Lecture

Domain Specific Computing CASE Study: TPU

Computer Architecture A Quantitative Approach v.6

Hennessey and Patterson

Chung-Ho Chen

NCKU EE

General Purpose Computing Processor

- Moore's Law enabled:
  - 1st-level, 2nd-level, 3rd-level, and even 4th-level caches
  - 512-bit SIMD floating-point units
  - 15+ stage pipelines
  - Branch prediction
  - Out-of-order execution
  - Speculative prefetching
  - Multithreading
  - Multiprocessing

Introduction

- Domain Specific Architecture
  - A special-purpose processor designed for a particular domain. It relies on other processors to handle processing outside that domain.
- DSA feature
  - Use dedicated memories to minimize data movement
    - Software-controlled scratchpad memory
  - Invest resources into more arithmetic units or larger on-chip memories
  - Use the easiest form of parallelism that matches the domain
    - Use SIMD or VLIW if it works well other than OOO or MIMD
  - Reduce data size and type to the simplest needed for the domain
    - Use Int 8, for instance
  - Use a domain-specific programming language
    - TensorFlow

Popular Types of DNN

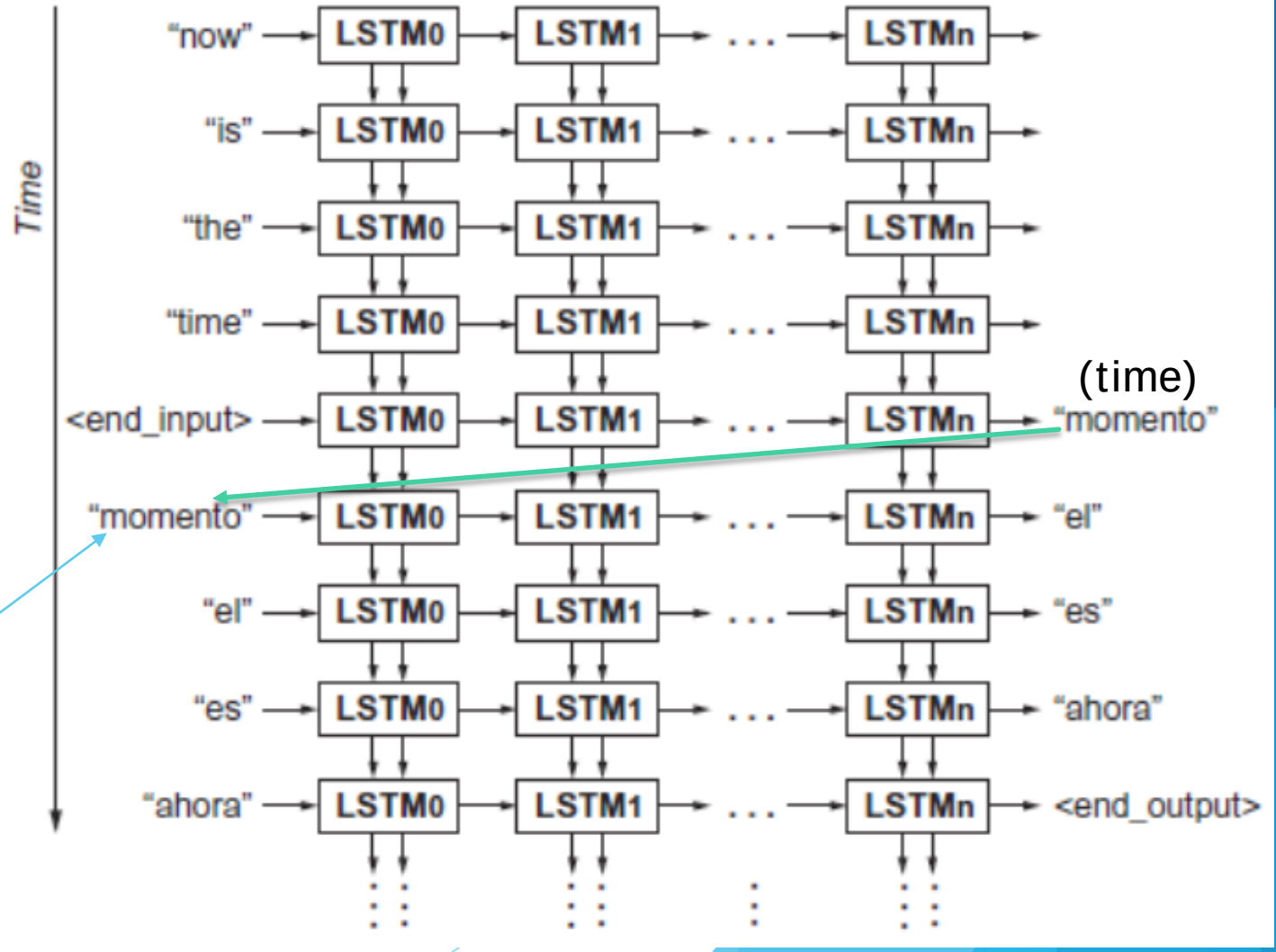
- Multilayer perceptron
- Convolutional neural network
- Recurrent neural network

Recurrent Neural Network

- Application : speech recognition or language translation
- RNN adds “state” to the DNN model so that it remembers facts. (Like state machine in logic system)
- Each layer of an RNN is a
  - Collection of weighted sums of inputs from the prior layer and the previous state.
  - The weights are reused across time steps.
- Exemplar RNN: Long short-term memory (LSTM)
  - Cells are templates that are linked together to create the full DNN.

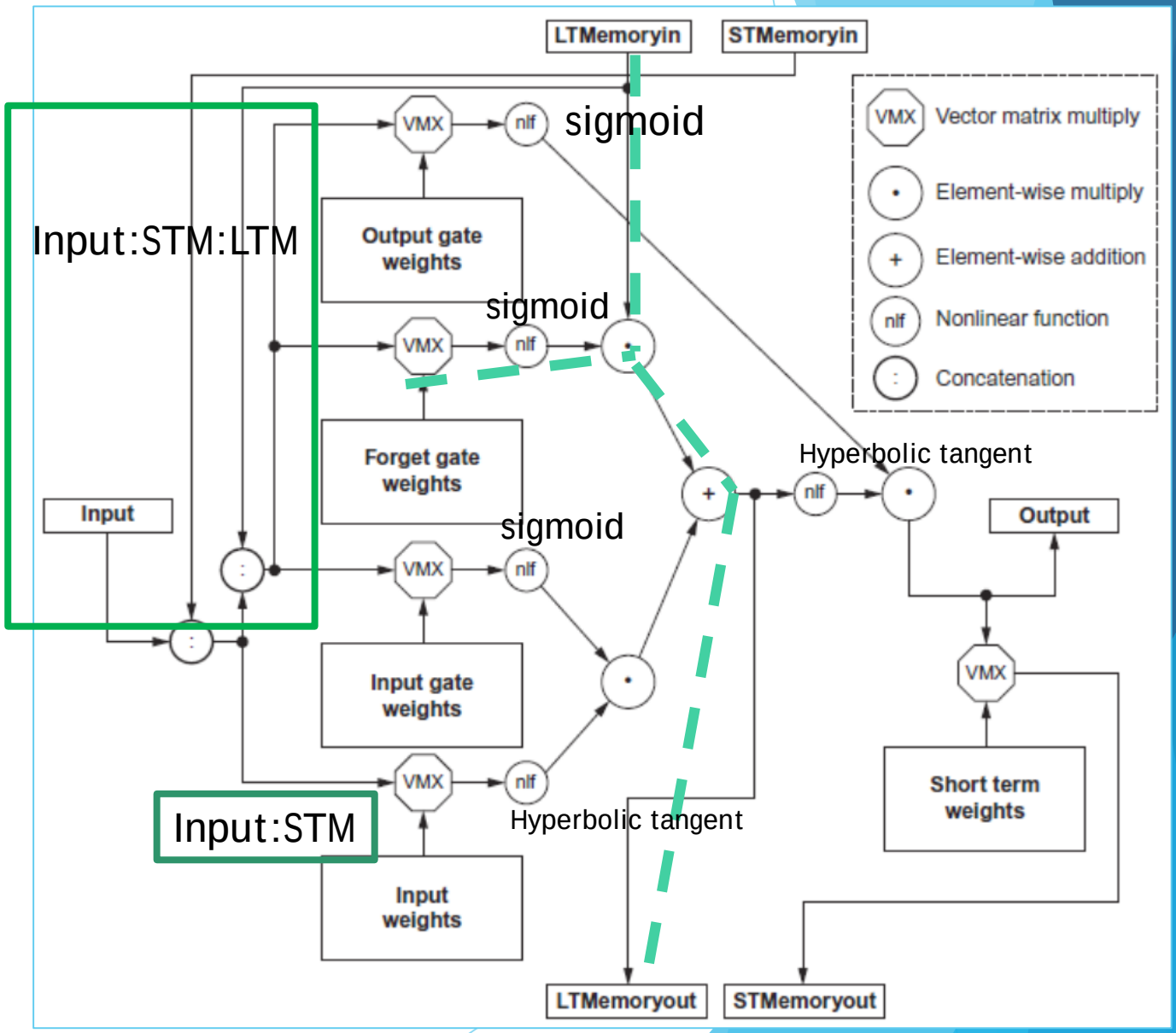
RNN with LSTM Cells

- Translation: English to Spanish
- From left to right, connecting the output of one cell to the input of the next
- Unrolled in time, runs top down (Y: time axis), unrolled rows = sentence word count
- Input a word at a time per iteration
- Translation produced in reverse order
  - Output is delayed until the end of input
  - Use the most recent translated word as the input to the next step



LSTM Cell

- Input on the left, output on the right
- Two memory inputs: Long term Memory, short term memory
- Two memory outputs: LTMemoryout, STMemoryout
- 5 vector matrix multiplications, VMX
- Input gate, forget gate, output gate limits how much information one source is passed to standard output or memory output.
- Forget gates determines what to forget along the long term memory path.
- STMemoryout = **output of this cell** VMX short term weights. (does not directly use any of the inputs to the cell)
- Same size for the 3 input-output pairs.



Computing Efficiency

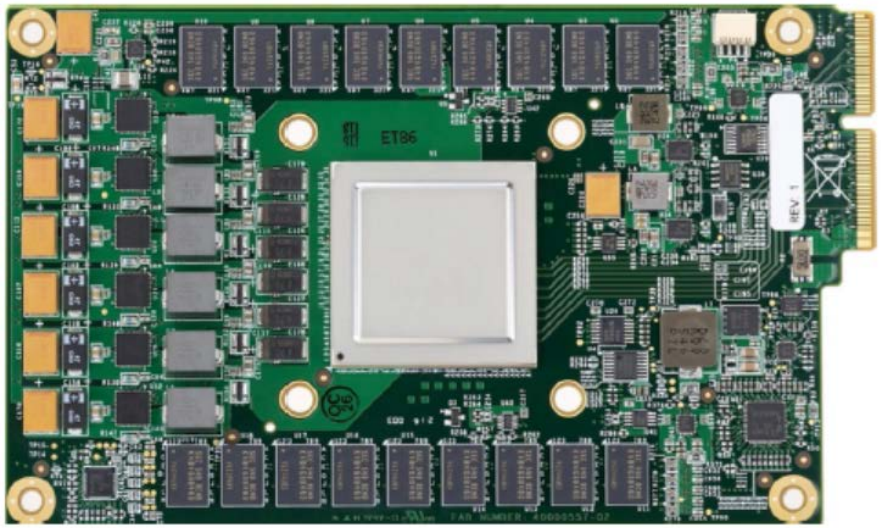
- Operation density: Operations/weight
  - Low: LSTM, MLP (Multilayer perceptron)
  - Higher: CNN

Name	DNN layers	Weights	Operations/Weight
MLP0	5	20M	200
MLP1	4	5M	168
LSTM0	58	52M	64
LSTM1	56	34M	96
CNN0	16	8M	2888
CNN1	89	100M	1750

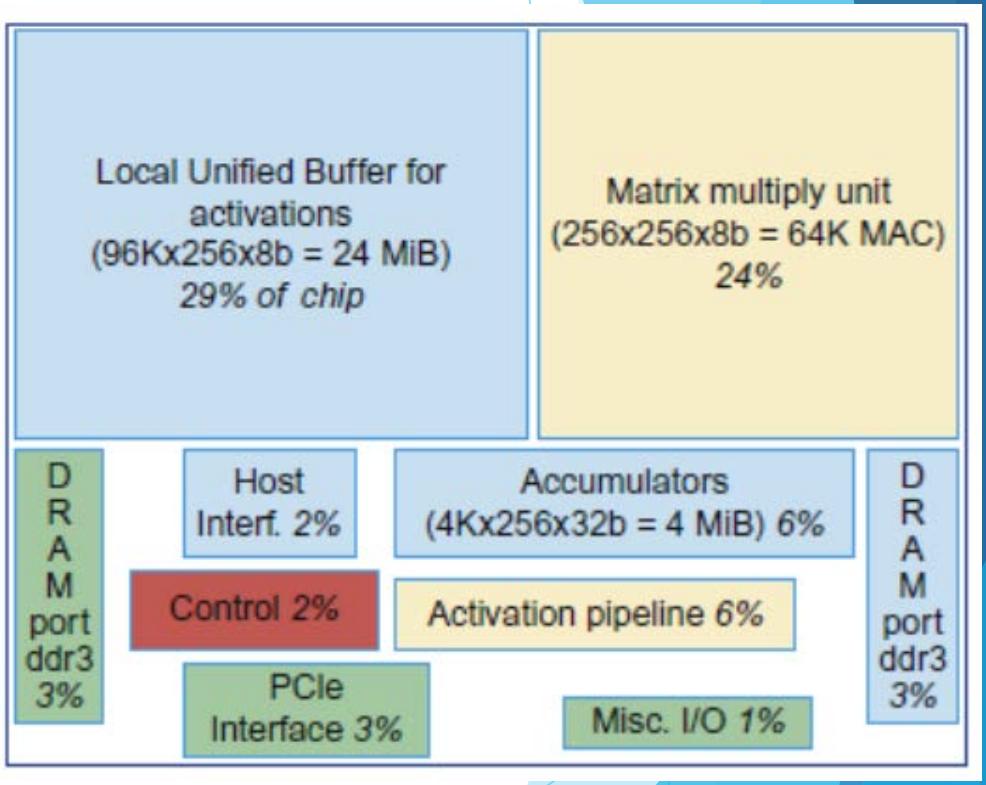
- Quantization
  - Fixed point, int8, data type for inference
- Matrix-oriented operations
  - Vector-matrix multiply, matrix-matrix multiply, stencil computation (convolution)
  - Non-linear functions
- Data reuse

Google's Tensor Processing Unit

- First, an inference data center accelerator (2015)
- Google's DNN ASIC
- 256 x 256 8-bit matrix multiply unit
- Large software-managed scratchpad
- Coprocessor on the PCIe bus
- TPU chip (v1): 28-nm, 700MHz
- Edge TPU (2018)

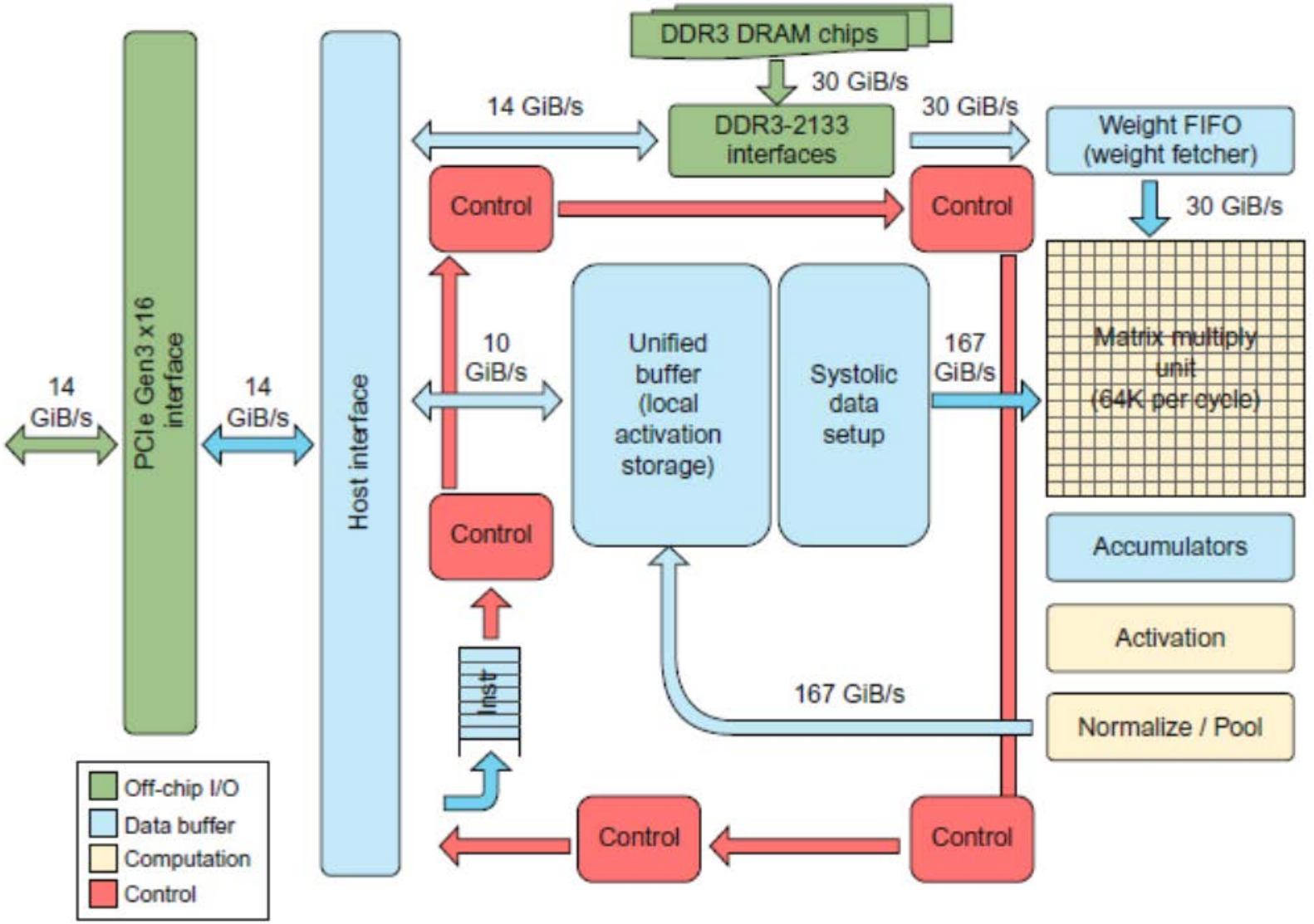


Floor plan of TPU die



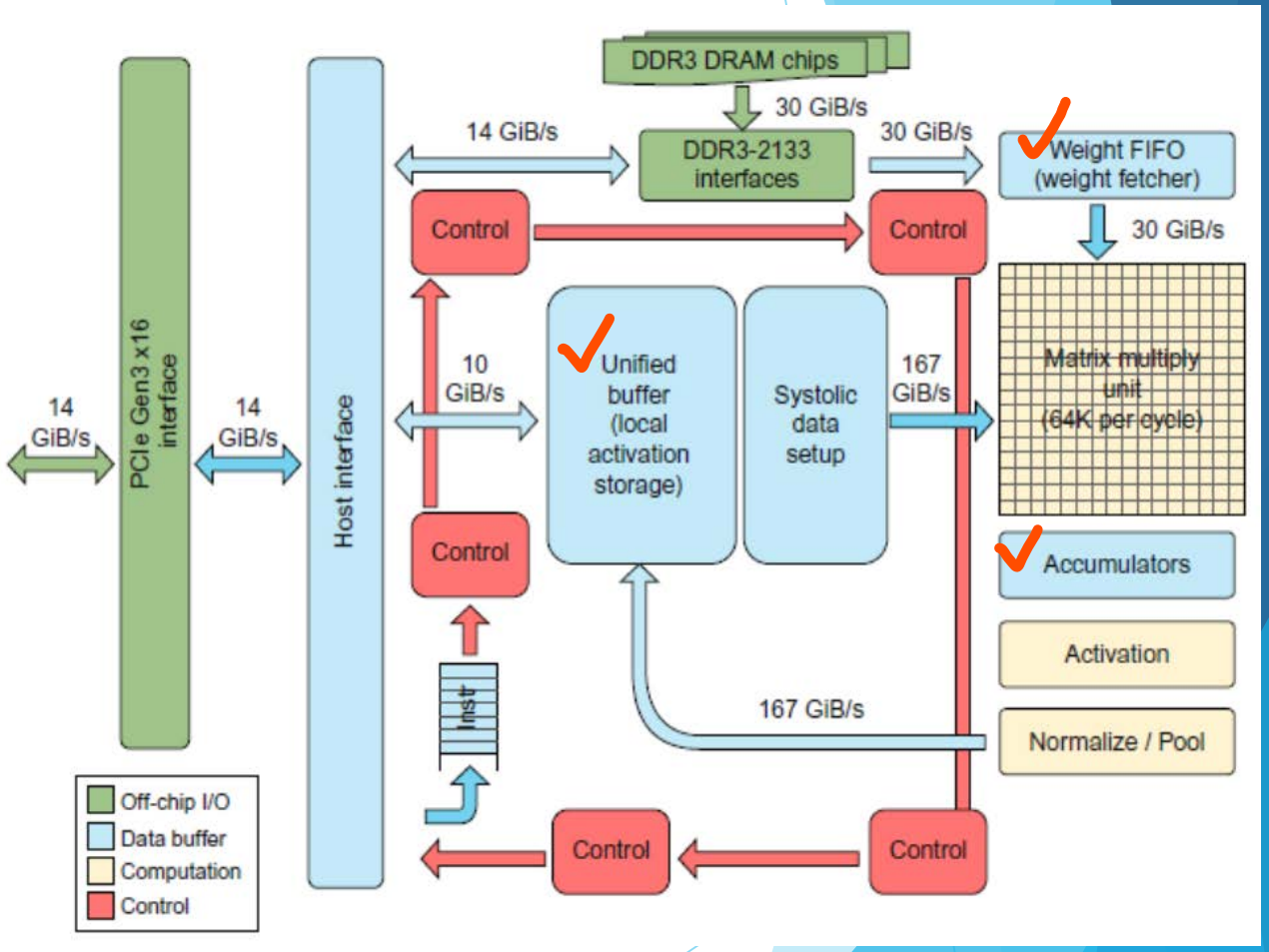
TPU Matrix Multiply Unit

- 256 x 256 ALU: 8-bit MAC on signed or unsigned integers
- MAC output: 16-bit products stored in 4MiB 32-bit Accs
- MiB: *mebi* byte means 2^{20}
- MB: *mega* byte now means 10^6
- MAC supports: 8 bits x 8 bits, 8 bit x 16 bits (half speed), 16 bits x 16 bits (quarter speed)
- MAC reads 256 values/clock and performs matrix multiply or a convolution



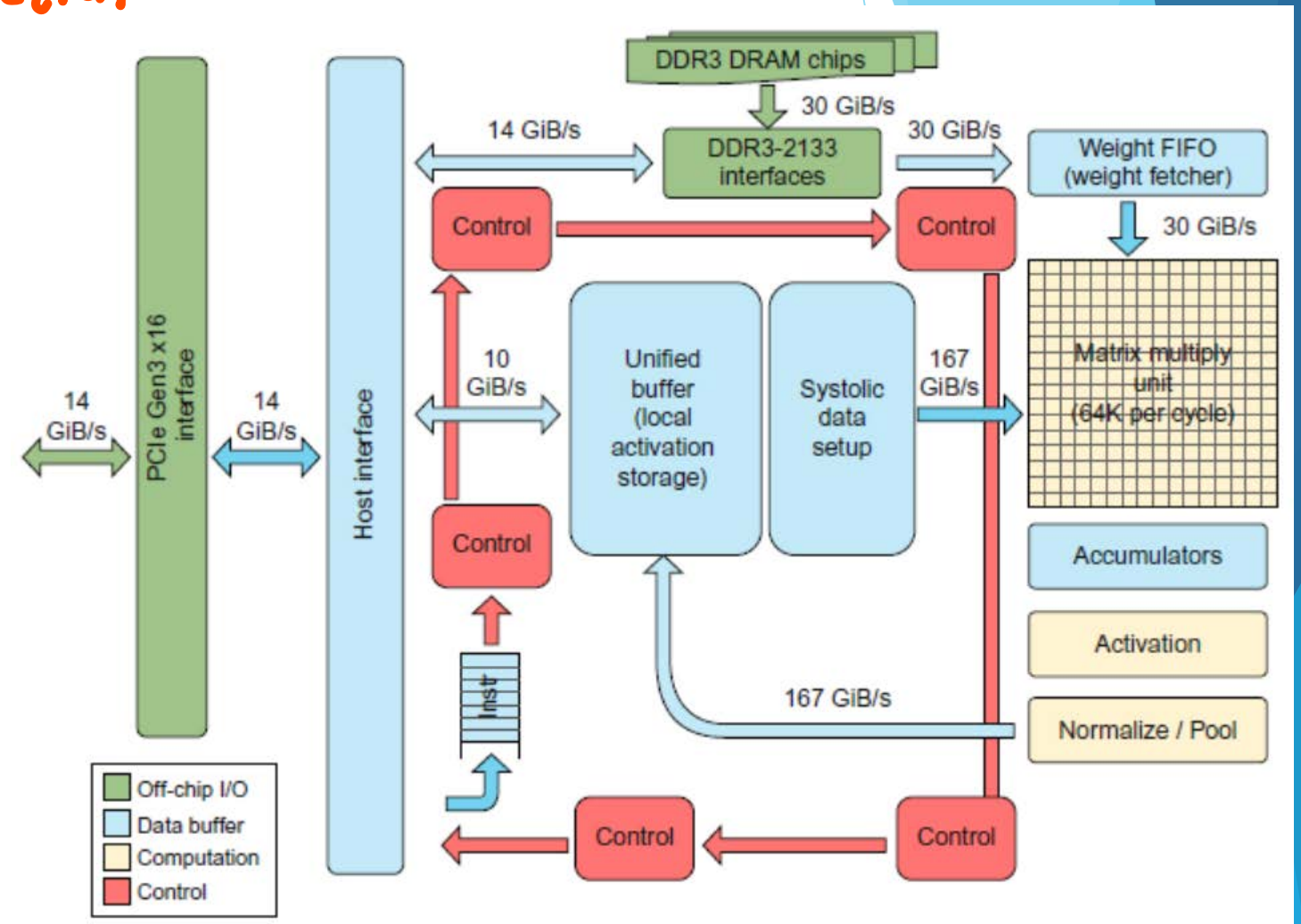
TPU Datapath

- On-chip weight FIFO that reads from off-chip 8GiB DRAM of weight memory (8 GiB support many active models)
- Unified buffer (24 MiB): buffer for input feature maps (served by a programmable DMA) and intermediate results. Designed for 256-byte access at a time.
- Accumulators (4MiB?), each 32-bits wide, collect output of the MAC unit and act as input to the Activation unit. (2^8 x 2^8 x 2^5 = 2MiB)
- Nonlinear function computed in Activation hardware



Key TPU ISA (Commands)

- CISC-like instruction sent from Host CPU, a dozen instructions overall
- Read_Host_Memory
  - Reads memory from the CPU memory into the unified buffer
- Read_Weights
  - Reads weights from the Weight Memory into the Weight FIFO as input to the Matrix Unit
- MatrixMatrixMultiply/Convolve
  - Performs 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
  - Takes a variable-sized B*256 input, multiplies it by a 256x256 constant input, and produces a B*256 output, taking B pipelined cycles to complete
- Activate
  - Computes activation function. Its inputs are the Accumulators, and its output is the Unified Buffer.
- Write_Host_Memory
  - Writes data from unified buffer into host memory



TPU ISA (cont.)

- Set configuration
- Synchronization
- Interrupt host
- Debug message
- Nop
- Halt

Example

CISC MatrixMultiply instruction: 12 bytes, of which 3 are Unified buffer address, 2 are accumulator address, 4 are length (or 2 dimensions for convolution), and opcode and flags.

A TPU instruction can execute for many clock cycles.

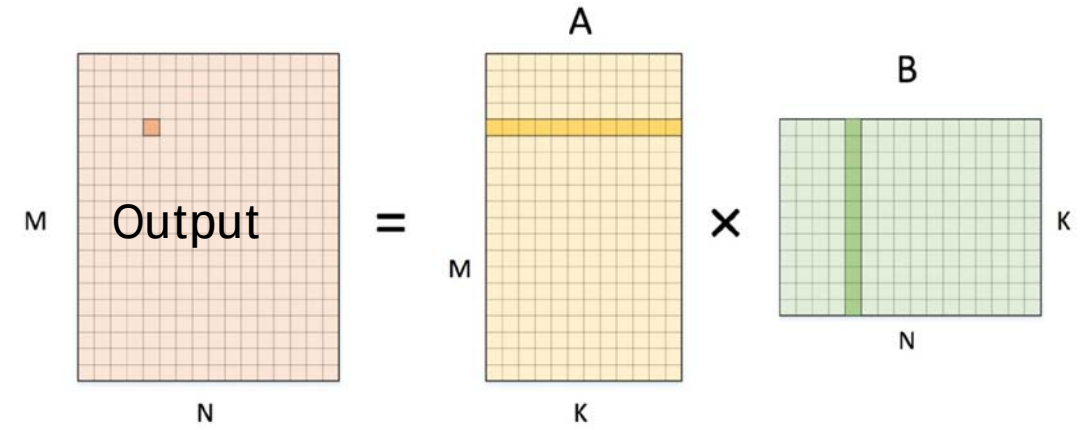
TPU Microarchitecture

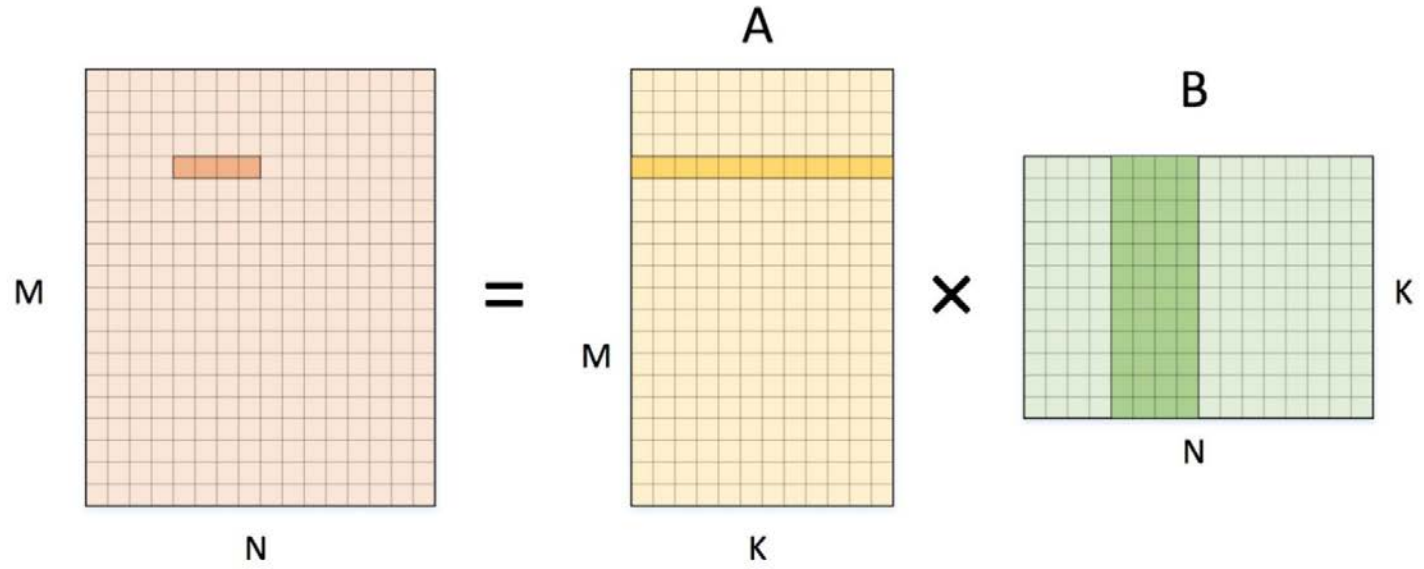
- Execute units
  - Each of the preceding four generation categories of instructions have separate execution hardware (read/write host memory combined into the same unit)
- Hardware optimizations
  - Read_weights instruction: complete when addresses are sent before the data weights return from the Weight memory (DRAM), load buffer concept.
  - Matrix unit has not-ready signals from Unified buffer and Weight FIFO if their data are not yet available

Matrix Multiply

An input row of A (K data) x a column weight of B (K data)

- Systolic array for MM
- GEMM: General Matrix Multiply





Systolic Array

weight stationary

- To reduce the SRAM reads/writes from the Unified Buffer
- 6 weights in the MAC units (weight location = MAC location)
- 3 inputs coming the top (in TPU, come in from the left)
  - The goal is to get this:

$$y_1 = w_{11}x_1 + w_{12}x_2 + w_{13}x_3$$

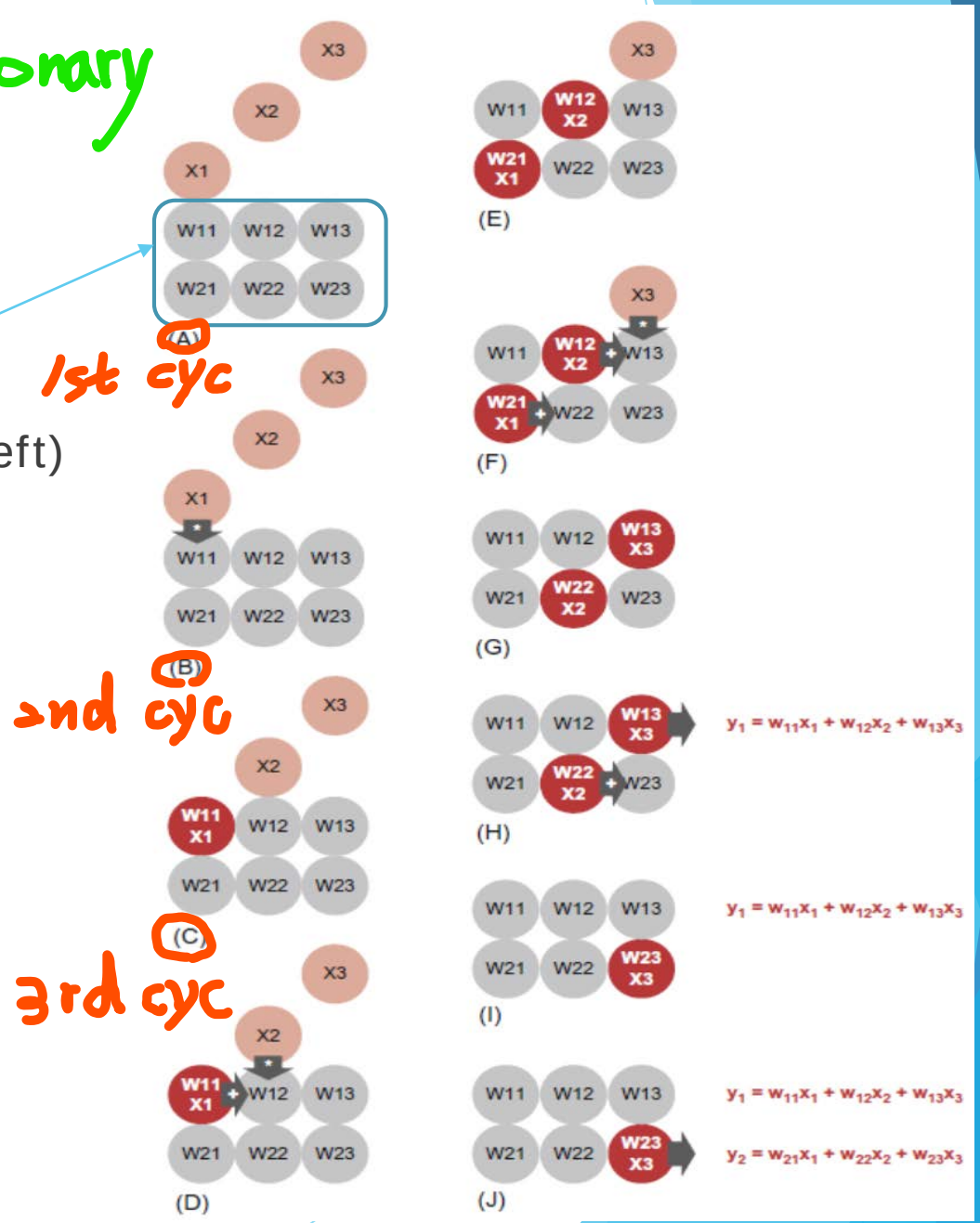
$$y_2 = w_{21}x_1 + w_{22}x_2 + w_{23}x_3$$

A: Input X1 vertically runs through W11 and W21

B: When X2 arrives at W12 and times W12, it adds W11X1 from the left, which was computed in the previous cycle.

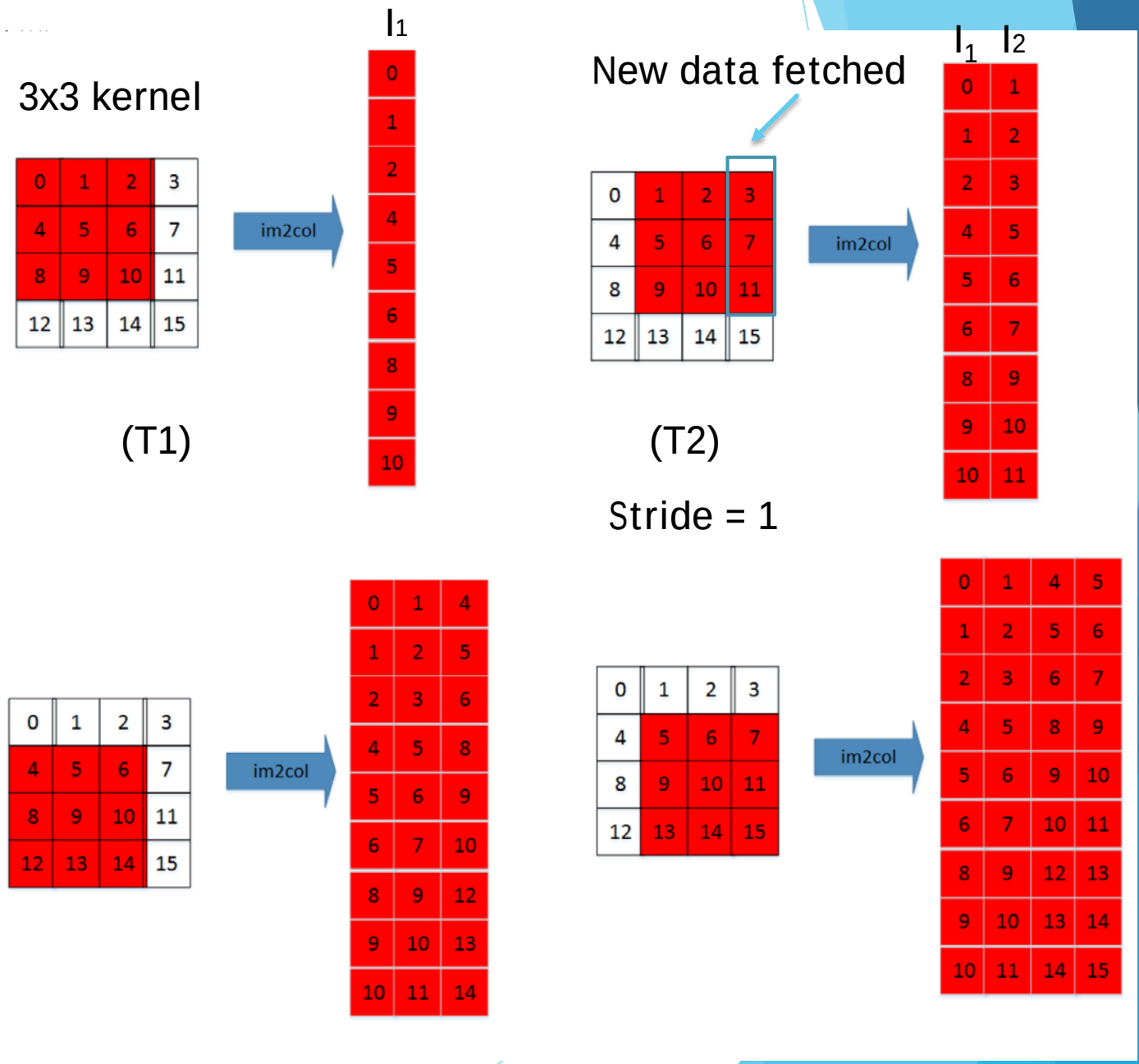
C: Result (= partial sum for y_i) goes to left and adds with local

D: Inputs all go down.



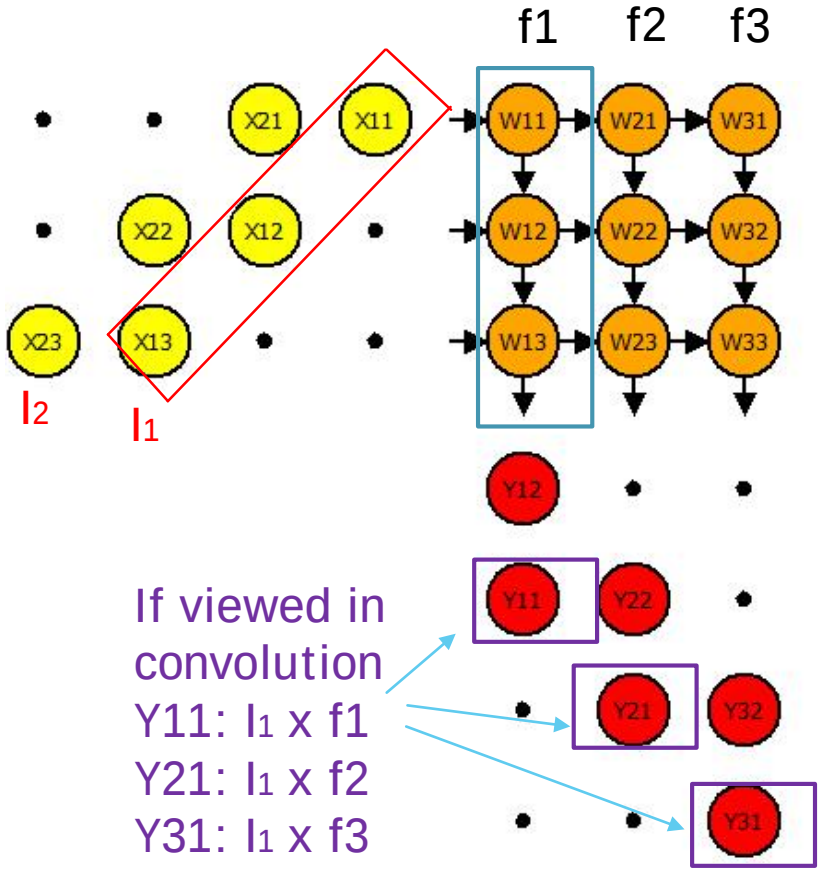
im2col

- 2D input array to one column linear representation
- Mapping this to Systolic Array, say 256x256
- Issues
  - Most of the kernel is often sized of 3x3, 5x5, 7x7, etc., much smaller than 256.
  - How to reuse input data? At T2, reuse data at cell location of 1,2,5,6,9,10.



TPU

- Weight stationary: Preloaded from the top
- TPU highlights a 256-element multiply-additions.



Y11 = W11 X11+W12 X12+W13 X13

Y12 = W11 X21+W12 X22+W13 X23

Y21 = W21 X11+W22 X12+W23 X13

Y22 = W21 X21+W22 X22+W23 X23

Y31 = W31 X11+W32 X12+W33 X13

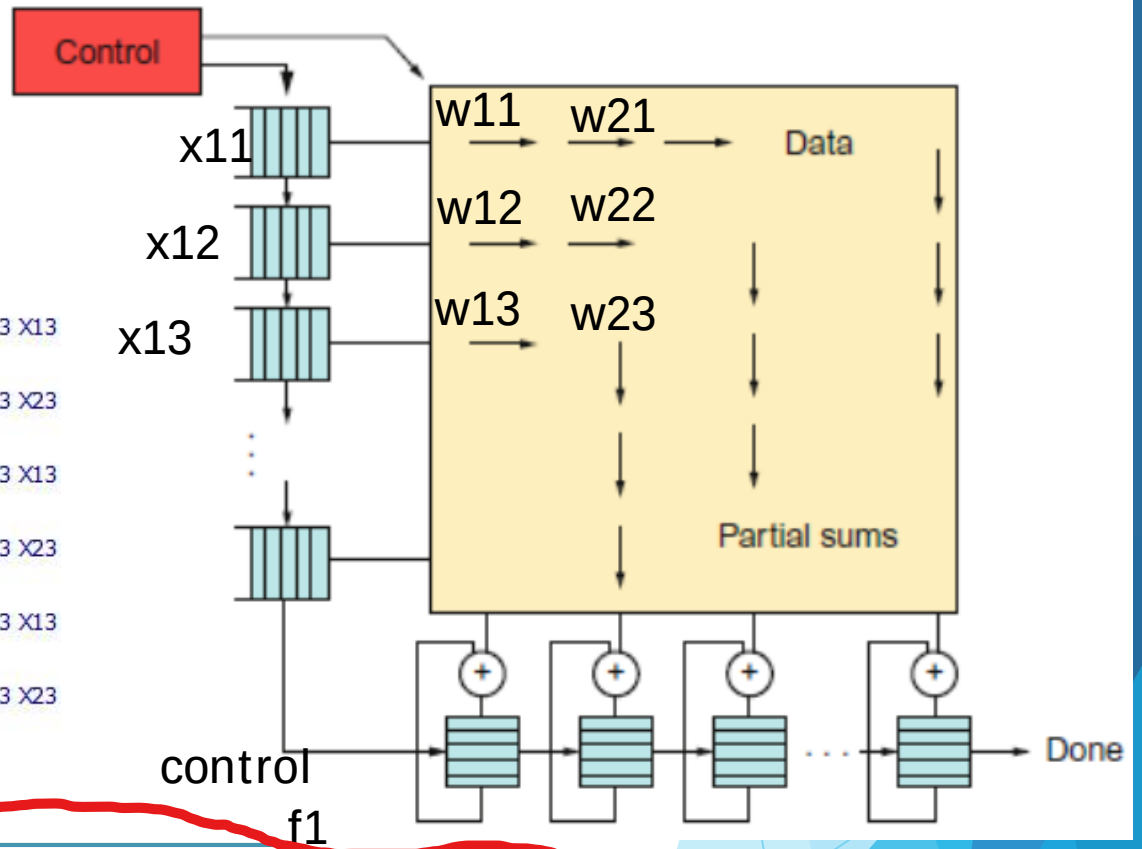
Y32 = W31 X21+W32 X22+W33 X23

If viewed in convolution

Y11: I_1 x f1

Y21: I_1 x f2

Y31: I_1 x f3



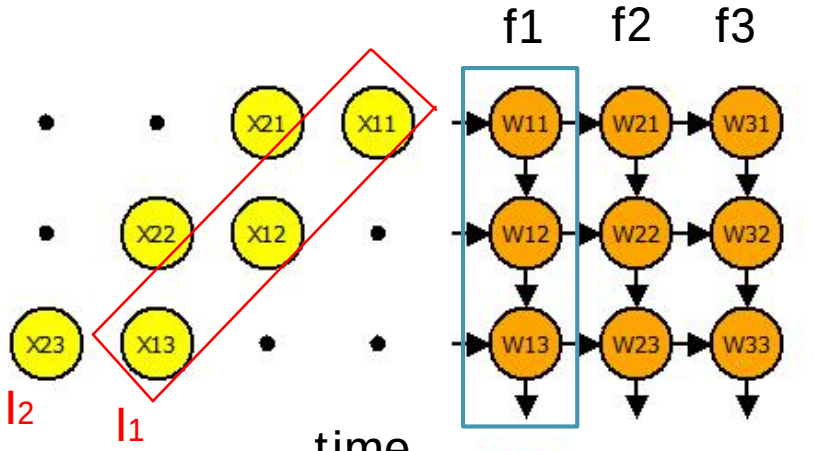
f1

Y11	Y12		w11 w12 w13		x11 x21
Y21	Y22	=	w21 w22 w23	X f2	x12 x22
Y31	Y32		w31 w32 w33	f3	x13 x23
					I_1 I_2

Note: X is denoted differently

TPU

- Weight stationary: Preloaded from the top
- TPU highlights a 256-element multiply-additions.



Y11 = W11 X11+W12 X12+W13 X13

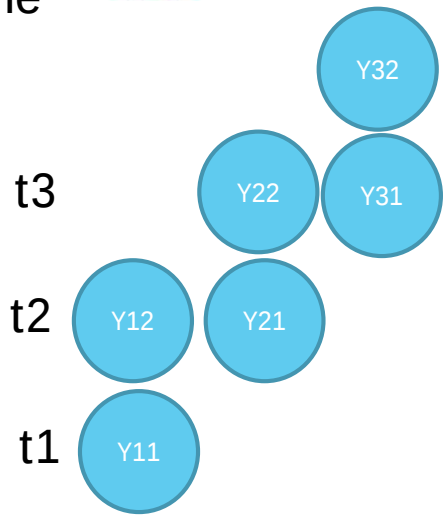
Y12 = W11 X21+W12 X22+W13 X23

Y21 = W21 X11+W22 X12+W23 X13

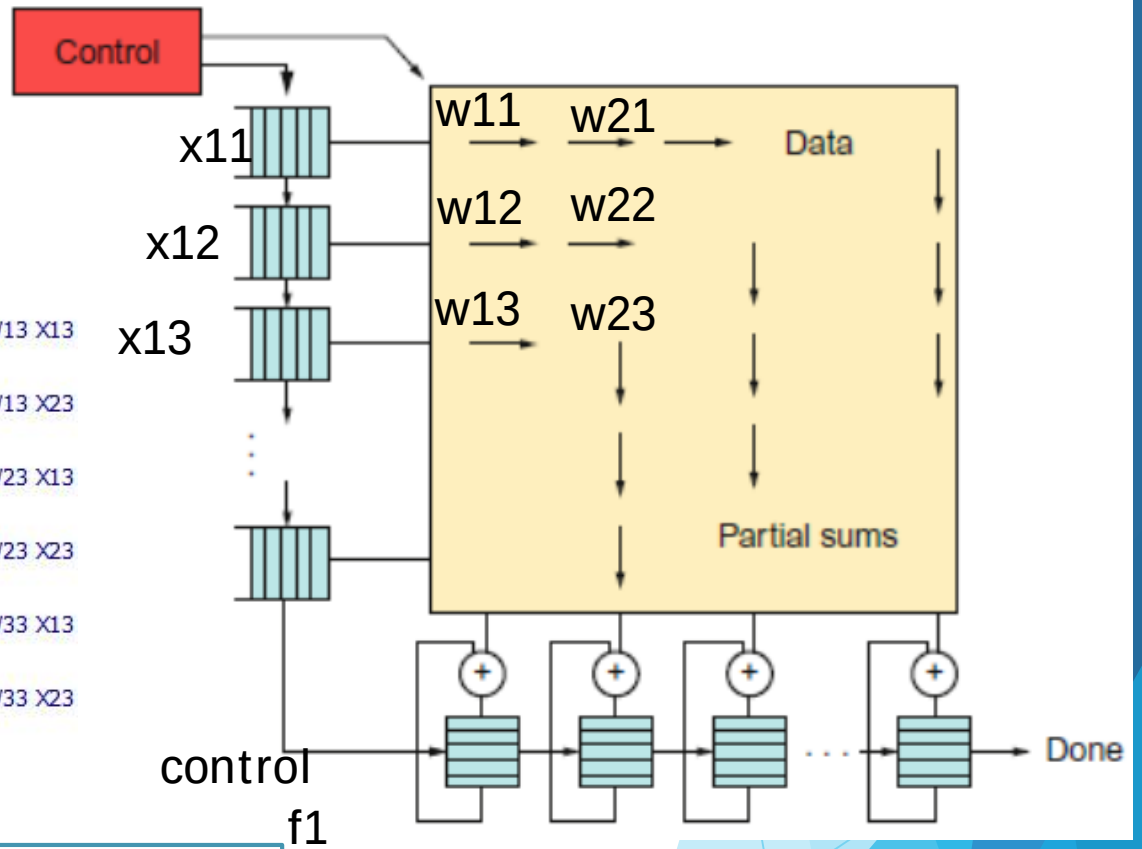
Y22 = W21 X21+W22 X22+W23 X23

Y31 = W31 X11+W32 X12+W33 X13

Y32 = W31 X21+W32 X22+W33 X23



If viewed in convolution
Y11: I_1 x f1
Y21: I_1 x f2
Y31: I_1 x f3



$$\begin{bmatrix} Y11 & Y12 \\ Y21 & Y22 \\ Y31 & Y32 \end{bmatrix} = \begin{matrix} f1 \\ f2 \\ f3 \end{matrix} \begin{bmatrix} w11 & w12 & w13 \\ w21 & w22 & w23 \\ w31 & w32 & w33 \end{bmatrix} X \begin{matrix} I_1 \\ I_2 \end{matrix} \begin{bmatrix} x11 & x21 \\ x12 & x22 \\ x13 & x23 \end{bmatrix}$$

Note: X is denoted differently

Volume Convolution Using MM on Systolic Array

- N channels of Input * N channels of a filter
- A total of M filters
- What is I_2? Stride?
- There are lots of "the same" data appeared in I_1 and I_2 respectively.
- Original data size for input
  - (H x W) x N
- Im2col
- R and Z depend on Stride and Filter size

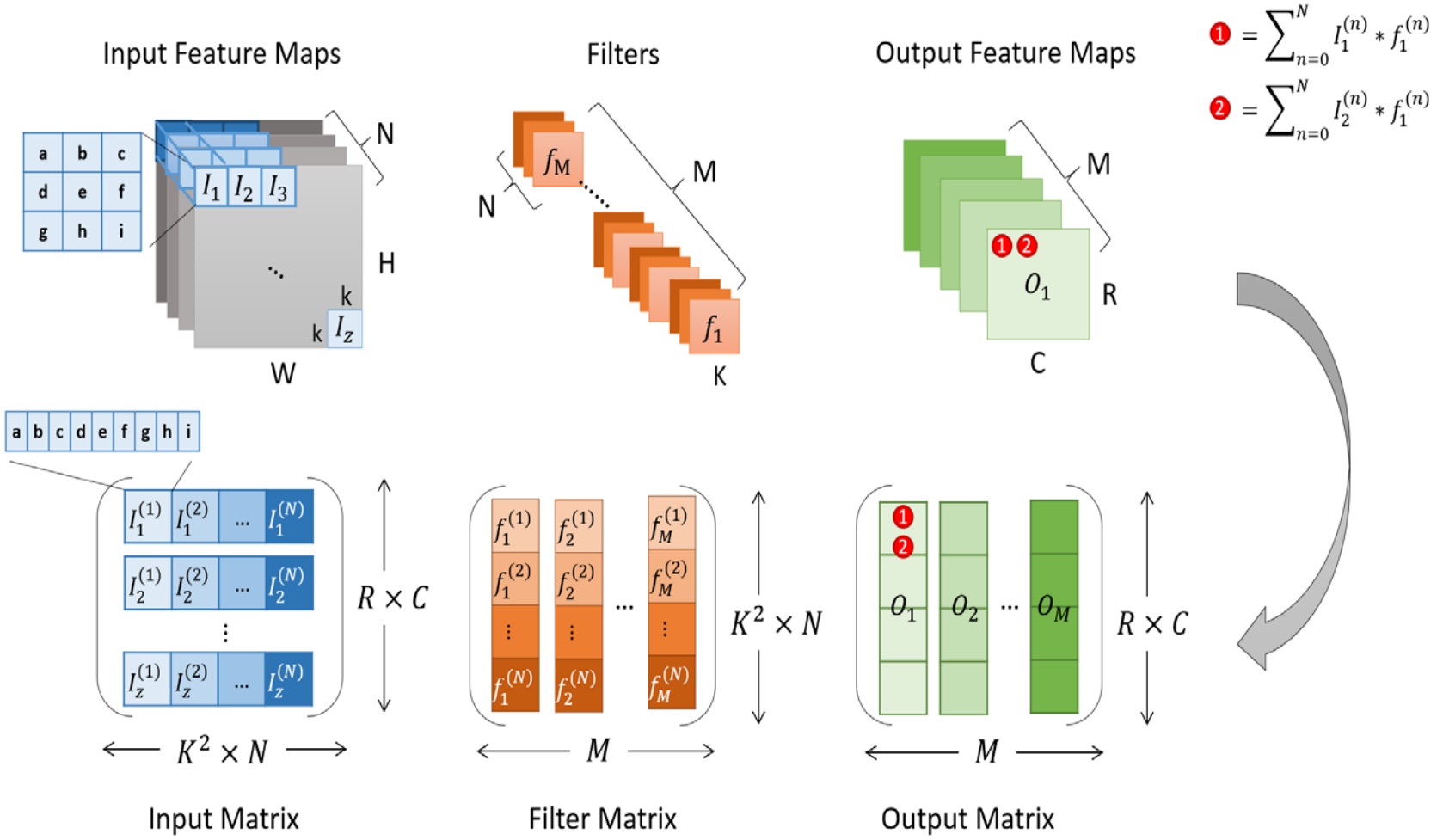


Figure by 劉珈寧

TPU Software

- Application runs on TPU is written in TensorFlow which is compiled into an API that can run on TPUs.
- TPU stack is split into a User space driver and a Kernel Driver.
  - Kernel driver: lightweight, handles only memory management and interrupts, designed for long-term stability
  - User space driver changes frequently. It sets up and controls TPU execution, reformats data into TPU order, and translates API calls into TPU instructions and turns them into application binary.
- Optimization of operations
  - User space driver compiles a model, caches the program image, writes the weight image into the TPU Weight Memory.
  - TPU computation is done often one layer at a time, with overlapped execution allowing the MAC unit to hide most noncritical path operations.

Wrap Up TPU

- Use dedicated memories
  - 24 MiB dedicated buffer (1/3 of the die area), 4 MiB accumulator buffers
- Invest resources in arithmetic units and dedicated memories
  - 60% of the memory and 250X the arithmetic units of a server-class CPU, with only half the CPU size and power.
- Use the easiest form of parallelism that matches the domain
  - Exploits 2D SIMD parallelism
- Reduce the data size and type needed for the domain
  - Primarily uses 8-bit integers
- Use a domain-specific programming language
  - Uses TensorFlow

Machine Learning

Hello!! R2-D2

Thanks You