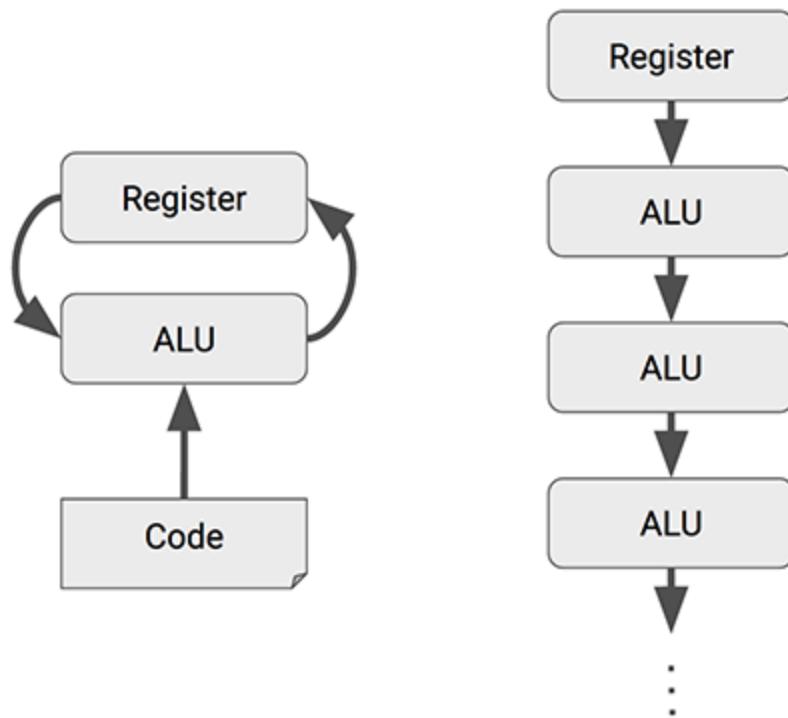


Fig 7. CPUs and GPUs often spend energy to access multiple registers per operation.  
A systolic array chains multiple ALUs together, reusing the result of reading a single register.

For an MXU, however, matrix multiplication reuses both inputs many times as part of producing the output. We can read each input value once, but use it for many different operations without storing it back to a register. Wires only connect spatially adjacent ALUs, which makes them short and energy-efficient. The ALUs perform only multiplications and additions in fixed patterns, which simplifies their design.

The design is called systolic because the data flows through the chip in waves, reminiscent of the way that the heart pumps blood. The particular kind of systolic array in the MXU is optimized for power and area efficiency in performing matrix multiplications, and is not well suited for general-purpose computation. It makes an engineering tradeoff: limiting registers, control and operational flexibility in exchange for efficiency and much higher operation density.

The TPU Matrix Multiplication Unit has a systolic array mechanism that contains  $256 \times 256 =$  total 65,536 ALUs. That means a TPU can process 65,536 multiply-and-adds for 8-bit integers every cycle. Because a TPU runs at 700MHz, a TPU can compute  $65,536 \times 700,000,000 = 46 \times 10^{12}$  multiply-and-add operations or 92 Teraops per second ( $92 \times 10^{12}$ ) in the matrix unit.

Operations per cycle	
CPU	a few
CPU (vector extension)	tens
GPU	tens of thousands
TPU	hundreds of thousands, up to 128K

In comparison, a typical RISC CPU without vector extensions can only execute just one or two arithmetic operations per instruction, and GPUs can execute thousands of operations per instruction. With the TPU, a single cycle of a MatrixMultiply instruction can invoke hundreds of thousands of operations.

During the execution of this massive matrix multiply, all intermediate results are passed directly between 64K ALUs without any memory access, significantly reducing power consumption and increasing throughput. As a result, the CISC-based matrix processor design delivers an outstanding performance-per-watt ratio: TPU provides a **83X** better ratio compared with contemporary CPUs and a **29X** better ratio than contemporary GPUs.

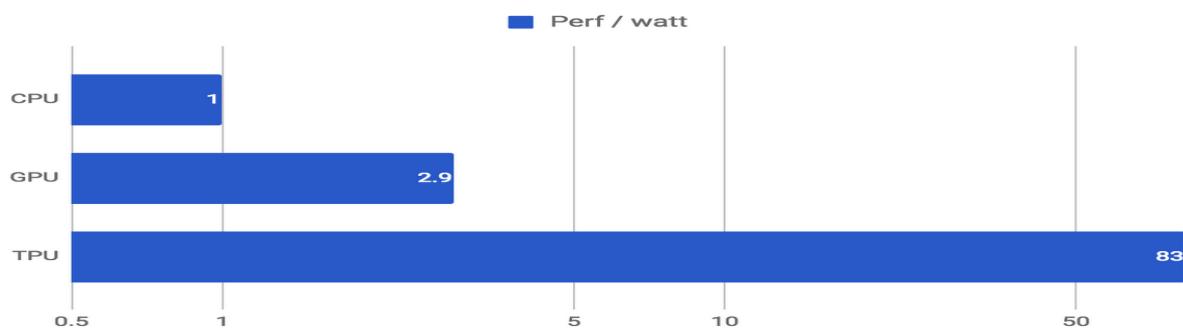
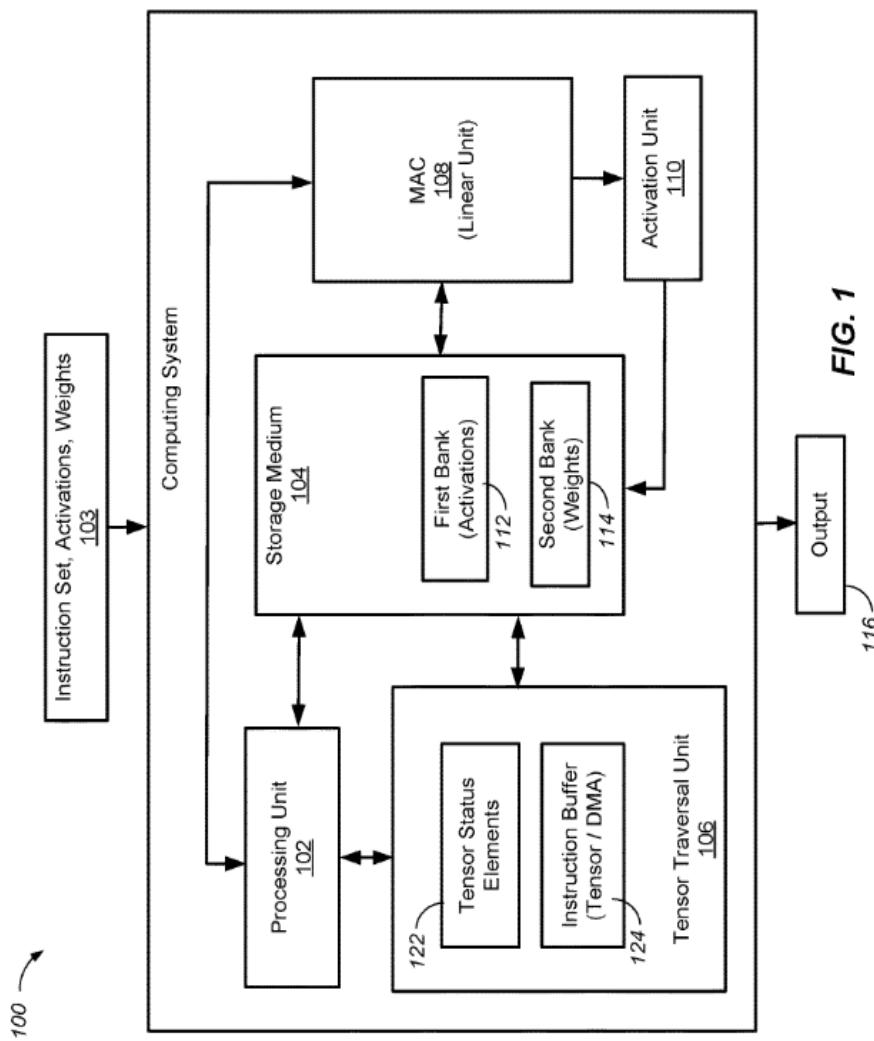


Fig 8. Performance / watt, relative to contemporary CPUs and GPUs (in log scale)(Incremental, weighted mean)

## INSTRUCTION SET

A computer-implemented method that includes receiving, by a processing unit, an instruction that specifies data values for performing a tensor computation . In response to receiving the instruction, the method may include, performing, by the processing unit, the tensor computation by executing a loop nest comprising a plurality of loops, wherein a structure of the loop nest is defined based on one or more of the data values of the instruction. The tensor computation can be at least a portion of a computation of a neural network layer. The data values specified by the instruction may comprise a value that specifies a type of the neural network layer, and the structure of the loop nest can be defined at least in part by the type of the neural network layer.



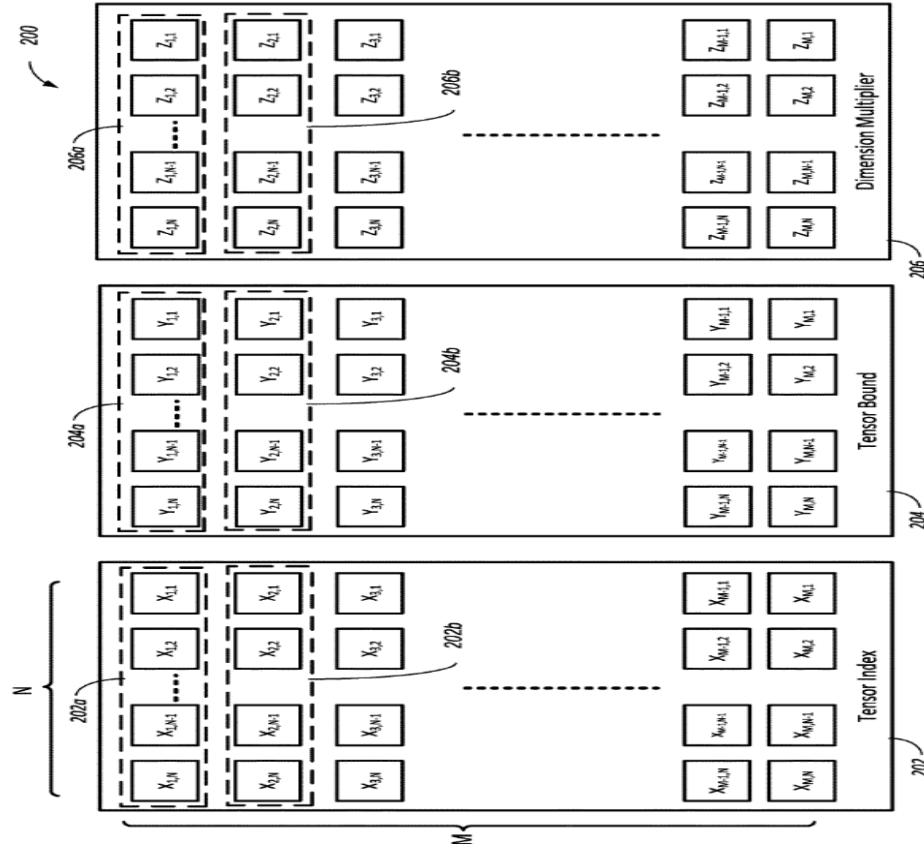
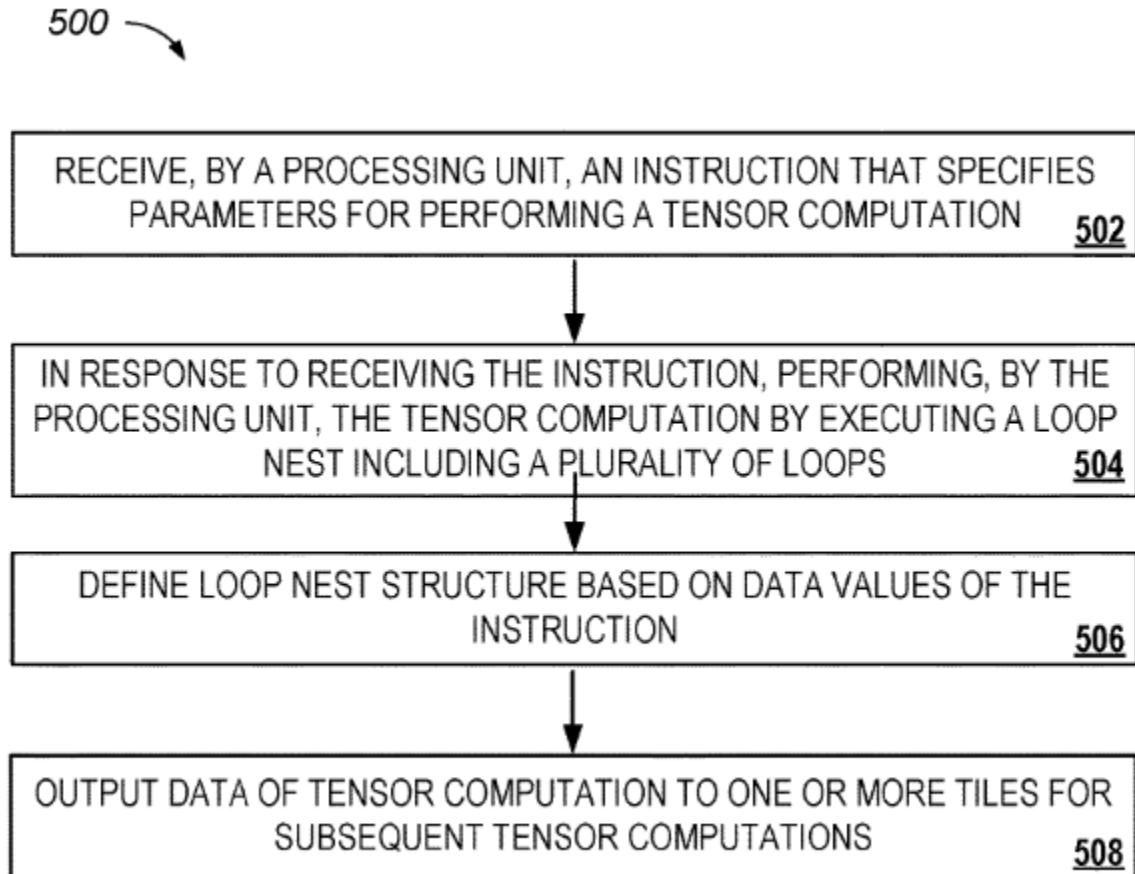


FIG. 2

Operation Code (Opcode)	Operation Type	Operation Description
0	TensorOp	Computations at the convolution and/or FC layers
1	TensorOp	Computations at the max pooling layers
2	TensorOp	Computations at the average pooling layers
3	TensorOp	Computations at the depth-wise convolution and/or element-wise multiply layers
4	DMAOp.In	Compute tile (system 100) receives input activations and other data from instruction set 103 (via an example data bus) and place data in first bank 112
5	DMAOp.Out	Compute tile reads data from first bank 112 and writes the data to an example data bus to output the data from system 100
6	DMAOp.W-N	Compute tile moves data between the first bank 112 (N-mem) and the second bank 114 (W-mem). With this operation, data is moved from W-mem to N-mem
7	DMAOp.N-W	Compute tile moves data between the first bank 112 (N-mem) and the second bank 114 (W-mem). With this operation, data is moved from N-mem to W-mem
8	DMAOp.R-bus	Compute tile writes data to second bank 114 after retrieving the data from instruction set 103 (via an example data bus); or operation to read data from second bank and write the data to the bus to output the data from the compute tile
9*	DMAOp.InFeed*	Higher level controller (external to compute tile) sends activations and weights/parameter to compute tile
10*	DMAOp.OutFeed*	Higher level controller moves inference workload results from classifier tile to external I/O device
11	TileFenceOp	Operation to prevent any new instructions from being issued until pending instructions in the respective instruction buffers are retired
12	ScalarFenceOp	Operation to prevent issuance of new instructions until a threshold tilefenceop count is reached

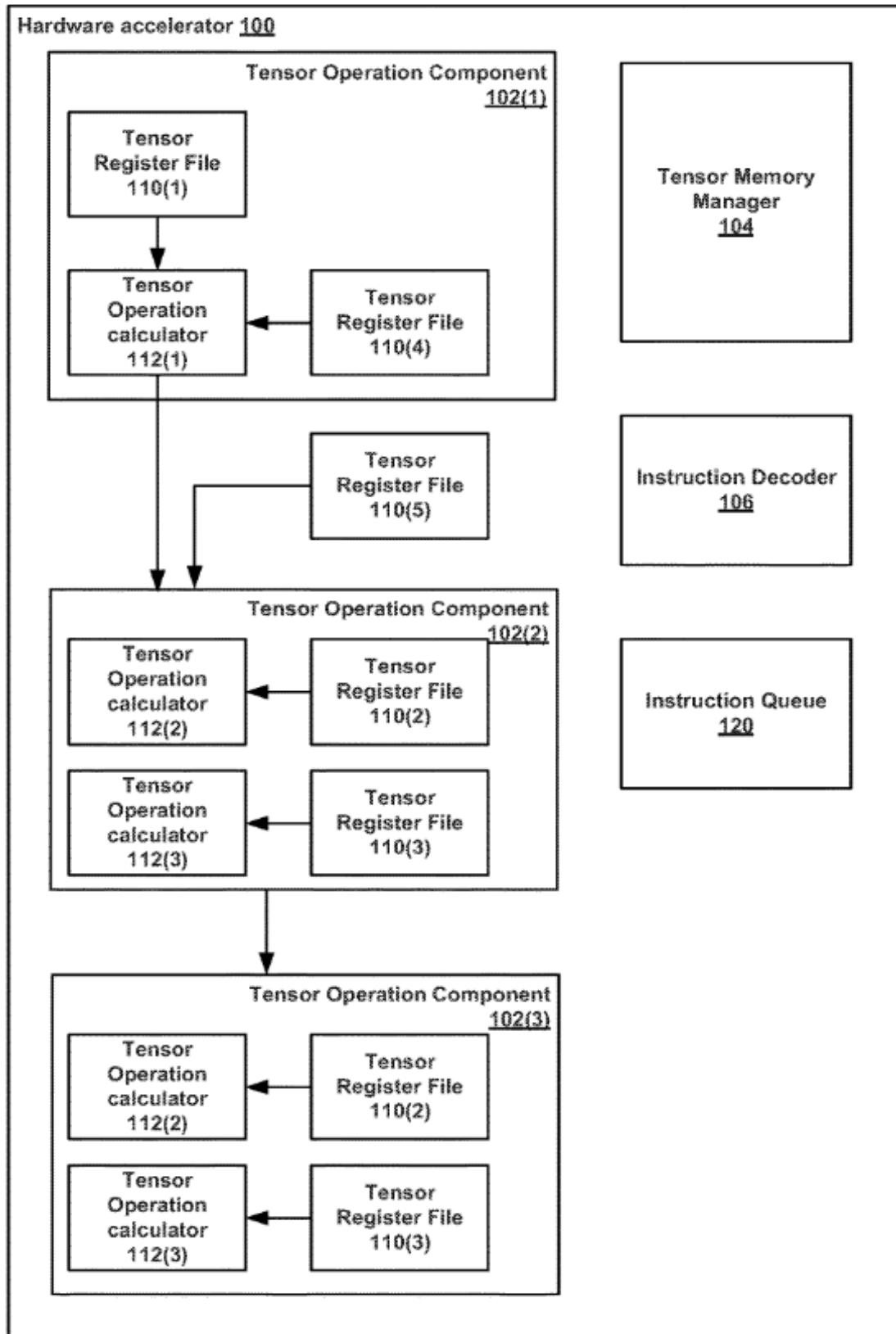
\* - Associated, in part, with operation performed by higher level controller device that is external to compute tile(s)

FIG. 3



#### FLOW DIAGRAM

A hardware accelerated having an efficient instruction set is disclosed . An apparatus may comprise logic configured to access a first and a second machine instruction . The second machine instruction may be missing a tensor operand needed to execute the second machine instruction. The logic may be further configured to execute the first machine instruction, resulting in a tensor. The logic may be further configured to execute the second machine instruction using the resultant tensor as the missing tensor operand.



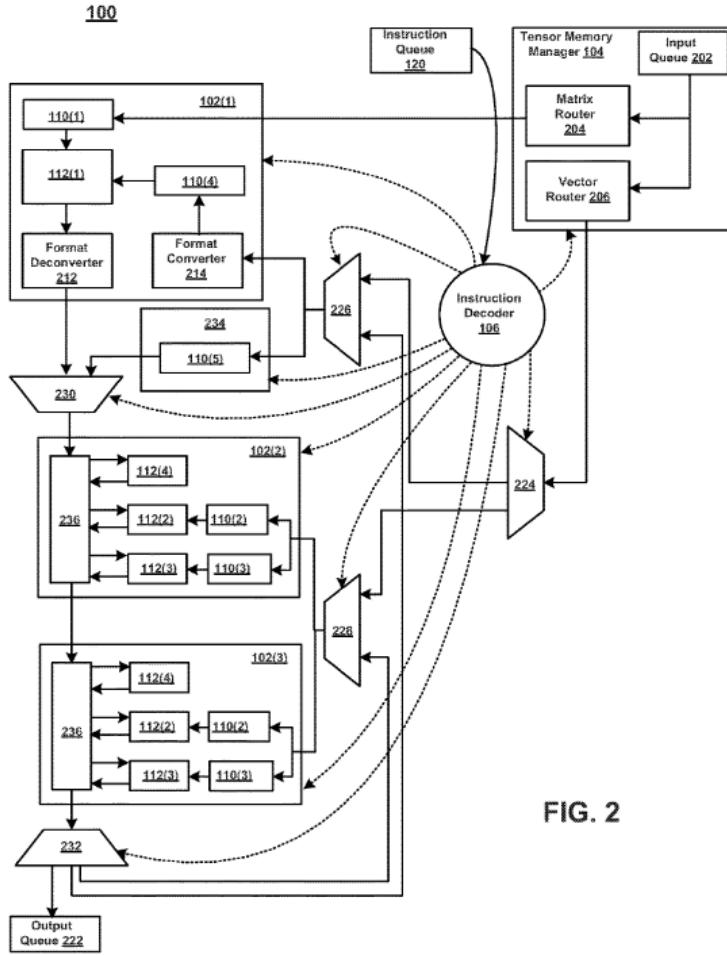


FIG. 2

## CONCLUSION

The TPU succeeded because of the large—but not too large—matrix multiply unit; the substantial software-controlled on-chip memory; the ability to run whole inference models to reduce dependence on host CPU; a single-threaded, deterministic execution model that proved to be a good match to 99th-percentile response time limits; enough flexibility to match the NNs of 2017 as well as of 2013; the omission of general-purpose features that enabled a small and low power die despite the larger data path and memory; the use of 8-bit integers by the quantized applications; and that applications were written using TensorFlow, which made it easy to port them to the TPU at high-performance rather than them having to be rewritten to run well on the very different TPU hardware. Order-of-magnitude differences between commercial products are rare in computer architecture, which may lead to the TPU becoming an archetype for domain-specific architectures. We expect that many will build successors that will raise the bar even higher.

## Works Cited

Norman P. Jouppi, Cliff Young, Nishant Patil, *In-Datacenter Performance Analysis of a Tensor Processing Unit*, 2017.

<https://patentimages.storage.googleapis.com/78/54/ba/c1e8901029486c/US20180341484A1.pdf>

<https://patentimages.storage.googleapis.com/11/74/c4/8390c55ced8356/US9836691.pdf>

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M. and Ghemawat, S., 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467

Jouppi, N. May 18, 2016. Google supercharges machine learning tasks with TPU custom chip.

<https://cloudplatform.googleblog.com>

Hennessy, J.L. and Patterson, D.A., 2018. *Computer architecture: a quantitative approach*, 6th edition, Elsevier

Shiva ram, <https://pages.cs.wisc.edu/~shivaram/cs744-fa21-slides/cs744-tpu-notes.pdf>

<https://www.kaggle.com/docs/tpu>

<https://cloud.google.com/tpu/docs/system-architecture-tpu-vm>

[https://www.tomshardware.com/news/google-tensor-processing-unit-machine-learning\\_31834.html](https://www.tomshardware.com/news/google-tensor-processing-unit-machine-learning_31834.html)