TPU Architecture

To reduce the chances of delaying deployment, the TPU was designed to be a coprocessor on the PCIe I/O bus, which allows it to be plugged into existing servers. Moreover, to simplify hardware design and debugging, the host server sends instructions over the PCIe bus directly to the TPU for it to execute, rather than having the TPU fetch the instructions. Thus the TPU is closer in spirit to an FPU (floating-point unit) coprocessor than it is to a GPU, which fetches instructions from its memory.

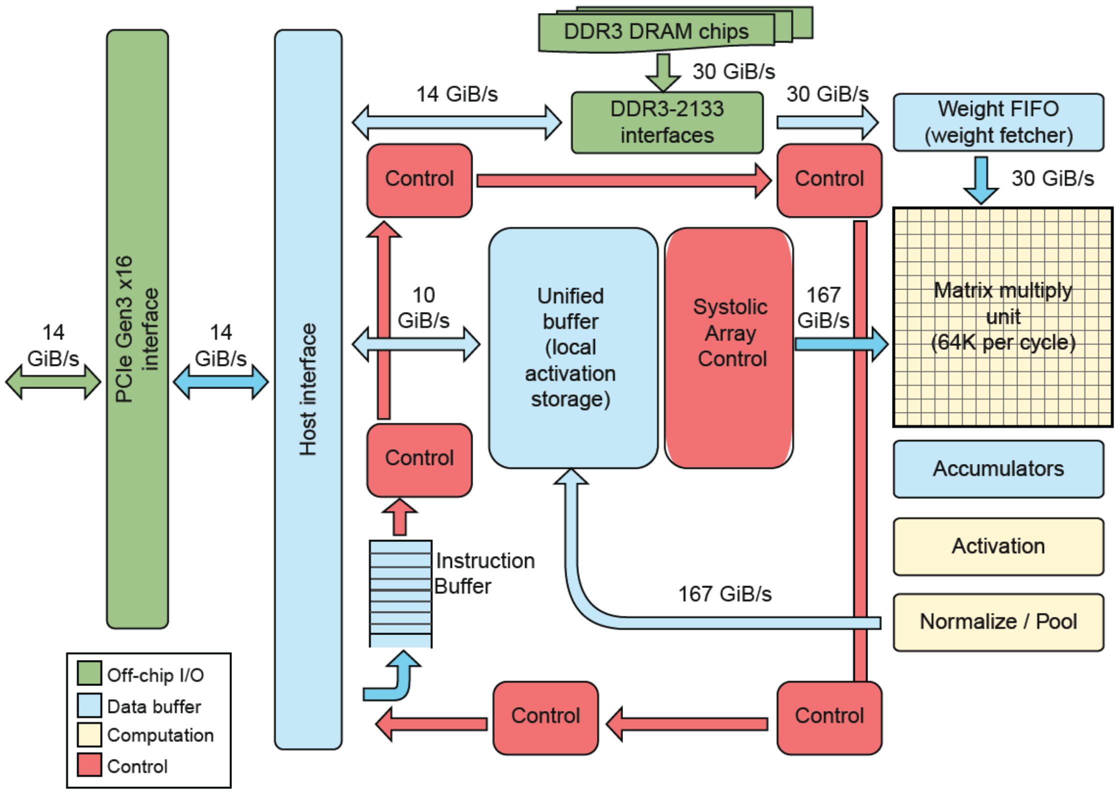

Figure 1. TPU Block Diagram

The host CPU sends TPU instructions over the PCIe bus into an *Instruction Buffer*. The internal blocks are typically connected together by 256-byte-wide (2048-bits) paths. Starting in the upper-right corner, the *Matrix Multiply Unit* is the heart of the TPU. It contains 256x256 ALUs that can perform 8-bit multiply-and-adds on signed or unsigned integers (65536 8-bit Multiply-ACcumulate units). The 16-bit products are collected in the 4 MiB of 32-bit *Accumulators* below the *Matrix Multiply Unit*.

The 4 MiB represents 4096 x 256-element x 32-bits accumulators. The matrix unit produces one 256-element partial sum per cycle (i.e.: It reads and writes 256 values per clock cycle) and can perform either a matrix multiply or a convolution. The nonlinear functions are calculated by the Activation hardware.

The weights for the *Matrix Multiply 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 simultaneous active models). The *Weight FIFO* is 4-tiles deep.

The intermediate results are held in the 24 MiB on-chip *Unified Buffer*, which can serve as inputs to the *Matrix Multiply Unit*. A *Programmable DMA Controlle*r transfers data to or from CPU Host memory and the *Unified Buffer*.

TPU Instruction Set Architecture

As instructions are sent over the relatively slow PCIe bus, TPU instructions follow the CISC tradition, including a repeat field. The TPU does not have a program counter, and it has no branch instructions; instructions are sent from the host CPU. The clock cycles per instruction (CPI) of these CISC instructions are typically 10–20. 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*.
2. **Read_Weights** reads weights from *Weight Memory* into the *Weight FIFO* as input to the *Matrix Multiply Unit*.
3. **MatrixMultiply/Convolve** causes the *Matrix Multiply Unit* to perform a matrix-matrix multiply, a vector-matrix multiply, an element-wise matrix multiply, an element-wise vector multiply, or a convolution from the *Unified Buffer* into the *Accumulators*. A matrix operation takes a variable-sized B*256 input, multiplies it by a 256 x 256 constant input, and produces a B*256 output, taking B pipelined cycles to complete. For example, if the input were 4 vectors of 256 elements, B would be 4, so it would take 4 clock cycles to complete.
4. **Activate** performs the nonlinear function of the artificial neuron, with options for ReLU, Sigmoid, tanh, 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.

TPU Microarchitecture

The microarchitecture philosophy of the TPU is to keep the *Matrix Multiply Unit* busy. The plan is to hide the execution of the other instructions by overlapping their execution with the MatrixMultiply instruction. Thus each of the preceding four general categories of instructions have separate execution hardware (with read and write host memory combined into the same unit). To increase instruction parallelism further, toward that end, the Read_Weights instruction follows the decoupled access/execute philosophy in that they can complete after sending its address but before the weights are fetched from *Weight Memory*. The *Matrix Multiply Unit* has not-ready signals from the *Unified Buffer* and the *Weight FIFO* that will cause the *Matrix Multiply Unit* to stall if the input activation or weight data are not yet available.

Because reading a large SRAM is much more expensive (power consumption) than arithmetic, the *Matrix Multiply Unit* uses systolic execution to save energy by reducing reads and writes of the *Unified Buffer*.

A systolic array is a two dimensional collection of arithmetic units that each independently compute a partial result as a function of inputs from other arithmetic units that are considered upstream to each unit. It relies on data from different directions arriving at cells in an array at regular intervals where they are combined. Because the data flows through the array as an advancing wave front, it is similar to blood being pumped through the human circulatory system by the heart, which is the origin of the systolic name.

A given 256-element multiply-accumulate operation moves through the matrix as a diagonal wave front. 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 after a feed delay (instantly), they update one location of each of 256 accumulator memories. From a correctness perspective, software is unaware of the systolic nature of the *Matrix Multiply Unit*, but, for performance, it does worry about (must account) the latency of the unit.