Tensor Processor Unit (TPU)
Deep Neural Networks
DNN Workloads

- Convolutions, matrix-matrix (M-M), matrix-vector (M-V), application of non-linearities, calculation of loss functions, weight updates …

- Convolution is the most expensive one & can be converted to matrix multiplication

- The idea: Accelerate matrix multiplication
Matrix Multiplication

\[
\begin{pmatrix}
a_{11} & a_{12} & a_{13} & a_{14} \\
a_{21} & a_{22} & a_{23} & a_{24} \\
a_{31} & a_{32} & a_{33} & a_{34} \\
a_{41} & a_{42} & a_{43} & a_{44}
\end{pmatrix}
\times
\begin{pmatrix}
b_{11} & b_{12} & b_{13} & b_{14} \\
b_{21} & b_{22} & b_{23} & b_{24} \\
b_{31} & b_{32} & b_{33} & b_{34} \\
b_{41} & b_{42} & b_{43} & b_{44}
\end{pmatrix}
= 
\begin{pmatrix}
c_{11}
\end{pmatrix}
\]

for (int i=1; i<=4; i++)
  for (int j=1; j<=4; j++) {
    sum = 0;
    for (int k=1; k<=4; k++)
      sum = sum + a[i][k] * b[k][j];
    c[i][j] = sum;
  }
Matrix Multiplication

\[
\begin{bmatrix}
  a_{11} & a_{12} & a_{13} & a_{14} \\
  a_{21} & a_{22} & a_{23} & a_{24} \\
  a_{31} & a_{32} & a_{33} & a_{34} \\
  a_{41} & a_{42} & a_{43} & a_{44}
\end{bmatrix}
\times
\begin{bmatrix}
  b_{11} & b_{12} & b_{13} & b_{14} \\
  b_{21} & b_{22} & b_{23} & b_{24} \\
  b_{31} & b_{32} & b_{33} & b_{34} \\
  b_{41} & b_{42} & b_{43} & b_{44}
\end{bmatrix}
=
\begin{bmatrix}
  c_{11} & c_{12}
\end{bmatrix}
\]

```java
for (int i=1; i<=4; i++)
    for (int j=1; j<=4; j++)
    {
        sum = 0;
        for (int k=1; k<=4; k++)
            sum = sum + a[i][k] * b[k][j];
        c[i][j] = sum;
    }
```
Matrix Multiplication

\[
\begin{bmatrix}
a_{11} & a_{12} & a_{13} & a_{14} \\
a_{21} & a_{22} & a_{23} & a_{24} \\
a_{31} & a_{32} & a_{33} & a_{34} \\
a_{41} & a_{42} & a_{43} & a_{44}
\end{bmatrix}
\times
\begin{bmatrix}
b_{11} & b_{12} & b_{13} & b_{14} \\
b_{21} & b_{22} & b_{23} & b_{24} \\
b_{31} & b_{32} & b_{33} & b_{34} \\
b_{41} & b_{42} & b_{43} & b_{44}
\end{bmatrix}
= 
\begin{bmatrix}
c_{11} & c_{12} & c_{13} & c_{14} \\
c_{21}
\end{bmatrix}
\]

for (int i=1; i<=4; i++)
    for (int j=1; j<=4; j++)
        { 
            sum = 0;
            for (int k=1; k<=4; k++)
                sum = sum + a[i][k] * b[k][j];
            c[i][j] = sum;
        }
Matrix Multiplication

\[
\begin{bmatrix}
a_{11} & a_{12} & a_{13} & a_{14} \\
a_{21} & a_{22} & a_{23} & a_{24} \\
a_{31} & a_{32} & a_{33} & a_{34} \\
a_{41} & a_{42} & a_{43} & a_{44}
\end{bmatrix}
\times
\begin{bmatrix}
b_{11} & b_{12} & b_{13} & b_{14} \\
b_{21} & b_{22} & b_{23} & b_{24} \\
b_{31} & b_{32} & b_{33} & b_{34} \\
b_{41} & b_{42} & b_{43} & b_{44}
\end{bmatrix}
= 
\begin{bmatrix}
c_{11} & c_{12} & c_{13} & c_{14} \\
c_{21} & c_{22}
\end{bmatrix}
\]

```java
for (int i=1; i<=4; i++)
    for (int j=1; j<=4; j++) {
        sum = 0;
        for (int k=1; k<=4; k++)
            sum = sum + a[i][k] * b[k][j];
        c[i][j] = sum;
    }
```
Matrix Multiplication

\[
\begin{bmatrix}
a_{11} & a_{12} & a_{13} & a_{14} \\
a_{21} & a_{22} & a_{23} & a_{24} \\
a_{31} & a_{32} & a_{33} & a_{34} \\
a_{41} & a_{42} & a_{43} & a_{44} \\
\end{bmatrix}
\begin{bmatrix}
b_{11} & b_{12} & b_{13} & b_{14} \\
b_{21} & b_{22} & b_{23} & b_{24} \\
b_{31} & b_{32} & b_{33} & b_{34} \\
b_{41} & b_{42} & b_{43} & b_{44} \\
\end{bmatrix}
= 
\begin{bmatrix}
c_{11} & c_{12} & c_{13} & c_{14} \\
c_{21} & c_{22} & c_{23} & c_{24} \\
c_{31} & c_{32} & c_{33} & c_{34} \\
c_{41} & c_{42} & c_{43} & c_{44} \\
\end{bmatrix}
\]

for (int i=1; i<=4; i++)
for (int j=1; j<=4; j++) {
    sum = 0;
    for (int k=1; k<=4; k++)
        sum = sum + a[i][k] * b[k][j];
    c[i][j] = sum;
}
Systolic Arrays

\[
\begin{align*}
a_{13} & \quad a_{12} & \quad a_{11} \\
a_{22} & \quad a_{21} \\
a_{31} & \quad & \quad \\
\end{align*}
\]

\[
\begin{align*}
w_{11} & \quad & \quad w_{21} & \quad & \quad w_{31} \\
w_{12} & \quad & \quad w_{22} & \quad & \quad w_{32} \\
w_{13} & \quad & \quad w_{23} & \quad & \quad w_{33} \\
\end{align*}
\]

\[t = 0\]
Systolic Arrays

\[ t = 1 \]

\[ [W] \times [A] = [X] \]
Systolic Arrays

\[
\begin{align*}
[b_{12}] & \quad [b_{11}] & \quad [a_{13}] & \quad [a_{12}w_{11}] & \quad [a_{11}w_{21}] & \quad [w_{31}] \\
[b_{21}] & \quad [a_{23}] & \quad [a_{22}] & \quad [a_{21}w_{12} + a_{11}w_{11}] & \quad [w_{22}] & \quad [w_{32}] \\
[a_{33}] & \quad [a_{32}] & \quad [a_{31}] & \quad [w_{13}] & \quad [w_{23}] & \quad [w_{33}] \\
\end{align*}
\]

\[t = 2\]

\[[W] \times [A] = [X]\]
Systolic Arrays

\[
[W] \times [A] = [X]
\]

t = 3
Systolic Arrays

\[ [W] \times [A] = [X] \]
\[ [W] \times [B] = [Y] \]
Systolic Arrays

\[ \begin{align*}
&W \ast [A] = [X] \\
&W \ast [B] = [Y]
\end{align*} \]
Systolic Arrays

\[
\begin{align*}
&W \times & &A = & &X \\
&W \times & &B = & &Y \\
\end{align*}
\]
Systolic Arrays

\[
\begin{align*}
[W] \times [A] &= [X] \\
[W] \times [B] &= [Y] \\
[W] \times [C] &= [Z] \\
&\quad \ldots
\end{align*}
\]
Tensor Processing Unit (TPU)
First Generation
Tensor Processing Unit (TPU) - Goals

- Custom ASIC developed by Google for Neural Networks Acceleration
- Improve cost-performance over GPUs
- Run whole inference models in the TPU
- Flexible to meet NNs needs of 2015 and beyond
Tensor Processing Unit (TPU) - Design Choices

- Coprocessor on the PCIe I/O bus
- CPU sends TPU instructions for it to execute
- Closer in spirit to an FPU coprocessor than to a GPU
- Quantization - use simple 8-bit integer instead of floats reducing power consumption
TPU v1 Architecture
- TPU instructions are sent from the host over the PCIe into an instruction buffer
- CISC Instructions with average CPI 10 to 20
Weight Memory & Weight FIFO

- Off-chip 8 GiB DRAM
- Four tiles deep on chip weight FIFO
- Same weights reused across multiple inputs
Matrix Multiply Unit

- 256x256 MACs
- 2 tiles of 64KiB weights
  - One active, the other used for double buffering
- Must keep the MMU busy
  - 4 stage pipeline for CISC instructions
- Implemented as a Systolic Array
- Designed for dense matrices
Accumulators

- Collect the 16-bit products of the MMU
  - 4MiB: 4096, 256-element, 32-bit
    - Why 4K vectors?
      - Peak performance observed at 1350
      - Round up to 2048
      - Double for double buffering
  - Calculate partial sums
Unified Buffer

- 24 MiB on-chip for intermediate results
- Inputs to the MMU
TPU Memory Management

CPU implicit memory management

TPU explicit memory management
Designing Next Generations of TPU
Problem : Training

- Computation
- Memory
- Programmability
- Quantization
Solution: Vector Units & Vector Memory

- Read-only weights → Write (Computation)
- 2D 128x8 vector registers (Storage)
- Replaces the fixed-function datapath (Programmability)
- Bfloat16 instead of int8 (No quantization)
Problem: Memory Bandwidth

- TPUv1 was memory bound
- TPUv2 required even more memory for training
Solution: High Bandwidth Memory (HBM)

- HMB DRAM (20x bw than v1)
- 4 short stacks of DRAM chips
Problem: ML on a Supercomputer Scale

- Chip-to-Chip Interconnect fabric
- 256 chips in 2D-Torus
Other Features of TPUv2/3

- Instructions no longer delivered from the CPU
- 322-bit VLIW instructions
- Scalar memory
TPU v2 Board

HBM

Scalar Unit

Vector Unit

MXU

Scalar Unit

Vector Unit

MXU

HBM

TensorCore

TensorCore
HBM capacity/bandwidth: 32GiB, 900 GB/s
Measured min/mean/max power: 123/220/262 W
Peak compute per chip: 123 teraflops
Peak compute per pod (1024 chips): 126 petaflops
Bisection bandwidth per pod (1024 chips): 6.4 TB/s
TPU v4

Added CMEM (Common Memory)
- Small (128 MB) scratchpad SRAM
- Reduces the number to the slowest and least energy efficient memory

Added OCI (On-Chip Interconnect)
- Connects all components of the chip together
- Allows to better control which memory is used, how data is transferred

4D Tensor DMA (Dynamic Memory Access)
- The chip contains tensor DMA engines that decode and execute DMA instructions
- Offloads work from the main TensorCore
TPU v4

- Improved Interconnect Topology
- Now able to connect up to 4096 chips in a 3D Torus
TPU v4 Board

HBM capacity/bandwidth: 32GiB, 1200 GB/s (33% ↑)
Measured min/mean/max power: 90/170/192 W (~25% ↓)
Peak compute per chip: 275 teraflops (~2 times ↑)
Peak compute per pod (4096 chips): 1.1 exaflops (~10 times ↑)
Bisection bandwidth per pod (4096 chips): 24 TB/s (~4 times ↑)
Fig. 1. Computation of a CNN layer.
Fig. 2. Eyeriss system architecture.
Fig. 4. Dataflow in a PE set for processing a 2-D convolution. (a) Rows of filter weights are reused across PEs horizontally. (b) Rows of ifmap values are reused across PEs diagonally. (c) Rows of psums are accumulated across PEs vertically. Reuse and accumulation of data within a PE set reduce accesses to the GLB and DRAM, saving data movement energy cost. In this example, the number of filter rows ($R$), ifmap rows ($H$), and ofmap rows ($E$) are 3, 5, and 3, respectively. Therefore, the PE set size is $3 \times 3$. Filter and ifmap values from different rows are sent to the PE set in a time-interleaved fashion; all the PEs that reuse the same value receive it at the same cycle. The psums generated from one PE are sent to its neighbor PE immediately.
Fig. 12. PE architecture. The datapaths in red show the data gating logic to skip the processing of zero ifmap data.
References

Google LLC., https://cloud.google.com/tpu/docs/system-architecture-tpu-vm