

Turtle TPU Final Report

Benjamin Chen
MIT
bchen27@mit.edu

Nathan Xiong
MIT
nxiong@mit.edu

William Yue
MIT
willyue@mit.edu

Abstract—We present a design for an implementation of Google’s Tensor Processing Unit (TPU) v1 on an FPGA. In particular, the TPU v1 is a vector processor designed specifically to accelerate neural network inference workloads. Its main feature is its systolic array, a hardware unit capable of delivering extremely high matrix multiplication throughput while minimizing data movement and control overhead. Our design represents a faithful recreation of the TPU v1’s architectural principles, albeit at a reduced scale for realization on an FPGA. Additionally, we extend the design to provide support for cross-device communication via bidirectional quad-SPI, thus enabling distributed inference workloads. Finally, we evaluate our design’s performance on an MNIST inference task in both the single-node and distributed settings.

Index Terms—digital systems, field programmable gate arrays, vector processors, systolic arrays

I. INTRODUCTION

Modern machine learning workloads utilize specialized hardware to accelerate their computations. The Tensor Processing Unit (TPU) designed by Google is an ASIC chip for extremely high-throughput matrix multiplications, utilizing a systolic array architecture. While Google has released many generations of the TPU and published several high-level block diagrams for its designs, much of the implementation details and even the TPU instruction set remain closed-source.

Hence, we have built the *Turtle TPU*, an RTL implementation of Google’s TPU v1, whose architecture is described in their seminal paper [1]. Due to hardware constraints, the scale of the design is shrunk to meet the requirements of the Xilinx Spartan-7 FPGA board. Nevertheless, it meets all the functional requirements of a TPU v1. Notably, it can

- Read/write input and output activations from BRAM.
- Perform matrix multiplications using a 4×4 systolic array.
- Perform vectorized activations, e.g., ReLU and scaling.
- Fetch and execute instructions in our custom ISA.

Like the TPU v1, the Turtle TPU only supports inference workloads. However, it extends the TPU v1 design by adding support for TPU-TPU communication, a dominant feature in the TPU v2 and onwards. This allows the Turtle TPU to support distributed inference, specifically for large models which cannot fit into the DRAM of one device.

We evaluate the Turtle TPU’s correctness and performance on an MNIST inference workload using a simple, quantized neural network, both in single-node and distributed settings.

II. TURTLE TPU DESIGN

The block diagram of the Turtle TPU is illustrated in Figure 1 and closely resembles the block diagram of the TPU

v1. It features the following key modules, which will be further described in the following subsections.

- **Bootstrapper.** The bootstrapper module receives weight and instruction data over UART and writes it into DRAM. Then, it enables execution for the rest of the processor.
- **Fetch.** The fetch module continuously prefetches instructions from DRAM while yielding a new instruction every cycle unless stalled.
- **Decode.** The decode module decodes instructions and enables various control signals for the execution and memory units.
- **Memory Handler.** The memory handler buffers requests to and responses from the DDR3 controller.
- **Matrix Multiply Unit.** The matrix multiply unit is a 4×4 systolic array that computes matrix multiplication on 8-bit integers.
- **Accumulators.** The accumulators hold partial sums from the matrix multiply unit and also support computing various activation functions on 4×1 vectors of 8-bit integers before writing them back to BRAM.
- **Interconnect Engine.** The interconnect engine facilitates communication and data-transfer with another Turtle TPU instance using bidirectional quad-SPI.

A. Bootstrapper

Before execution begins, the DRAM needs to be populated with the weights and instructions. Even with a very small neural network, the combined size of the weights and instructions can easily go over 100KB (since the ISA doesn’t support control flow, the instructions are typically several times larger than the weights). Hence, it’s unreasonable to store them in BRAM, necessitating a *boot phase* for the Turtle TPU.

On system reset, the bootstrapper module starts listening for data over UART. It stacks the bytes it receives into 128-bit DRAM lines and issues write requests to the memory controller. Incoming data is directly written to the next address of DRAM, so all instructions and weights are contiguous in memory. In particular, all addressing must be absolute and handled prior to boot by the compiler/assembler.

Once the bootstrapper has received a pre-determined number of bytes (the binary size is also communicated over UART), it stops listening and enables execution for the rest of the processor.

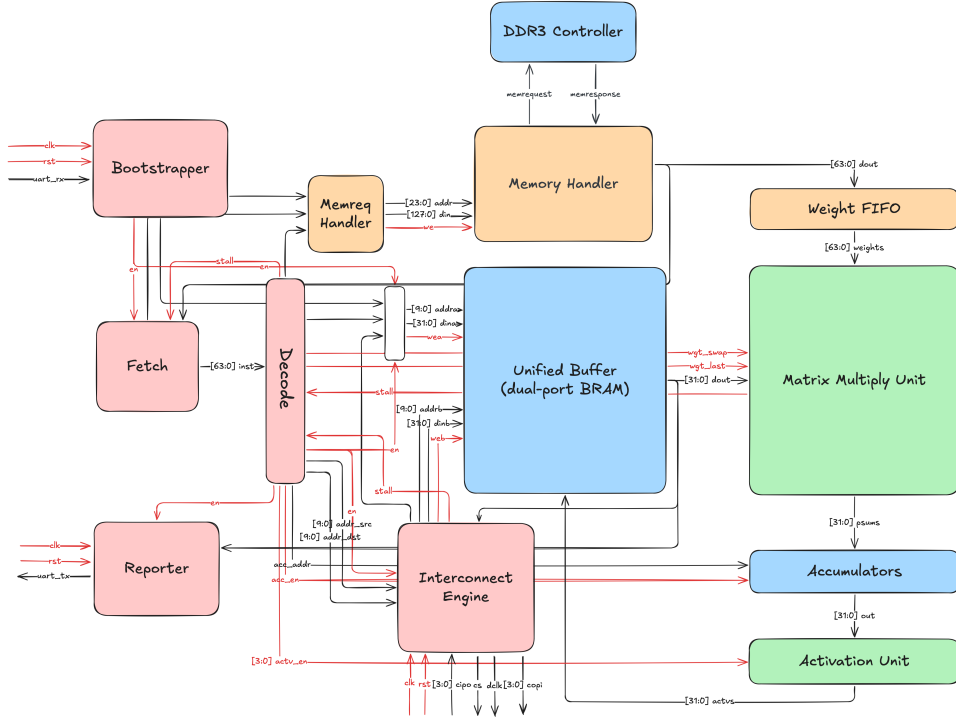


Fig. 1. Turtle TPU block diagram

B. Fetch

The fetch module loads instructions from DRAM in sequential order. Since the ISA does not support control flow, the fetch logic is much simpler than that of a regular processor, and aggressive prefetching is effective. The fetch module keeps a FIFO of prefetched instructions which the decode module reads from. It initiates prefetch commands to DRAM whenever the FIFO occupancy drops below a certain threshold.

C. Decode

The decode module decodes instructions from the fetch module's instruction buffer. Importantly, it behaves as a state machine and needs to stall for different cycle counts depending on the executing instruction. In particular, it stalls immediately upon decoding most instructions and only unstalls when the signaled by the relevant execution unit. When it stalls it also indicates to the fetch module to stall.

D. Memory Handler

The off-chip DRAM stores weights and instructions packed into 128-bit DRAM lines. The controller accepts generic read/write commands that are tagged with a message type:

- 1) MSG_BOOT: for the bootstrapper to write weight and instruction data into DRAM.
- 2) MSG_LD_WGT: for the processor to load weights from DRAM into the weight FIFO units.
- 3) MSG_LD_INST: for the fetch unit to prefetch instructions from DRAM into its own buffer.

To handle multiple modules potentially issuing memory requests in the same cycle, a memory-request handler sits in

front of the memory handler. If two DRAM requests come in the same cycle, the memory-request handler buffers the lower-priority request to send later. Only instruction prefetch requests are considered low-priority.

The memory handler itself is a wrapper over the UberDDR3 controller and keeps two FIFOs to cross clock domains (between the 100MHz processor and 83.3MHz controller) and buffer requests and responses.

E. Matrix Multiply Unit

The matrix multiply unit (MXU) is the main workhorse of the Turtle TPU. It features a 4×4 systolic array, used for performing efficient, pipelined matrix multiplication with a latency of 8 cycles and a throughput of one matrix-vector product per cycle. The systolic array consists of a 4×4 grid of processing elements, each of which performs a single INT8 fused-multiply-add (FMA) instruction.

A systolic array works by rhythmically passing inputs across the grid of processing elements and accumulating results at the bottom. First, weights are preloaded from the top of the array. Once the entire matrix of weights is loaded, an input can be loaded in from the left and the matrix-vector product will be computed. The input is staggered with delay buffers and flows left-to-right along the systolic array, while the partial sums from the FMA flow top-to-bottom. The final partial sums are the outputs, and are staggered by another set of delay buffers to synchronize latencies.

Importantly, weights are double buffered within each processing element, so one round of matrix-vector products can be ongoing (using the active buffer) while the next set of weights

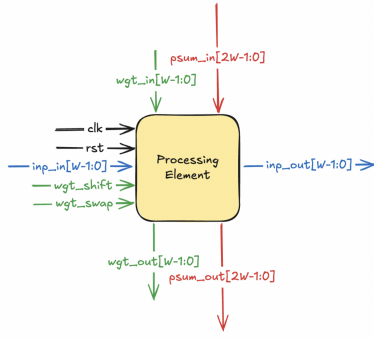


Fig. 2. Processing element.

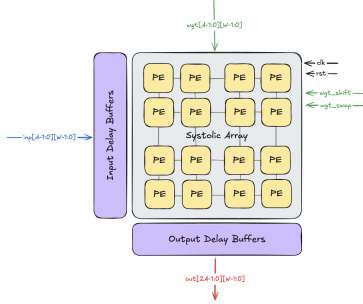


Fig. 3. Systolic array.

is being loaded (using the inactive buffer). See Figure 2 for the relevant signals in a processing element, and Figure 3 for the relevant signals of the entire systolic array.

The MXU is a small FSM that wraps the systolic array. When the decode module passes a valid signal to start running a matmul, we transition into a waiting stage. If the instruction requires us to swap a new round of weights in (via WGT_SWAP) then we won't proceed until the next set of weights is ready in the weight FIFO above (so we can populate the inactive buffer while this matmul instruction runs). The only exception is if this matmul loads in the final set of weights, as indicated by the WGT_LAST signal. The MXU also needs to wait on the input for the matmul from BRAM. It only proceeds when both weights and inputs are ready.

Then, it enters a compute stage, where the current matmul instruction is executed and, if required, the next round of weights is preloaded into the back buffer. After completion of the matmul instruction, outputs are handed off to the accumulators and we reenter an idle state.

F. Accumulators

The accumulators hold and accumulate the results from the matrix multiply unit. Since the MXU computes results in 32-bit precision, the accumulators are also stored in 32-bit precision and need to be clamped appropriately to INT8 before writing back to BRAM.

Additionally, the accumulators support the computation of various activation functions before writing back. Currently, it supports ReLU and a variable right-shift.

G. Interconnect Engine

The interconnect engine allows two Turtle TPU instances to communicate and transfer data. All large-scale inference jobs today require multi-node communication to support very large models. This typically takes the form of several “collective primitives” that are exposed in the language specification and implemented by the hardware. The Turtle TPU currently supports one collective primitive, the AllGather instruction, which is enough to enable basic *tensor parallelism* by sharding the weight matrix along the columns.

We use the bidirectional quad-SPI protocol to handle communication between the two Turtle TPU instances. Bidirectional communication allows for overlapping of sending and receiving data and is already supported within SPI. To ensure optimal throughput without exhausting the pins on the FPGA board, we opt for quad-SPI, which puts the maximum communication throughput at 8 bits per 100 cycles, or 1MB/s.

Due to the controller-peripheral nature of the SPI protocol, we actually end up with two separate but very similar RTL designs. Additionally, the controller and peripheral need to properly synchronize before initiating communication, e.g., if one Turtle TPU gets to the AllGather instruction before the other.

H. Reporter

After execution finishes (at an `exit` instruction), the reporter module transfers the output activations from BRAM back to the host. In particular, it starts reading from a pre-specified BRAM address, unstacks the 32-bit BRAM line into bytes, and transmits the bytes over UART to the host. The host is assumed to be listening on its end to capture and display the results.

III. INSTRUCTION SET ARCHITECTURE

The Turtle TPU supports a small set of CISC instructions. See the specification below. All instructions are 64 bits wide.

A. Instruction Specification

- `nop`
Opcode: 0000
Operands: None.
Description: Standard no-op.
- `ld_wgt`
Opcode: 0001
Operands: D_ADDR (24-bit).
Description: Loads weight matrix from DRAM at D_ADDR into the weight FIFO units.
- `matmul`
Opcode: 0010
Operands: B_ADDR (10-bit), ACC_ADDR (3-bit), WGT_SWAP (1-bit), WGT_LAST (1-bit), ACC_EN (1-bit).
Description: Performs a matrix multiplication on inputs loaded from BRAM at B_ADDR with the weights

currently loaded in the matrix multiply unit. The output is written to the accumulator address specified by `ACC_ADDR`. The control signal `WGT_SWAP` is set to swap in new pre-buffered weights before performing the matrix multiplication. The control signal `WGT_LAST` is set when swapping in the last set of weights. The control signal `ACC_EN` specifies whether to accumulate or override the old values on output.

- `activate`

Opcode: 0011

Operands: `B_ADDR` (10-bit), `ACC_ADDR` (3-bit), `ACTV_EN` (4-bit), `ACTV_DATA` (32-bit).

Description: Performs activation functions on the accumulators specified by `ACC_ADDR` and writes the output to BRAM at address `B_ADDR`. `ACTV_EN` is a one-hot vector for different activation functions that can be enabled; currently, only ReLU and right-shift are supported. The shift amount is stored in the lower 8 bits of `ACTV_DATA`.

- `exit`

Opcode: 0101

Operands: None.

Description: Indicates end of program, initiating write-back of output activations to host.

- `all_gather`

Opcode: 1000

Operands: `B_ADDR1` (10-bit), `B_ADDR2` (10-bit), `XFER_SZ` (10-bit).

Description: Synchronizes two Turtle TPU instances, and then initiates a copy of data from source address `B_ADDR1` of size `XFER_SZ` while simultaneously writing incoming data to destination address `B_ADDR2`. The unit of transfer is 32 bits. Note that two corresponding `all_gather` instructions can have different source/destination addresses but must share the same transfer size.

B. Instruction Latency

Almost all the instructions require more than 1 cycle of latency. In particular,

- `nop` has 1-cycle latency.
- `ld_wgt` has 2-cycle latency (more if the systolic array is larger). It does not wait for the DRAM request to complete.
- `matmul` has 12-cycle latency, accounting for both the systolic array and the delay buffers.
- `activate` has 4-cycle latency, since it needs to wait for the writeback to BRAM.
- `exit` has 1-cycle latency.
- `all_gather` has variable latency, depending on the amount of data to be sent and how long it takes to synchronize. Ignoring synchronization time, it takes roughly 900 cycles to send 1 BRAM line (32 bits) at a data clock period of 100 cycles.

IV. EVALUATION

We evaluate the performance of the Turtle TPU on a small inference workload using a quantized neural network for the MNIST dataset.

First, we train a 2-layer neural network with a hidden dimension of 32. This produces two weight matrices: one with dimensions 784×32 and one with dimensions 32×10 . After training, we apply symmetric quantization to reduce all weights to INT8 values.

Next, we write a program in our custom ISA to perform the forward pass. Since the program has to map exactly onto the hardware (just like any high-performance kernel on a hardware accelerator), all matrix dimensions need to be multiples of 4. Hence, the latter matrix is actually expanded to 32×12 and padded with zeros. This brings the total size of the weights to $784 \times 32 + 32 \times 12 = 25472$ bytes. To make efficient use of the hardware, the forward pass program should also optimally overlap memory and compute resources. This leads us to pipeline the `ld_wgt` instructions with the `matmul` instructions, ensuring that the systolic array is running as frequently as possible. Also, since our ISA does not support control flow, the program must be a sequential stream of instructions. In particular, it ends up being quite large, at 3244 instructions or 25952 bytes. This brings the total data stored in DRAM to 51424 bytes.

In the distributed setting, we use two Turtle TPU instances connected via the quad-SPI signals. To utilize *tensor parallelism*, we split both weight matrices in two along the columns, so that each board only needs to hold a 784×16 weight matrix and 32×6 weight matrix. Again, since matrix dimensions must be a multiple of 4, the second matrix needs to be expanded to 32×8 and padded with zeros. Hence, each Turtle TPU has $784 \times 16 + 32 \times 8 = 12800$ bytes in weights. Furthermore, each Turtle TPU has a shorter program now, since they each operate on smaller matrices. In particular, each program is now just 1618 instructions long, or 12944 bytes. This brings the total data stored in DRAM for each Turtle TPU to 25744 bytes.

A. Correctness

Both single-node and distributed setups arrive at the correct result. We checked the resulting logits against a Python script to ensure that the Turtle TPU was indeed performing the computation correctly.

B. Performance

1) *Single-node*: Ignoring boot and report times (which just consist of UART transfers), the single-node setup finishes in approximately 37,000 cycles. At a 100MHz clock, this is 370us. Recall that our problem consists of two matrix multiplications, which combined require roughly 50,000 INT8 FLOPs. From these numbers, the single-node setup achieves an INT8 throughput of 135 MFLOP/s.

2) *Distributed*: Ignoring boot and report times, the distributed setup finishes in approximately 24,000 cycles. At a 100MHz clock, this is 240us. Note that in the distributed case, if the two Turtle TPU instances don't start at the same time,

then the earlier one will need to wait for the later one to catch up during the synchronization phase of the AllGather; hence, we measure the cycle count for the later-starting one. Again, our problem consists of roughly 50,000 INT8 FLOPs, which means the distributed setup achieves a throughput of 208 MFLOP/s.

C. Discussion

The peak compute throughput of the Turtle TPU is given by the throughput of the systolic array. With our 4×4 systolic array, each of the 16 processing elements can perform 2 FLOPs per cycle, so the peak compute throughput is 3.2 GFLOP/s. Hence, the single-node setup is currently only at 1/23 of the peak compute throughput.

However, we can also analyze the memory. The program needs to load 51424 bytes from DRAM over its lifetime (it never writes to DRAM). Hence, the memory bandwidth achieved in the single-node setup is 139 MB/s. The peak memory bandwidth of our DDR3 memory is 1.33 GB/s, so we achieve 1/10 of the peak memory throughput.

With a larger systolic array, we should be able to achieve much better compute throughput.

D. Utilization

The Turtle TPU uses

- 11453 LUTs (35.13% utilization)
- 23519 Flip-Flops (36.07% utilization)
- 62 LUTRAMs
- 5 36-kilobit BRAM tiles and 2 18-kilobit BRAM tiles, for 216 kilobits of BRAM total.
- 0 DSP blocks

The design meets timing requirements with a slack of WNS = 0.127ns.

These number will change dramatically if the size of the systolic array is varied, because most of the LUTs and Flip-Flops belong to the matrix multiply unit.

V. INDIVIDUAL CONTRIBUTIONS

- Ben wrote much of the DRAM memory handling code as well as the quad-SPI interface for the interconnect engine.
- Nathan wrote the boot modules, fetch and decode logic, and also helped with fighting DRAM.
- William wrote the main systolic array and its surrounding modules. He also trained the quantized neural network and wrote the forward pass program.

VI. CODE

Code can be found at <https://github.mit.edu/6205F25/fa25-6205-team15>.

REFERENCES

- [1] X. Ji, *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," ISCA '17: Proceedings of the 44th Annual International Symposium on Computer Architecture, June. 2017.