Deep Learning Optimization HW Accelerator

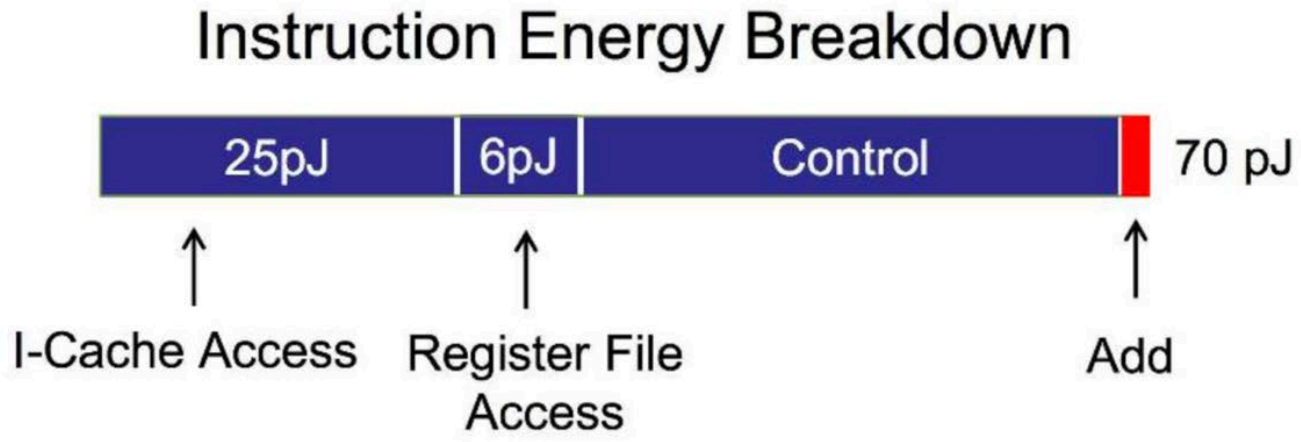
May 22, 2023

Eunhyeok Park

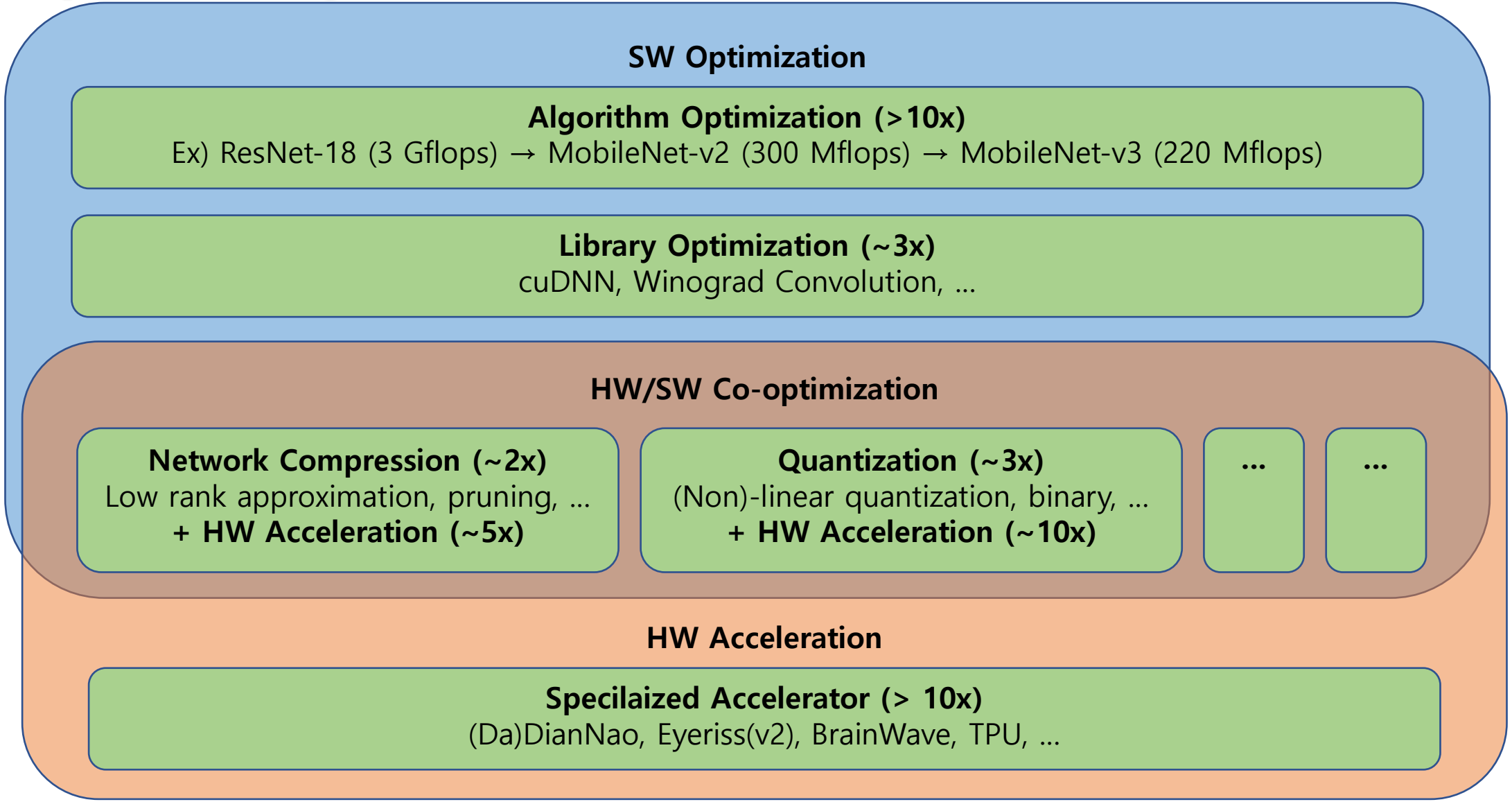
CPU Consumes Power for the Other Jobs than Computation!

From Mark Horowitz's ISSCC 2014 Keynote: "Computing's Energy Problem (and what we can do about it)".

Energy for control logic, SRAM, and register accesses needed by matrix multiply dominates in conventional processors.



Optimizing Neural Networks

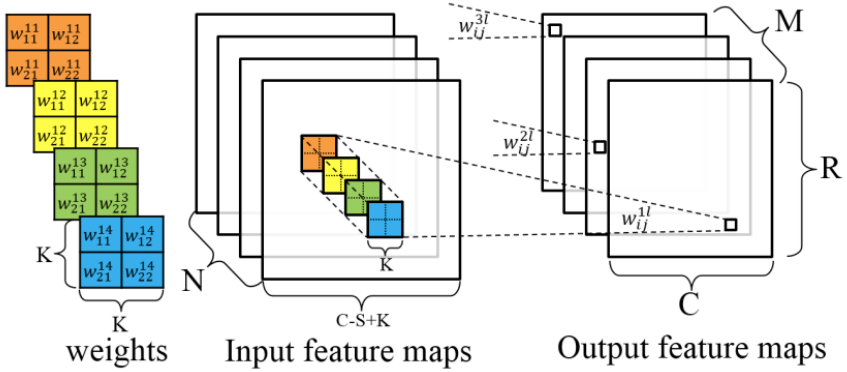


CNN Roofline Optimization

Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks

CNN Computation

- Convolutional operation performs multiple nested loop operations to achieve a single output
  - The convolutional layer receives N feature maps as input.
  - Each input feature map is convolved by a shifting window with a K×K kernel to generate one pixel in one output feature map.
  - The stride of the shifting window is S, which is normally smaller than K.
  - A total of M output feature maps will form the set of input feature maps for the next convolutional layer.

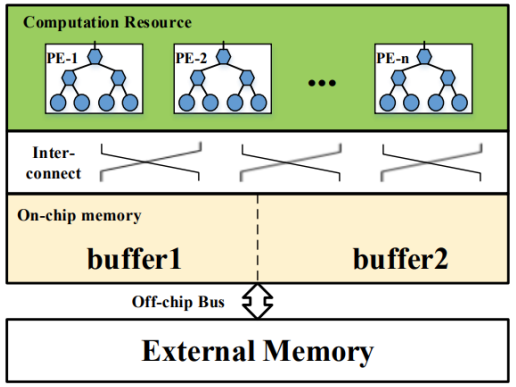


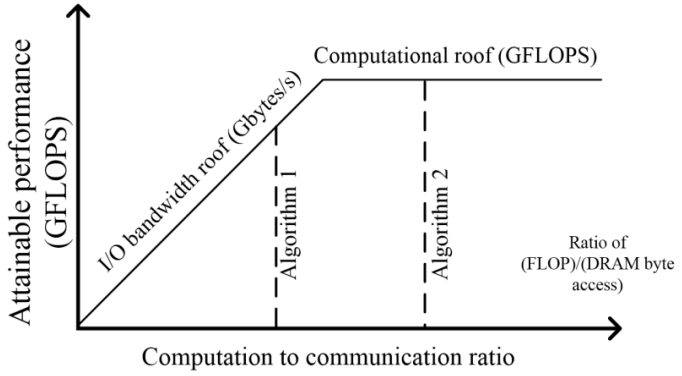

```
for(row=0; row<R; row++) {
 for(col=0; col<C; col++) {
  for(to=0; to<M; to++) {
   for(ti=0; ti<N; ti++) {
    for(i=0; i<K; i++) {
     for(j=0; j<K; j++) {
   L:  output_fm[to][row][col] +=
          weights[to][ti][i][j]*
          input_fm[ti][S*row+i][S*col+j];
} } } } } }
```

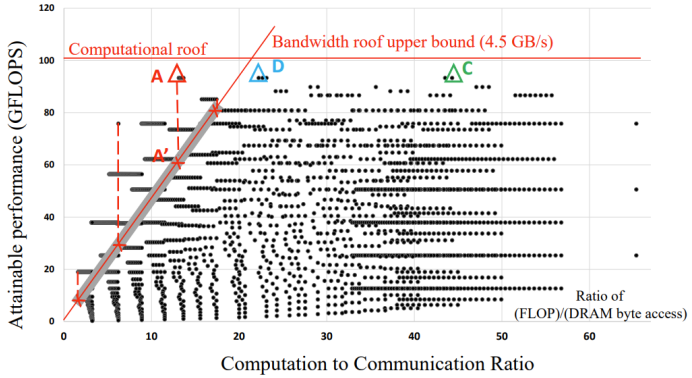
Roofline Optimization for CNN

- An implementation can be either computation-bounded or memory-bounded
  - # of computation unit and on-chip/off-chip bandwidth is restricted
  - Within the same resource budget, attainable performance can be highly different depending on the hardware configuration / optimization

$$Attainable\ Perf. = min \begin{cases} Computational\ Roof \\ CTC\ Ratio \times BW \end{cases}$$







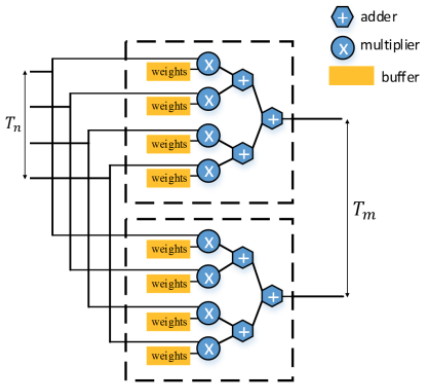
Design Challenges of CNN Accelerator

- Loop tiling is mandatory to fit a small portion of data on-chip
- The organization of PEs, memory shaping, and interconnects should be carefully considered
- The data processing throughput of PEs should match the off-chip bandwidth provided by the FPGA platform.

```
for(row=0; row<R; row+=Tr) {
 for(col=0; col<C; col+=Tc) {
  for(to=0; to<M; to+=Tm) {
   for(ti=0; ti<N; ti+=Tn) {
    //load output feature maps
    //load weights
    //load input feature maps
                                        External data transfer

                                        On-chip data computation
    for(trr=row;trr<min(row+Tr,R);trr++){
     for(tcc=col;tcc<min(col+Tc,C);tcc++){
      for(too=to;too<min(to+Tm,M);too++){
       for(tii=ti;tii<min(ti+Tn,N);tii++){
        for(i=0; i<K; i++) {
         for(j=0; j<K; j++) {
        L: output_fm[too][trr][tcc] +=
             weights[too][tii][i][j]*
             input_fm[tii][S*trr+i][S*tcc+j];
    } } } } } }
    //store output feature maps
} } } }
```

Loop Optimization



- Basic optimization: loop unrolling & pipelining
  - Unrolling along different loop dimensions will generate different implementation variants
  - The data sharing relations between different loop iterations of a loop dimension should be considered

```
//on-chip data computation
for(trr=row; trr<min(row+Tr,R); trr++){
 for(tcc=col; tcc<min(col+Tc,C); tcc++){
  for(too=to; too<min(to+Tm,M); too++){
   for(tii=ti; tii<min(ti+Tn,N); tii++){
    for(i=0; i<K; i++) {
     for(j=0; j<K; j++) {
    L:  output_fm[too][trr][tcc] +=
          weights[too][tii][i][j]*
          input_fm[tii][S*trr+i][S*tcc+j];
} } } } } }
```

```
//on-chip data computation
for(i=0; i<K; i++) {
 for(j=0; j<K; j++) {
  for(trr=row; trr<min(row+Tr,R); trr++){
   for(tcc=col; tcc<min(col+Tc,C); tcc++){
#pragma HLS pipeline
    for(too=to; too<min(to+Tm,M); too++){
#pragma HLS UNROLL
     for(tii=ti; tii<min(ti+Tn,N); tii++){
#pragma HLS UNROLL
    L:  output_fm[too][trr][tcc] +=
          weights[too][tii][i][j]*
          input_fm[tii][S*trr+i][S*tcc+j];
} } } } } }
```

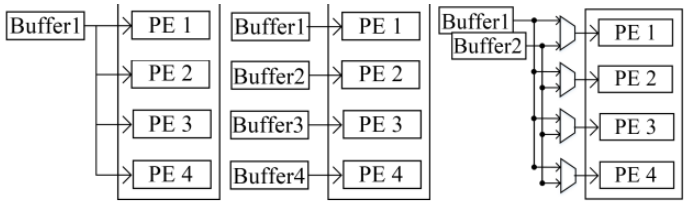
Data Dependency Analysis

- Irrelevant
  - If a loop iterator i_k does not appear in any access functions of an array A, the corresponding loop dimension is irrelevant to array A
- Independent
  - If the union of data space accessed on an array A is totally separable along a certain loop dimension i_k or is disjoint with any two distinct parameters p_1 and p_2, the data accessed by the loop dimension i_k is independent of array
- Dependent
  - If the union of data space accessed on an array A is not separable along a certain loop dimension i_k, the loop dimension i_k is dependent of array

Data Dependency Analysis - 2

- After analyzing the data dependency of loop variables and loop iterations, we should modify the array shape accordingly

```
for(row=0; row<R; row+=Tr) {                          External data transfer
 for(col=0; col<C; col+=Tc) {
  for(to=0; to<M; to+=Tm) {
   for(ti=0; ti<N; ti+=Tn) {
   //load output feature maps
   //load weights
   //load input feature maps
                                                      On-chip data computation
   for(trr=row;trr<min(row+Tr,R);trr++){
    for(tcc=col;tcc<min(col+Tc,C);tcc++){
     for(too=to;too<min(to+Tm,M);too++){
      for(tii=ti;tii<min(ti+Tn,N);tii++){
       for(i=0; i<K; i++) {
        for(j=0; j<K; j++) {
        L: output_fm[too][trr][tcc] +=
             weights[too][tii][i][j]*
             input_fm[tii][S*trr+i][S*tcc+j];
   //store output feature maps
} } } }
```



(a) irrelevant (b) independent (c) dependent

	input_fm	*weights*	*output_fm*
trr	dependent	irrelevant	independent
tcc	dependent	irrelevant	independent
too	irrelevant	independent	independent
tii	independent	independent	irrelevant
i	dependent	independent	irrelevant
j	dependent	independent	irrelevant

```
//on-chip data computation
for(i=0; i<K; i++) {
 for(j=0; j<K; j++) {
  for(trr=row; trr<min(row+Tr,R); trr++){
   for(tcc=col; tcc<min(col+Tc,C); tcc++){
#pragma HLS pipeline
    for(too=to; too<min(to+Tm,M); too++){
#pragma HLS UNROLL
     for(tii=ti; tii<min(ti+Tn,N); tii++){
#pragma HLS UNROLL
    L: output_fm[too][trr][tcc] +=
         weights[too][tii][i][j]*
         input_fm[tii][S*trr+i][S*tcc+j];
} } } } } }
```

Computation Optimization

- The range of tile size is determined by available resource and network structure

$$\begin{cases} 0 < Tm \times Tn \leq (\#\ of\ PEs) \\ 0 < Tm \leq M \\ 0 < Tn \leq N \\ 0 < Tr \leq R \\ 0 < Tc \leq C \end{cases}$$

- The computation loop can be computed analytically based on the tile size

$$\begin{aligned} & computational\ roof \\ = & \frac{total\ number\ of\ operations}{number\ of\ execution\ cycles} \\ = & \frac{2 \times R \times C \times M \times N \times K \times K}{\lceil \frac{M}{T_m} \rceil \times \lceil \frac{N}{T_n} \rceil \times \frac{R}{T_r} \times \frac{C}{T_c} \times (T_r \times T_c \times K \times K + P)} \\ \approx & \frac{2 \times R \times C \times M \times N \times K \times K}{\lceil \frac{M}{T_m} \rceil \times \lceil \frac{N}{T_n} \rceil \times R \times C \times K \times K} \end{aligned}$$

where $P = pipeline\ depth - 1$.

Memory I/O Optimization - 1

- Local Memory Promotion
  - The innermost loop dimension t_i is irrelevant to array $output\_fm$
  - The accesses to array $output\_fm$ can be promoted to outer loops.
  - The trip count of memory access operations on array $output\_fm$ reduces from $2 \times \frac{M}{T_m} \times \frac{N}{T_n} \times \frac{R}{T_r} \times \frac{C}{T_c}$ to $\frac{M}{T_m} \times \frac{R}{T_r} \times \frac{C}{T_c}$
  - Likewise, other data buffers can be promoted to minimize data I/O

```
for(row=0; row<R; row+=Tr) {
 for(col=0; col<C; col+=Tc) {
  for(to=0; to<M; to+=Tm) {
   for(ti=0; ti<N; ti+=Tn) {
   //load output feature maps
   //load weights
   //load input feature maps

        L: foo(output_fm(to,row,col),
               weights(to,ti),
               input_fm(ti,row,col));
   //store output feature maps
   }

  } } }
```

	irrelevant dimension(s)
$input\_fm$	to
$weights$	row,col
$output\_fm$	ti

Memory I/O Optimization - 2

- Computation to communication (CTC) ratio also can be computed analytically
  - Required buffer size

$$B_{in} = T_n(ST_r + K - S)(ST_c + K - S)$$
$$B_{wght} = T_m T_n K^2$$
$$B_{out} = T_m T_r T_c$$
$$0 < B_{in} + B_{wght} + B_{out} \leq BRAM_{capacity}$$

  - Trip count of each buffer

$$\alpha_{in} = \alpha_{wght} = \frac{M}{T_m} \times \frac{N}{T_n} \times \frac{R}{T_r} \times \frac{C}{T_c}$$

$$\alpha_{out} = 2 \times \frac{M}{T_m} \times \frac{N}{T_n} \times \frac{R}{T_r} \times \frac{C}{T_c} \quad \text{or} \quad \alpha_{out} = \frac{M}{T_m} \times \frac{R}{T_r} \times \frac{C}{T_c} \quad \text{(w/ data reuse)}$$

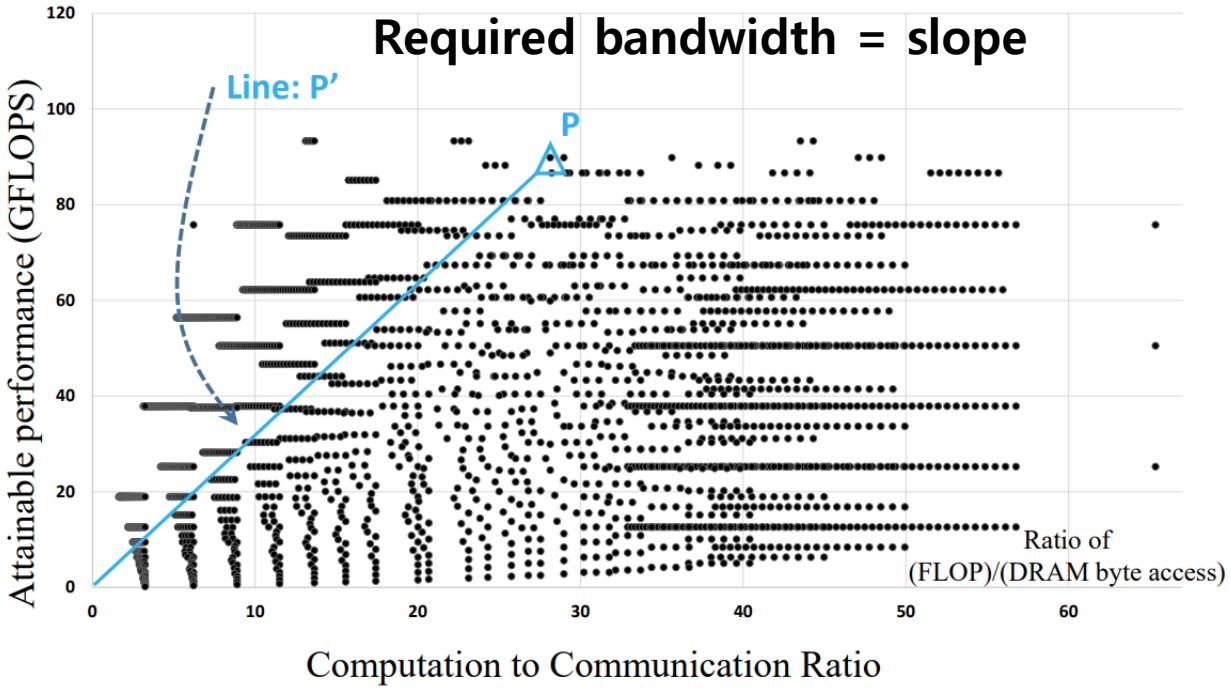
Memory I/O Optimization - 2

- Computation to communication (CTC) ratio also can be computed analytically
  - Computation to Communication Ratio

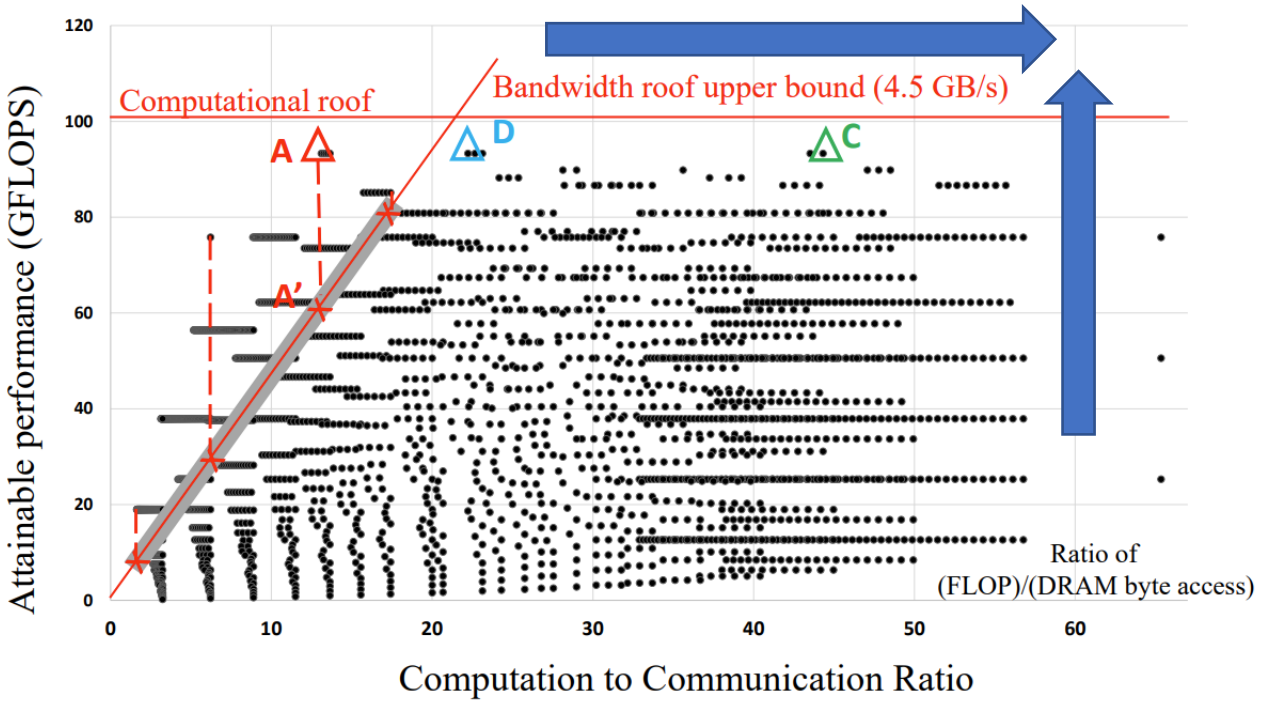
$$
\begin{aligned}
&Computation\ to\ Communication\ Ratio \\
&= \frac{total\ number\ of\ operations}{total\ amount\ of\ external\ data\ access} \\
&= \frac{2 \times R \times C \times M \times N \times K \times K}{\alpha_{in} \times B_{in} + \alpha_{wght} \times B_{wght} + \alpha_{out} \times B_{out}}
\end{aligned}
$$

Design Space Exploration

- Explore the possible design choice and find out the best configuration
  - Optimality could be differ depending on the layer structure
    - Flexibility should be considered to maximize the utilization of hardware

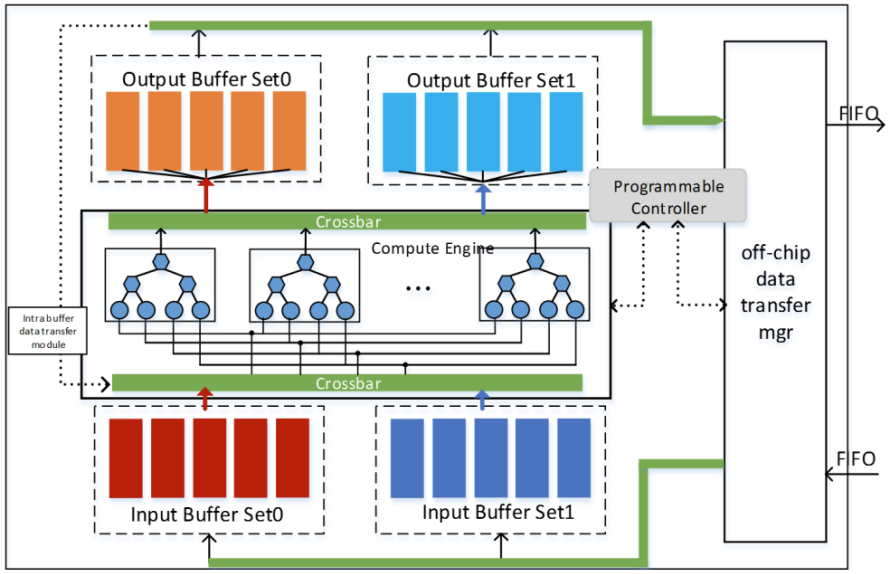


(a) Design space of all possible designs

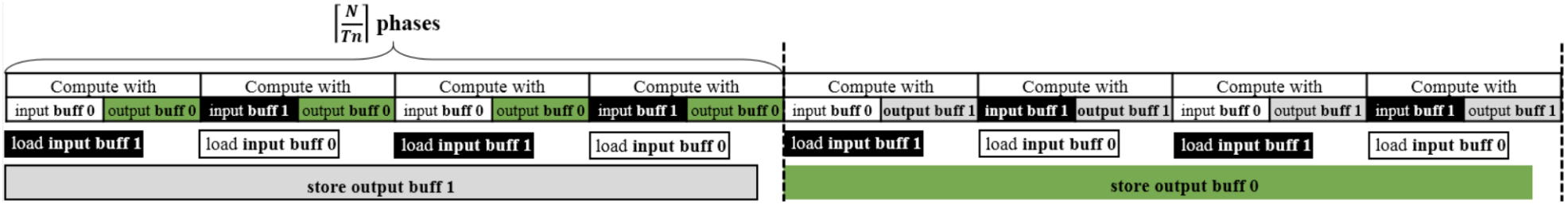


(b) Design space of platform-supported designs

Implementation Details



	Optimal Unroll Factor $\langle Tm, Tn \rangle$	Execution Cycles
Layer 1	$\langle 48, 3 \rangle$	366025
Layer 2	$\langle 20, 24 \rangle$	237185
Layer 3	$\langle 96, 5 \rangle$	160264
Layer 4	$\langle 95, 5 \rangle$	120198
Layer 5	$\langle 32, 15 \rangle$	80132
Total	-	963804
Cross-Layer Optimization	$\langle 64, 7 \rangle$	1008246

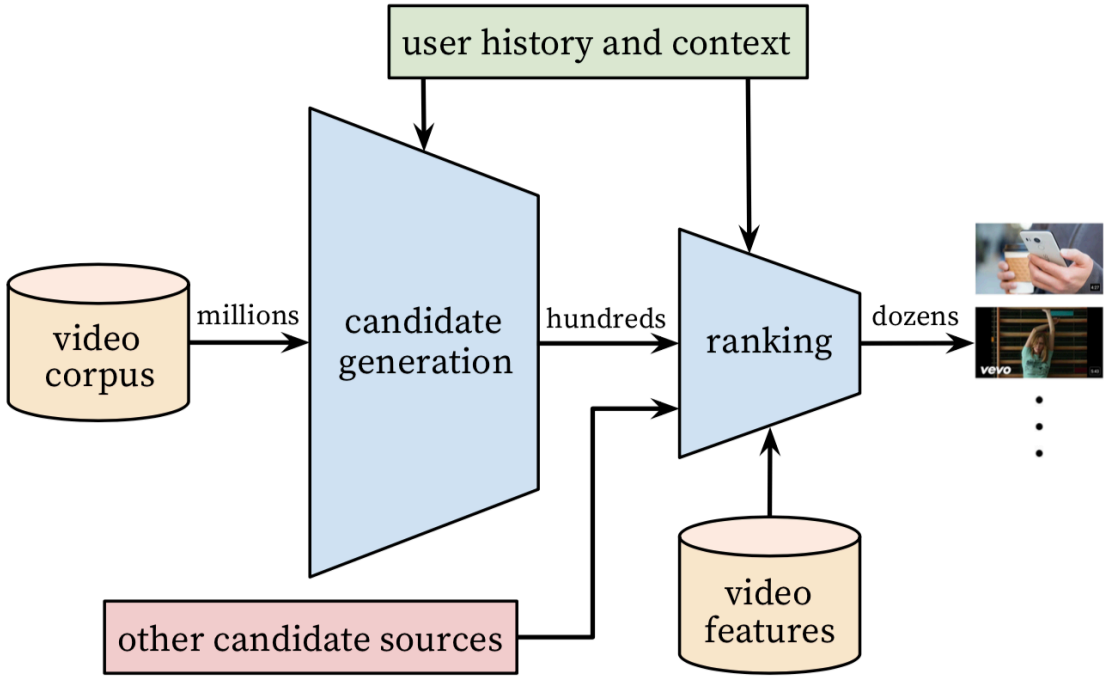


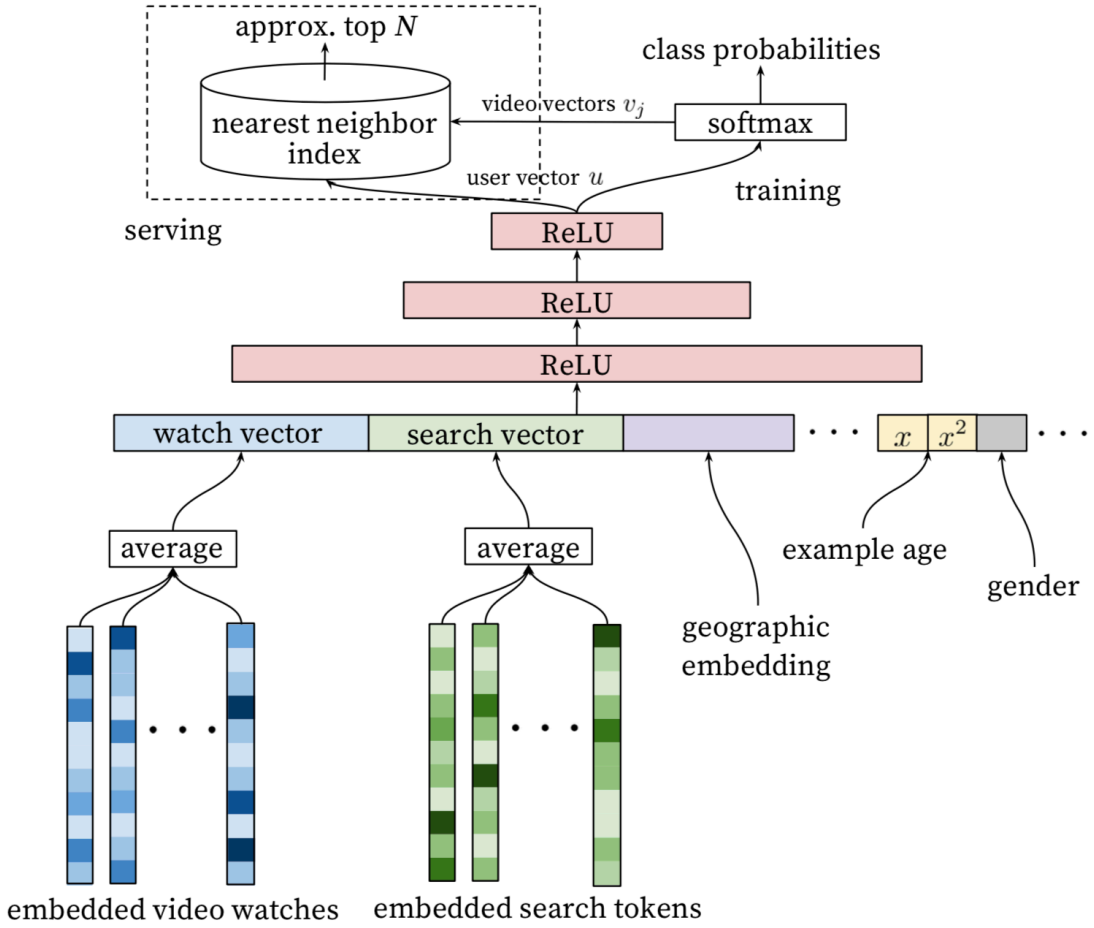
Google TPU

[Covington, 2016]

Recommendation System in Servers

- Neural collaborative filtering used in YouTube recommendation system

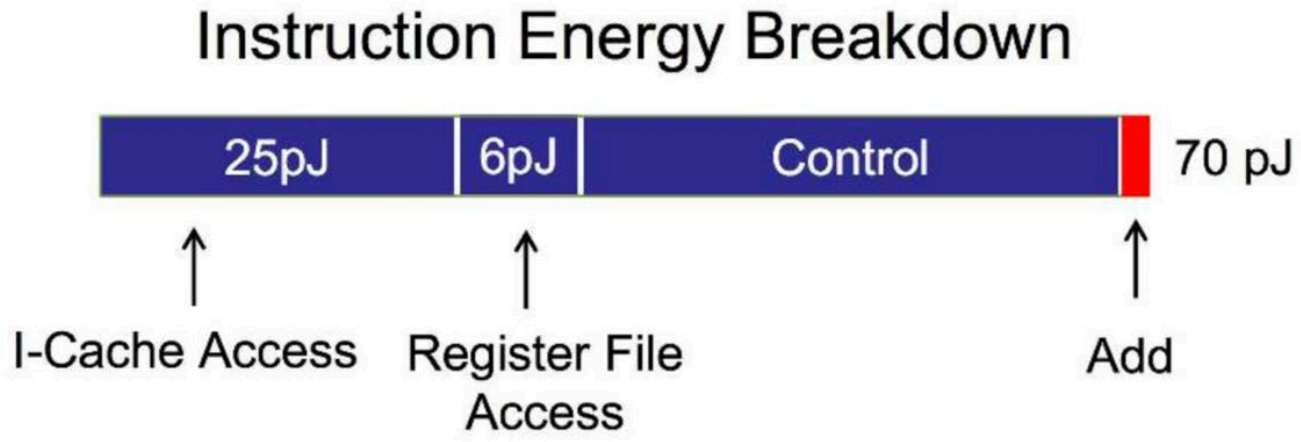




CPU Consumes Power for the Other Jobs than Computation!

From Mark Horowitz's ISSCC 2014 Keynote: "Computing's Energy Problem (and what we can do about it)".

Energy for control logic, SRAM, and register accesses needed by matrix multiply dominates in conventional processors.



[Google, 2017]

Google's Custom ASIC, Tensor Processing Unit (TPU)

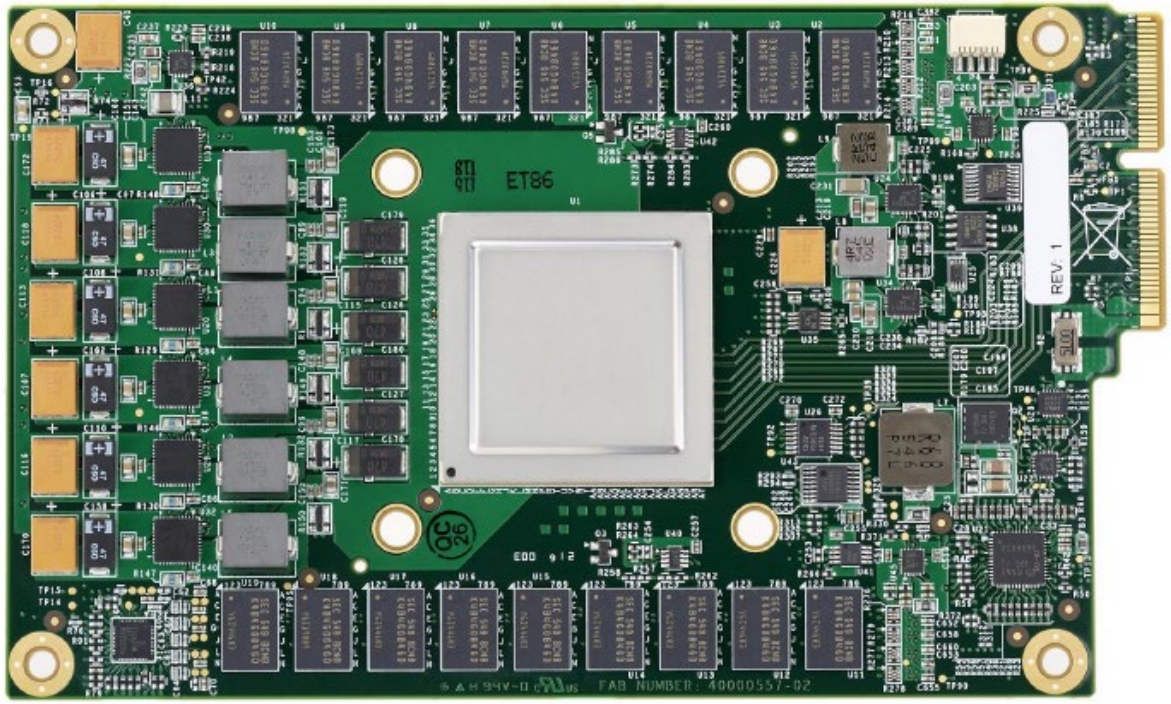


Figure 3. TPU Printed Circuit Board. It can be inserted in the slot for an SATA disk in a server, but the card uses PCIe Gen3 x16.

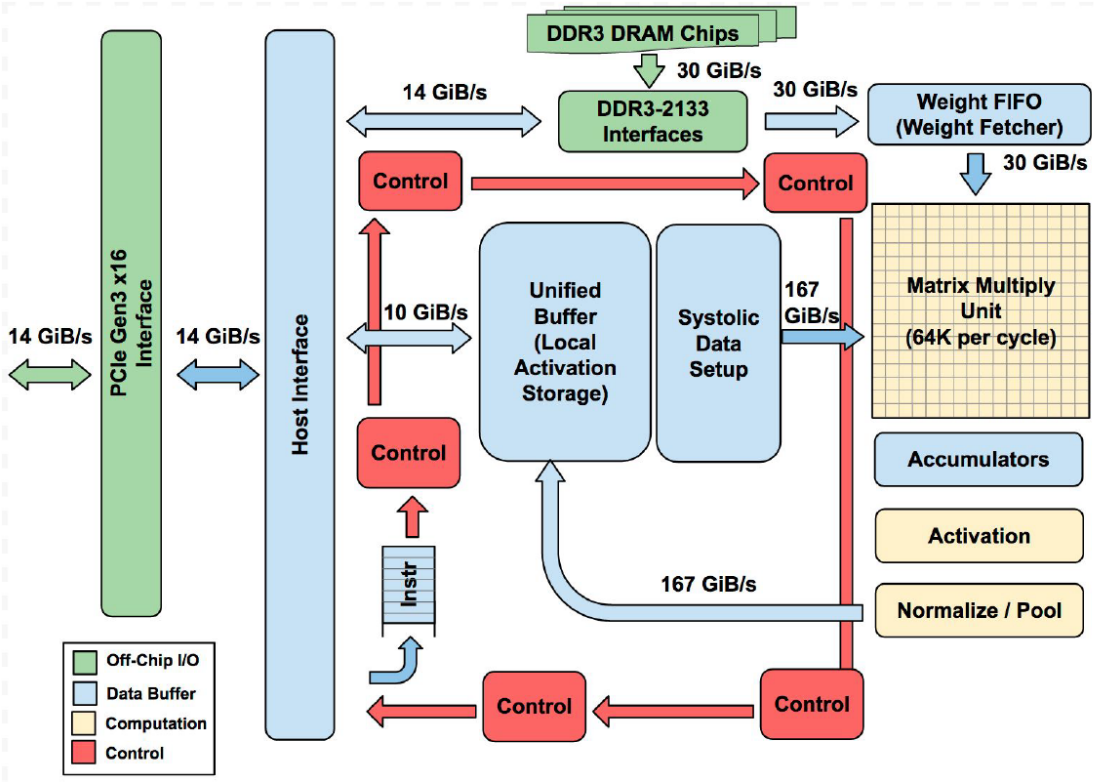
Google's Target Application

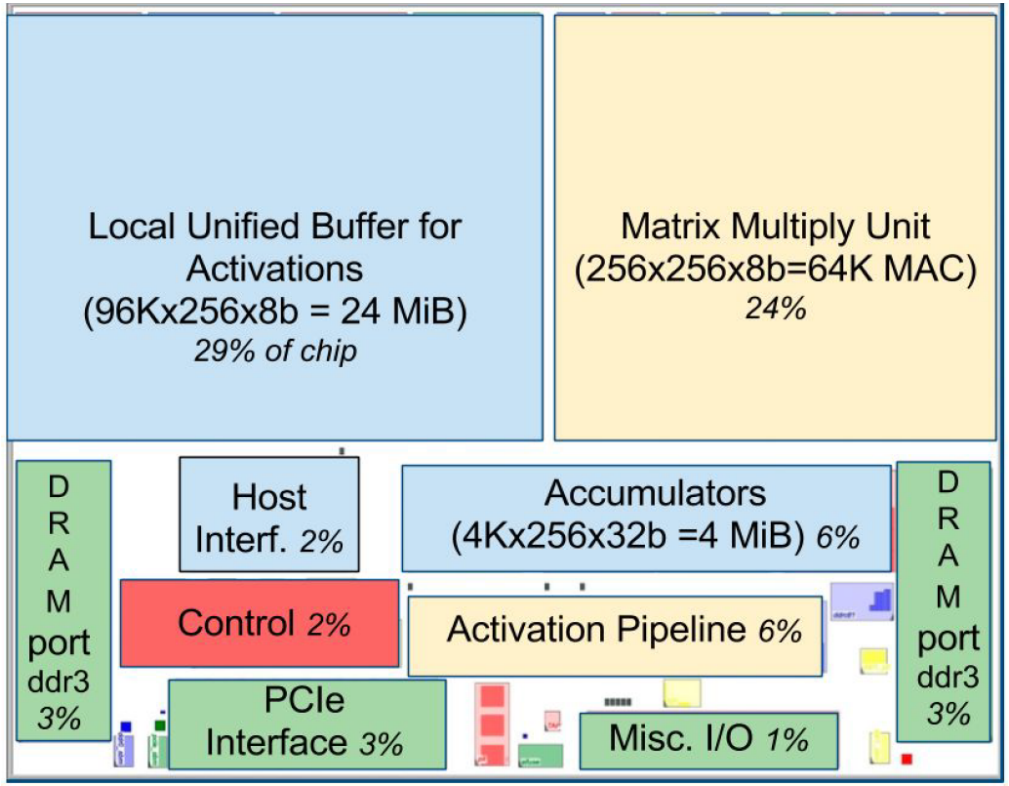
Name	LOC	Layers					Nonlinear function	Weights	TPU Ops / Weight Byte	TPU Batch Size	% of Deployed TPUs in 2016
		FC	Conv	Vector	Pool	Total					
MLP0	100	5				5	ReLU	20M	200	200	61%
MLP1	1,000	4				4	ReLU	5M	168	168	
LSTM0	1,000	24		34		58	sigmoid, tanh	52M	64	64	29%
LSTM1	1,500	37		19		56	sigmoid, tanh	34M	96	96	
CNN0	1,000		16			16	ReLU	8M	2,888	8	5%
CNN1	1,000	4	72		13	89	ReLU	100M	1,750	32	

- Representative tasks of google datacenter
  - Recommendation (RankBrain), translation(GNM Translate), image recognition (Inception), AlphaGo,…

- Large batches can mitigate the bandwidth problem
- Interestingly, CNN takes only 5 % of execution time…

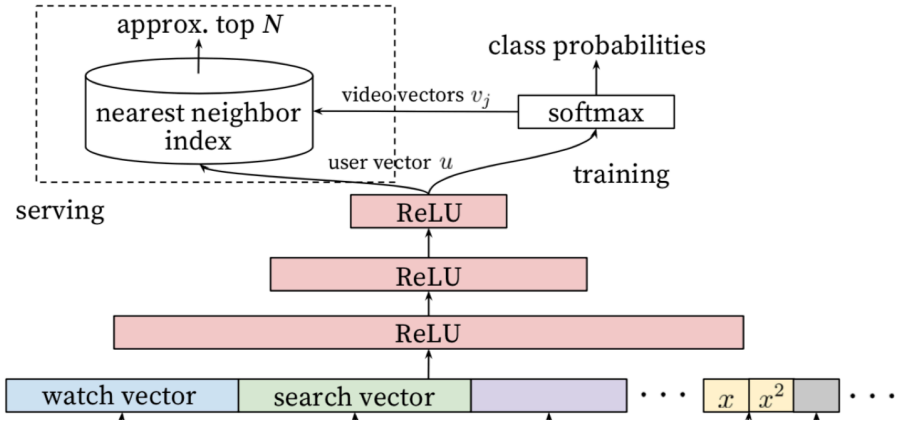
TPU Overview





Matrix Vector Multiplication in MLP

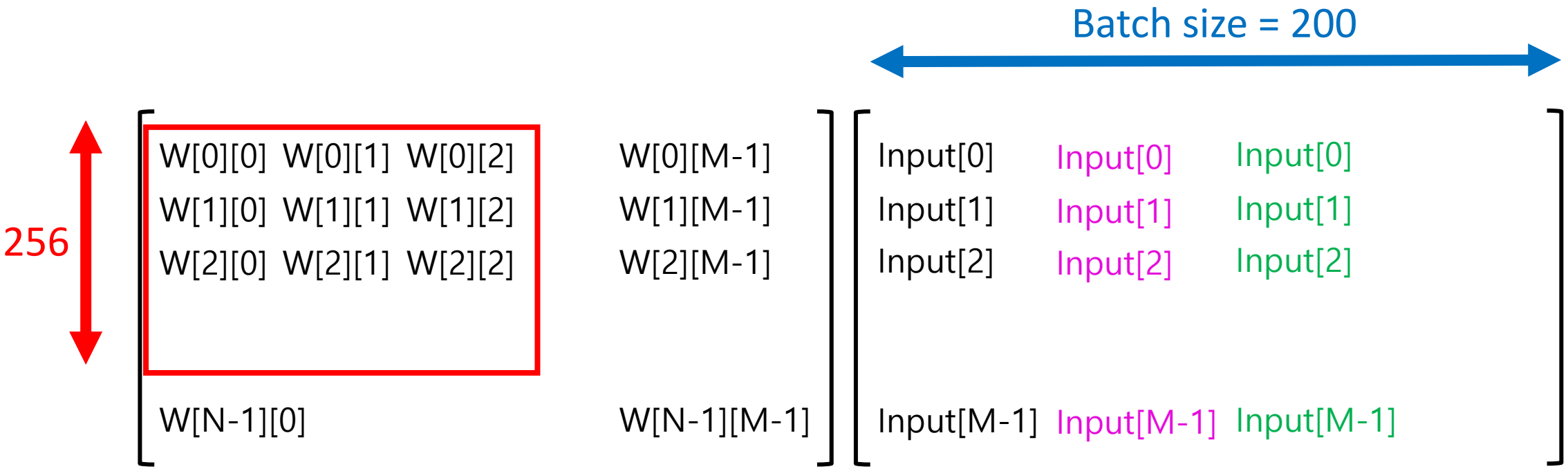
- In M*V multiplication, each weight is used only once
- Total size of weights is larger than cache size
- → Large amount of main memory access



$$
\begin{aligned}
&\text{Output[0]} = \Sigma_j \text{ Input[j]*W[0,j]} \\
&\text{Output[1]} = \Sigma_j \text{ Input[j]*W[1,j]} \\
&\text{Output[2]} = \Sigma_j \text{ Input[j]*W[2,j]} \\
&\\
&\\
&\text{Output[N-1]} = \Sigma_j \text{ Input[j]*W[N-1,j]}
\end{aligned}
\quad
\begin{bmatrix}
\text{W[0][0]} & \text{W[0][1]} & \text{W[0][2]} & \text{W[0][M-1]} \\
\text{W[1][0]} & \text{W[1][1]} & \text{W[1][2]} & \text{W[1][M-1]} \\
\text{W[2][0]} & \text{W[2][1]} & \text{W[2][2]} & \text{W[2][M-1]} \\
 & & & \\
 & & & \\
\text{W[N-1][0]} & & & \text{W[N-1][M-1]}
\end{bmatrix}
\begin{bmatrix}
\text{Input[0]} \\
\text{Input[1]} \\
\text{Input[2]} \\
 \\
 \\
\text{Input[M-1]}
\end{bmatrix}
$$

M*M Multiplication with Large Batch

- Weight reuse ~ batch size (# user requests processed together)
- In order to exploit 256x256 matrix array, we need to apply tiling
- For each tile, a new set of weights need to be fetched



[Google, 2017]

M*M in Systolic Array of TPU

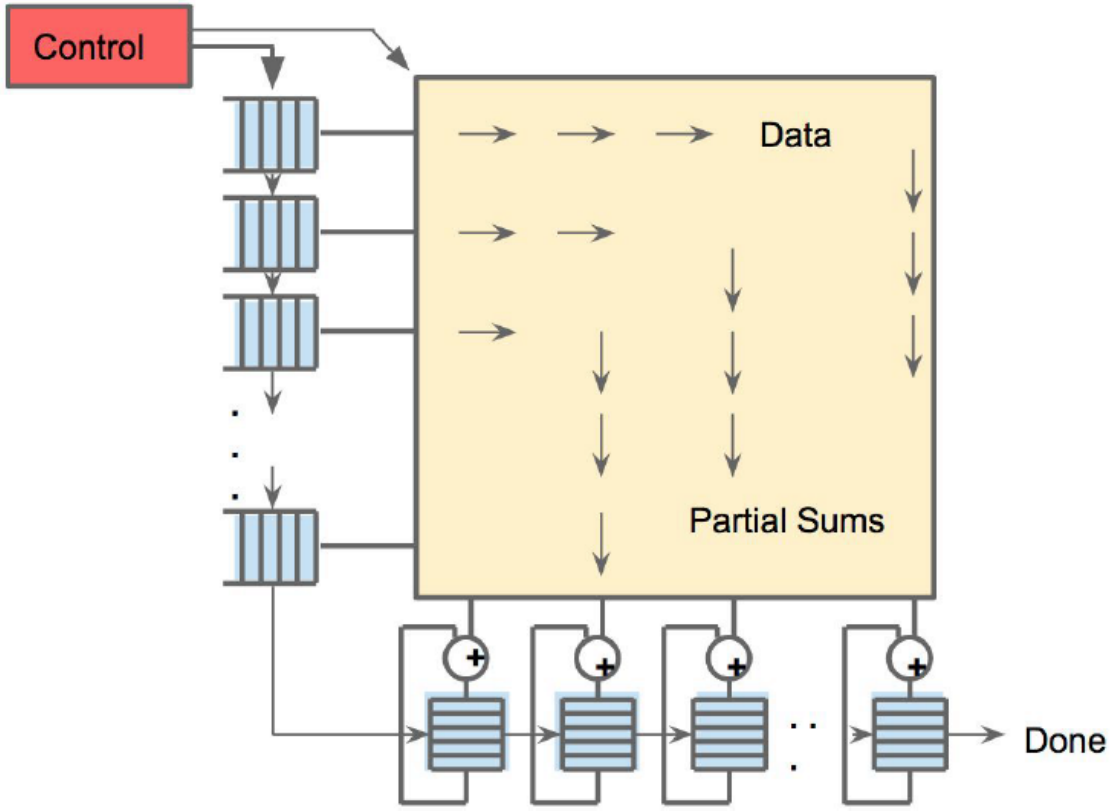
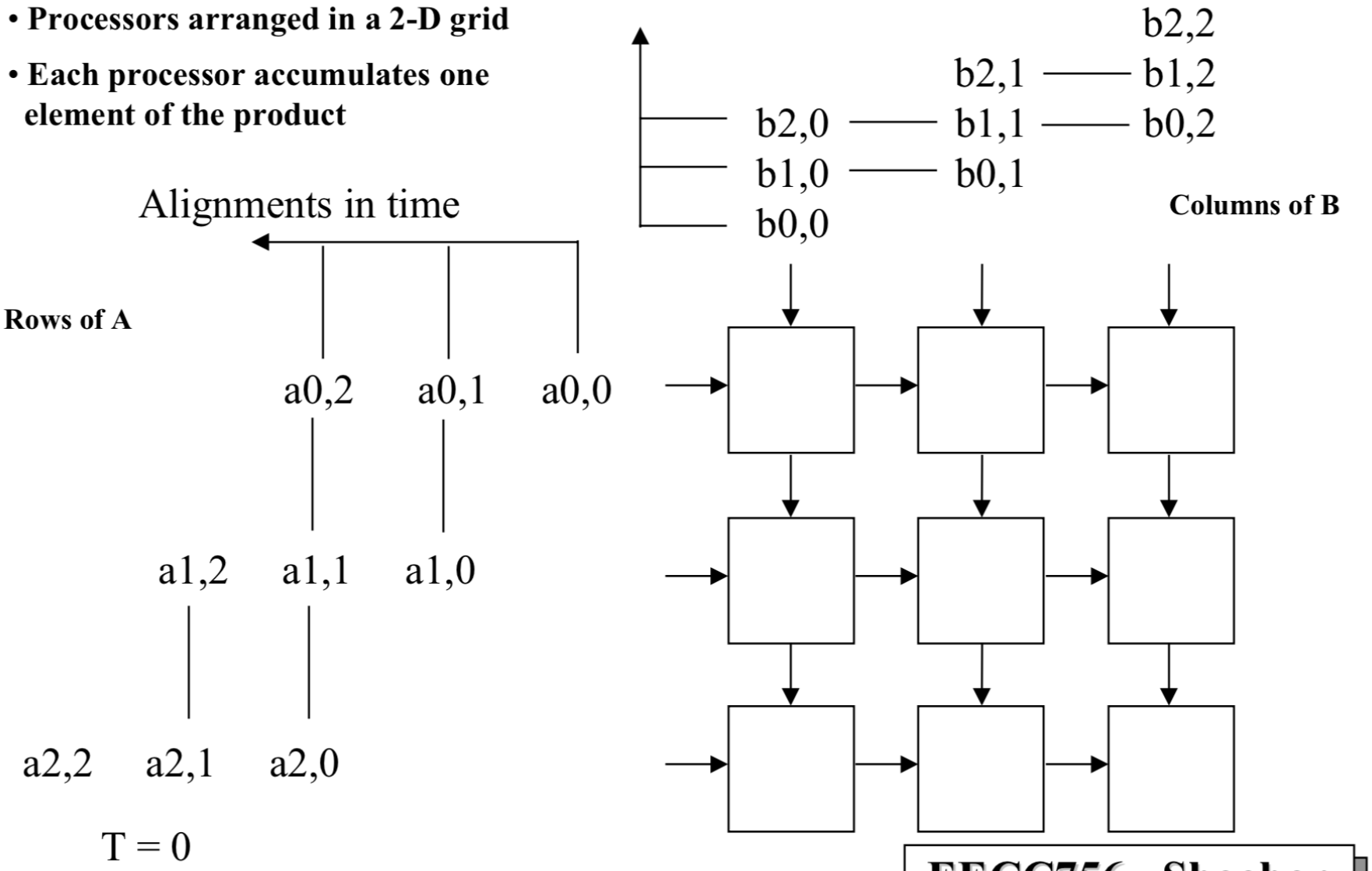


Figure 4. Systolic data flow of the Matrix Multiply Unit. Software has the illusion that each 256B input is read at once, and they instantly update one location of each of 256 accumulator RAMs.

- Control & data are pipelined
  - Reuses both inputs many times as part of producing the output
  - Read input value once but use it for many different operations without storing it back to a register
- Wires only connect spatially adjacent ALUs
  - Short and energy-efficient

Systolic Array Example:
3x3 Systolic Array Matrix Multiplication

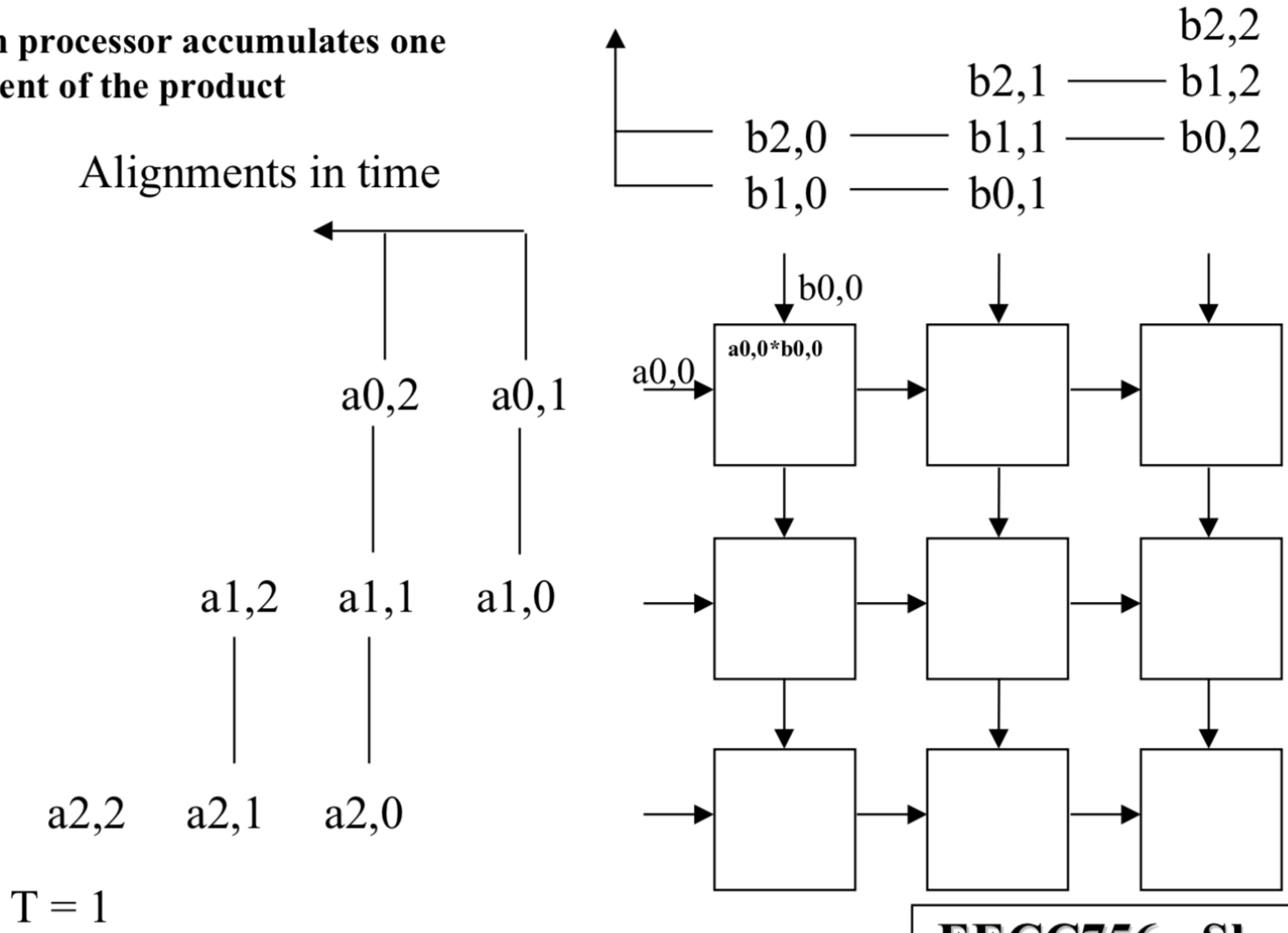
- **Processors arranged in a 2-D grid**
- **Each processor accumulates one element of the product**



Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/

Systolic Array Example:
3x3 Systolic Array Matrix Multiplication

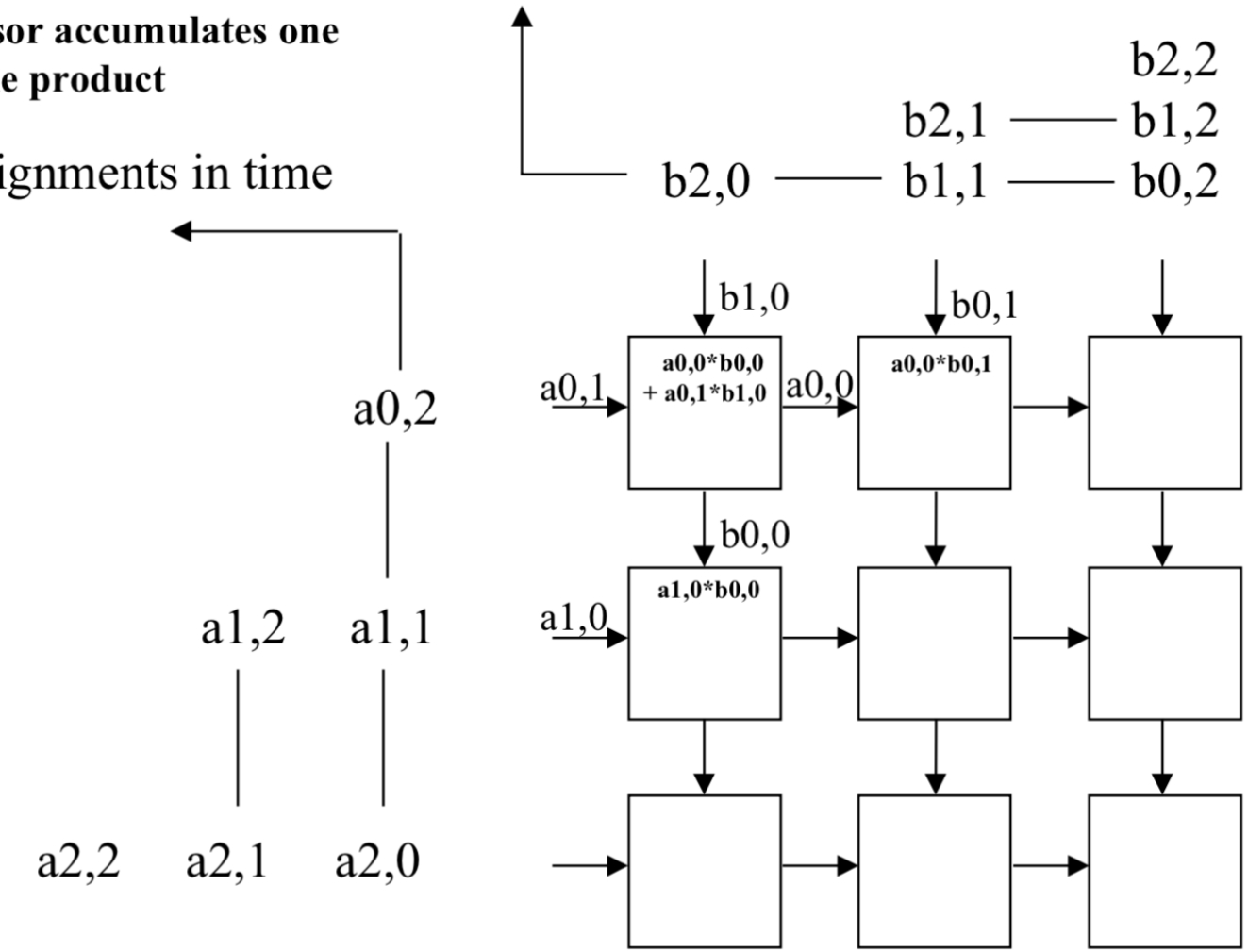
- **Processors arranged in a 2-D grid**
- **Each processor accumulates one element of the product**



Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/

Systolic Array Example:
3x3 Systolic Array Matrix Multiplication

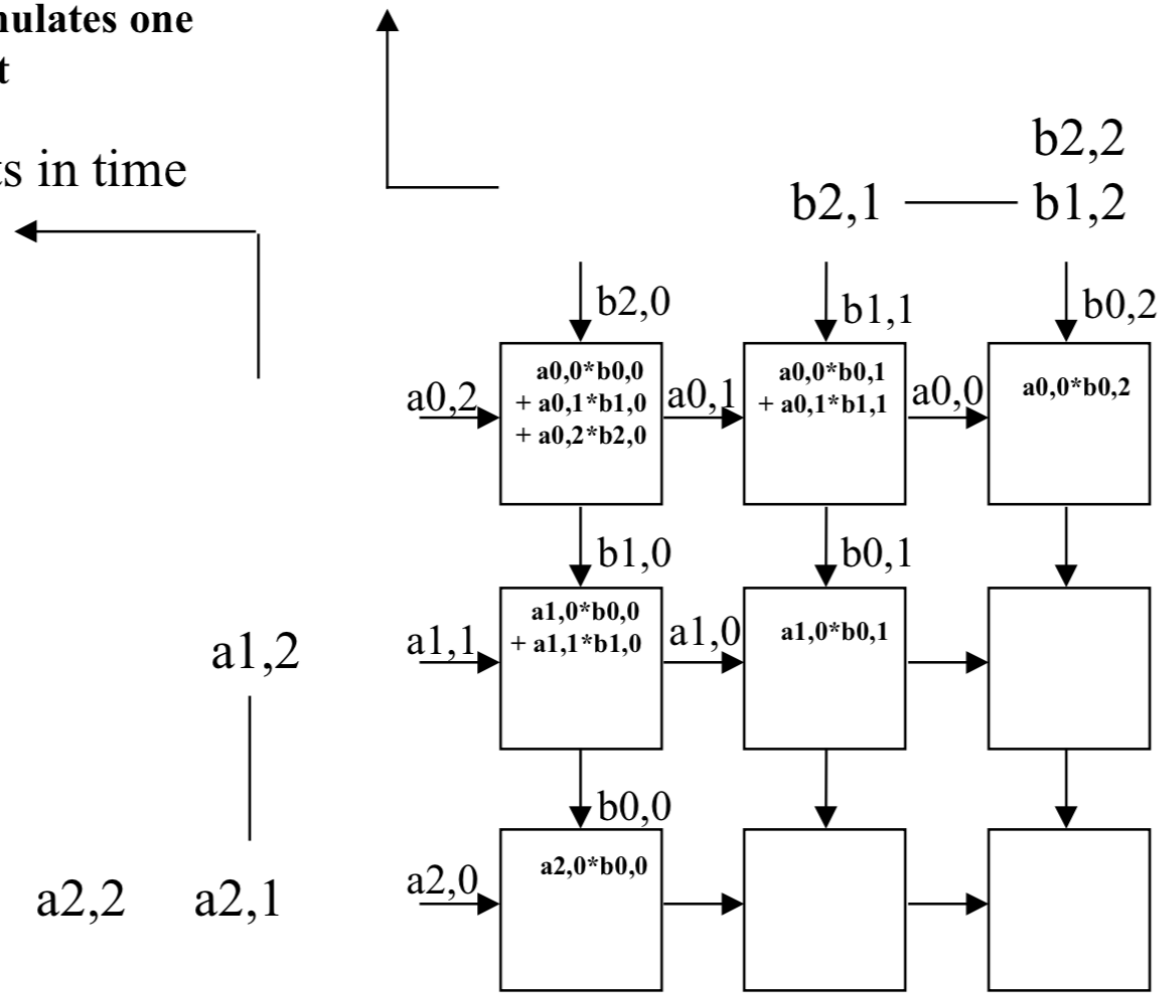
- **Processors arranged in a 2-D grid**
- **Each processor accumulates one element of the product**



Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/

Systolic Array Example:
3x3 Systolic Array Matrix Multiplication

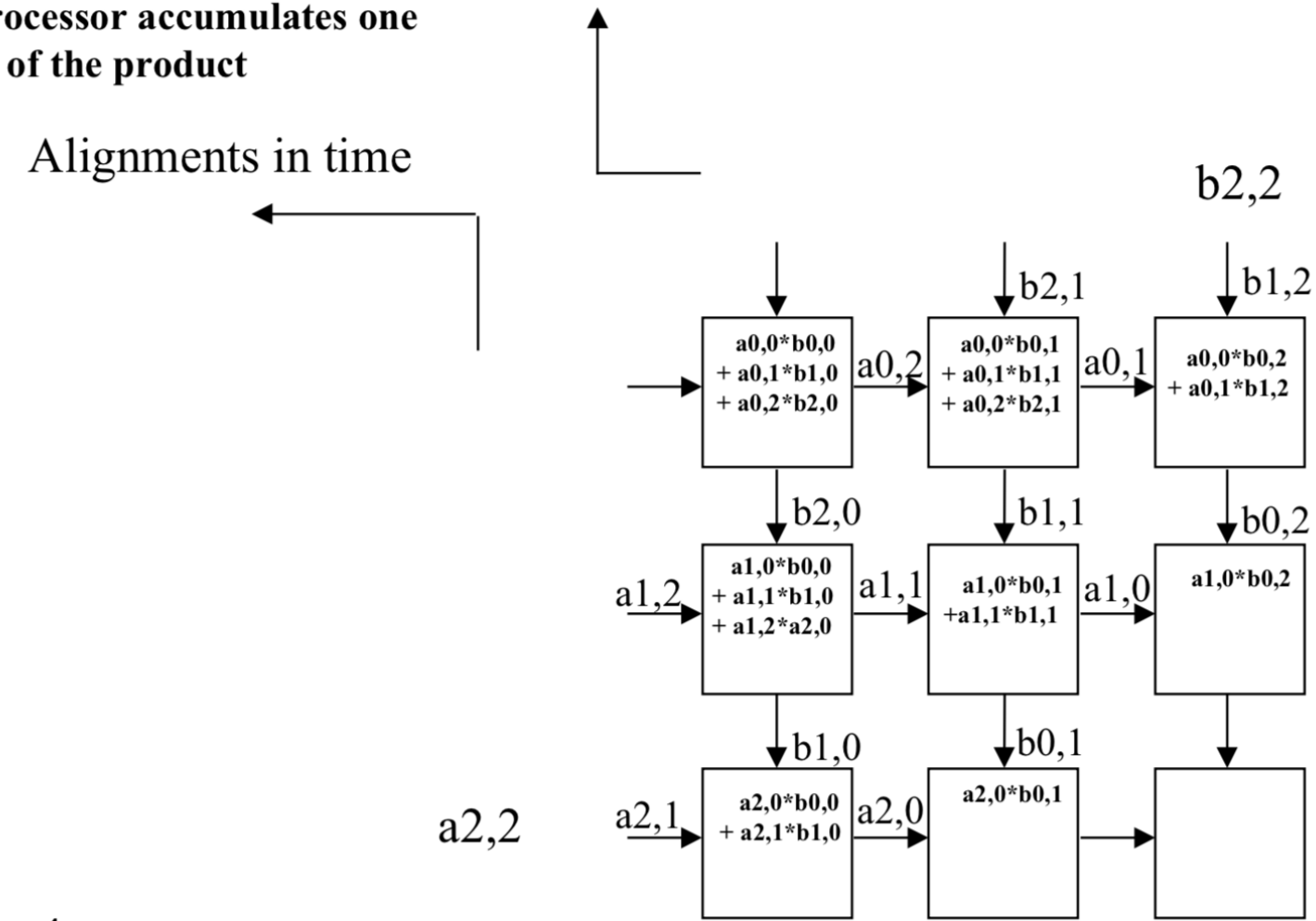
- **Processors arranged in a 2-D grid**
- **Each processor accumulates one element of the product**



Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/

Systolic Array Example:
3x3 Systolic Array Matrix Multiplication

- **Processors arranged in a 2-D grid**
- **Each processor accumulates one element of the product**

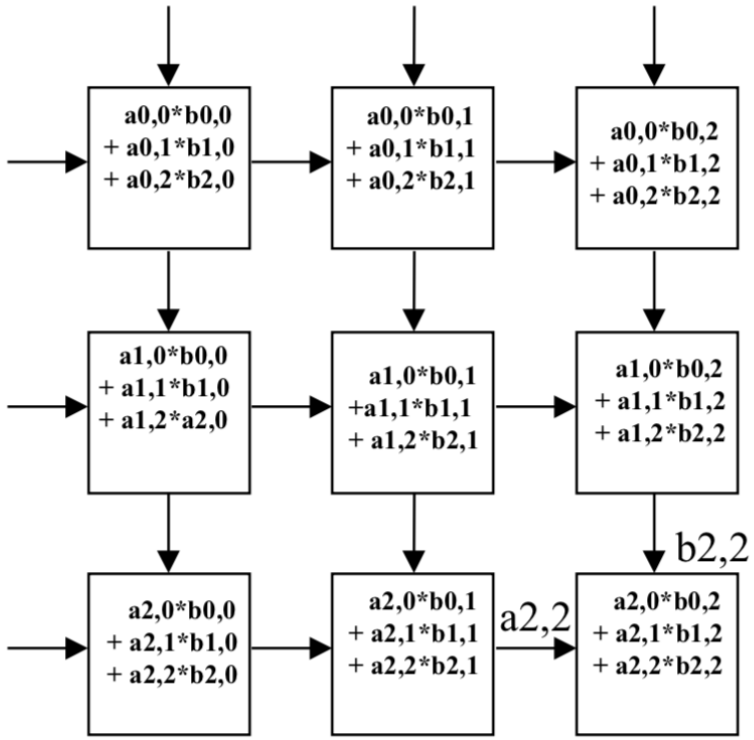


Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/

Systolic Array Example:
3x3 Systolic Array Matrix Multiplication

- **Processors arranged in a 2-D grid**
- **Each processor accumulates one element of the product**

Alignments in time



Done

But, it takes time to read out the results.

T = 7

Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/

[Google, 2017]

TPU Systolic Execution

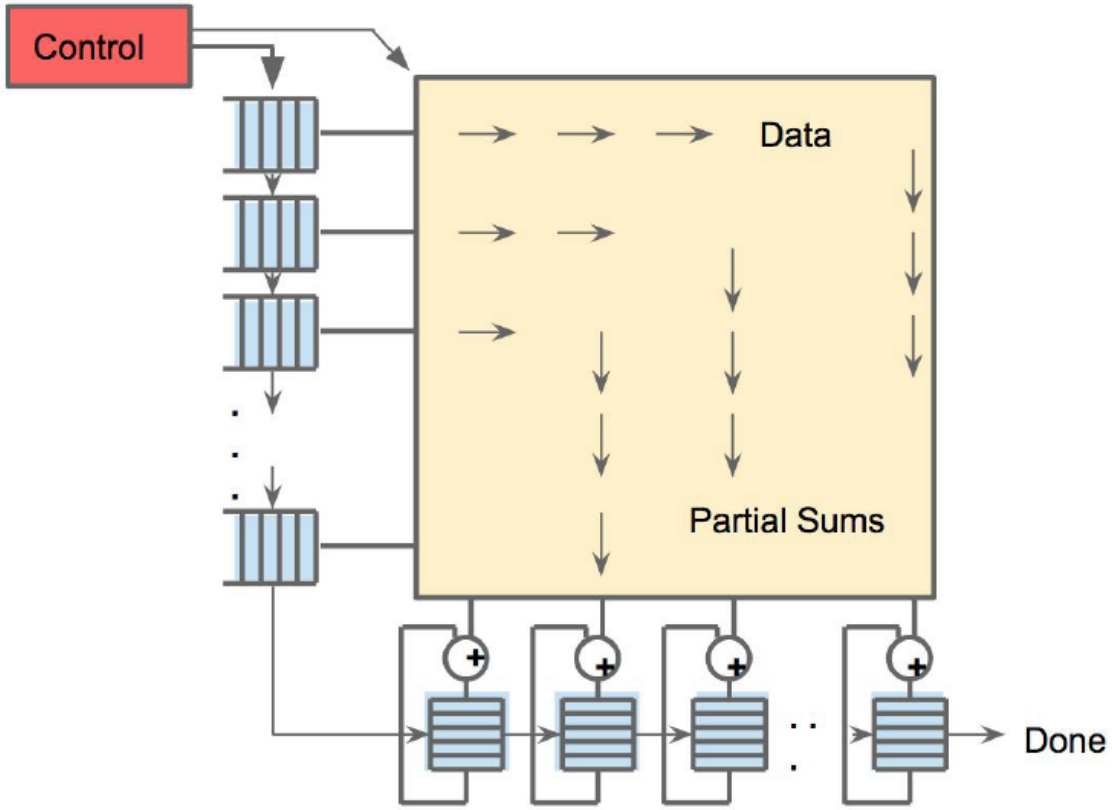
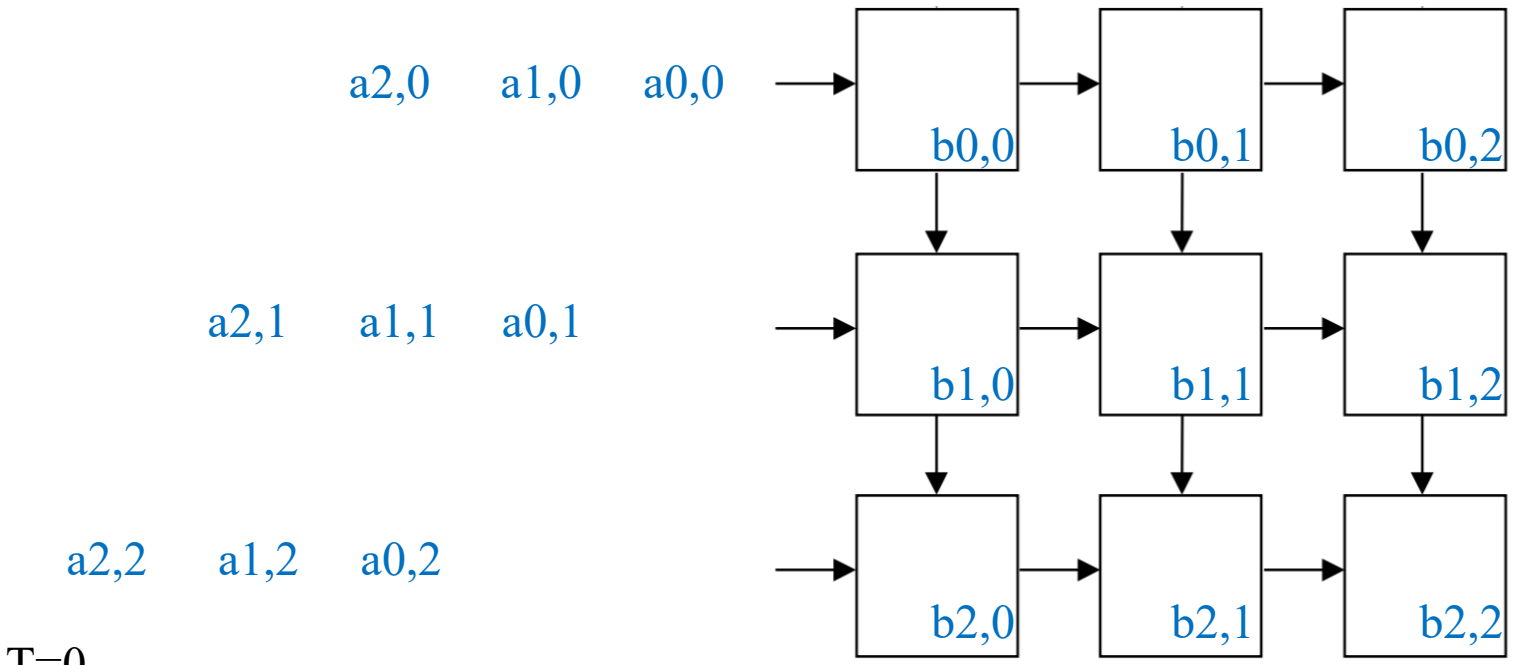


Figure 4. Systolic data flow of the Matrix Multiply Unit. Software has the illusion that each 256B input is read at once, and they instantly update one location of each of 256 accumulator RAMs.

- Partial sums are accumulated while flowing downwards and are kept in the partial sum buffer of its column

Systolic Array Example:
3x3 Systolic Array Matrix Multiplication

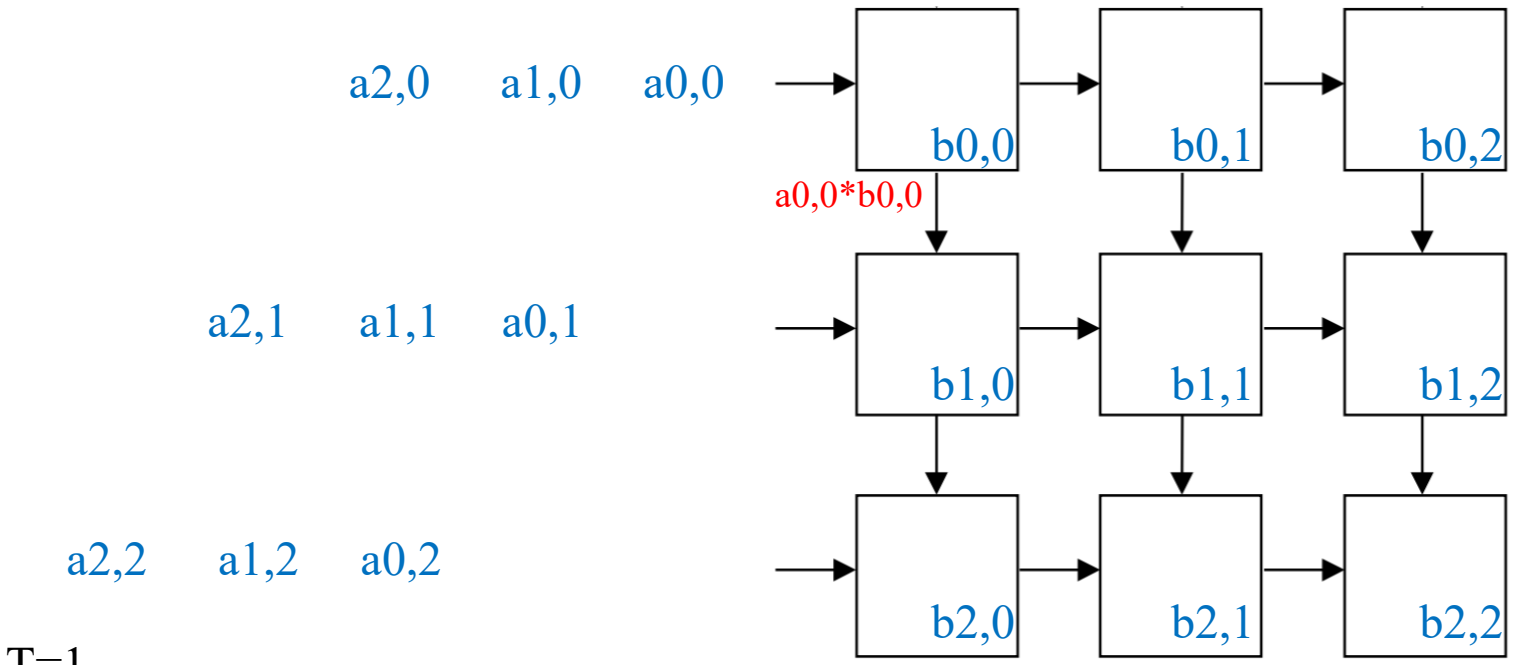
- Processors arranged in a 2-D grid
- Each processor accumulates one element of the product



Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/

Systolic Array Example:
3x3 Systolic Array Matrix Multiplication

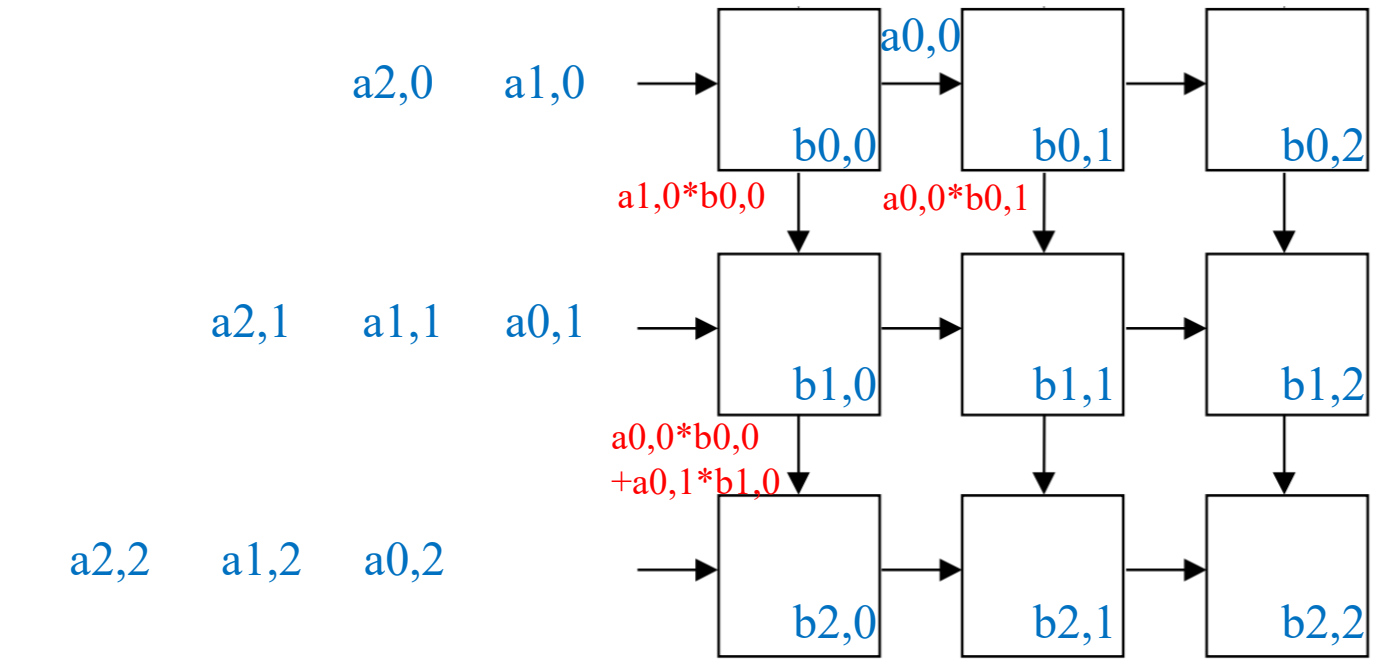
- Processors arranged in a 2-D grid
- Each processor accumulates one element of the product



Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/

Systolic Array Example:
3x3 Systolic Array Matrix Multiplication

- Processors arranged in a 2-D grid
- Each processor accumulates one element of the product

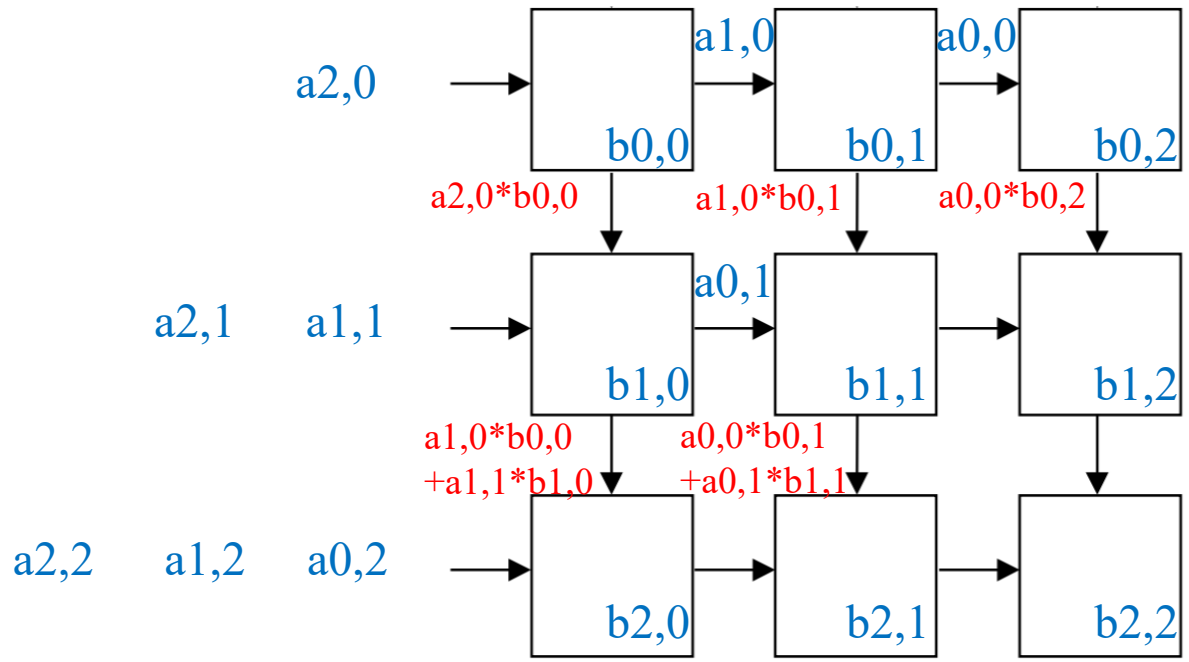


T=2

Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/

Systolic Array Example:
3x3 Systolic Array Matrix Multiplication

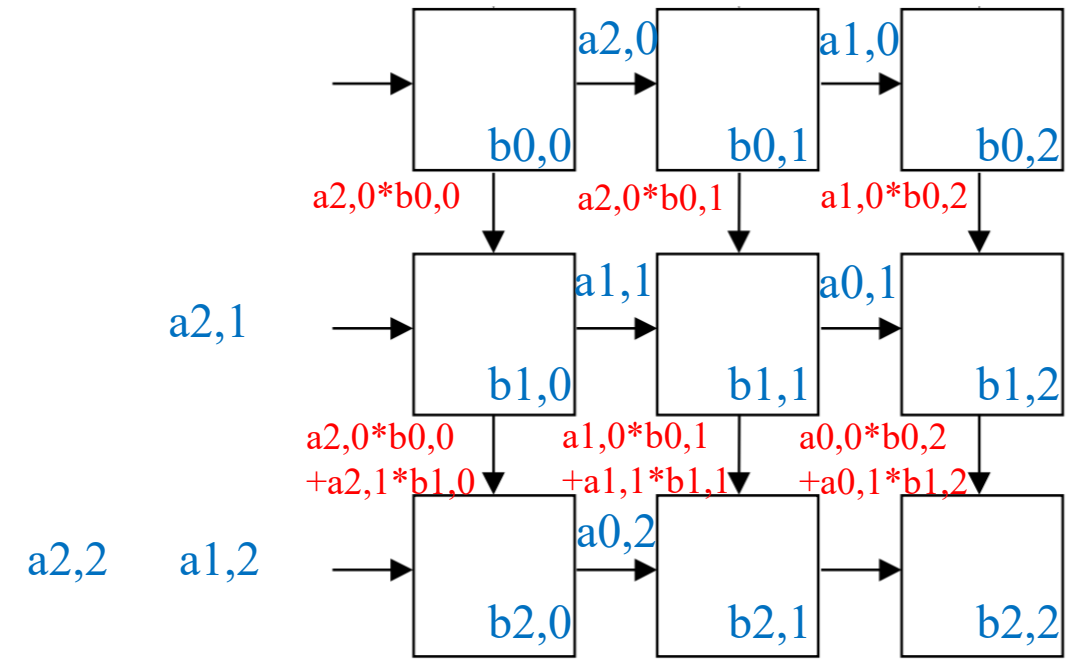
- Processors arranged in a 2-D grid
- Each processor accumulates one element of the product



Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/

Systolic Array Example:
3x3 Systolic Array Matrix Multiplication

- Processors arranged in a 2-D grid
- Each processor accumulates one element of the product



T=4

Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/

[Google, 2017]

TPU Systolic Execution

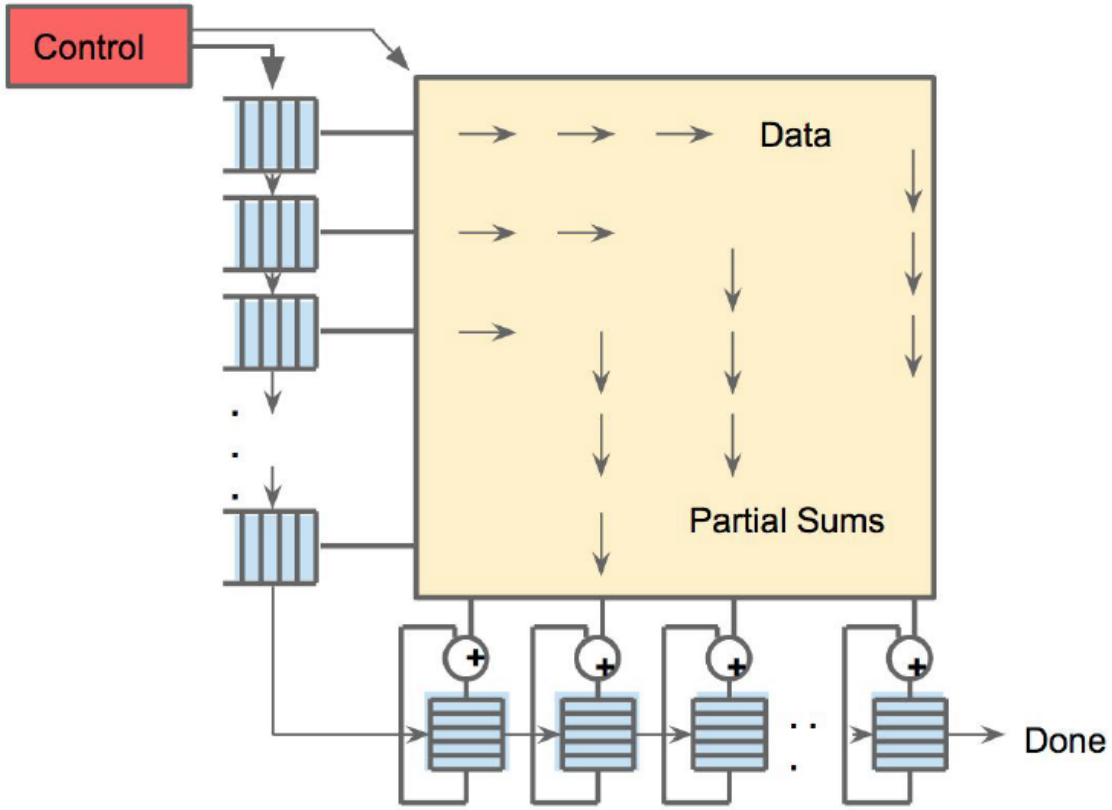
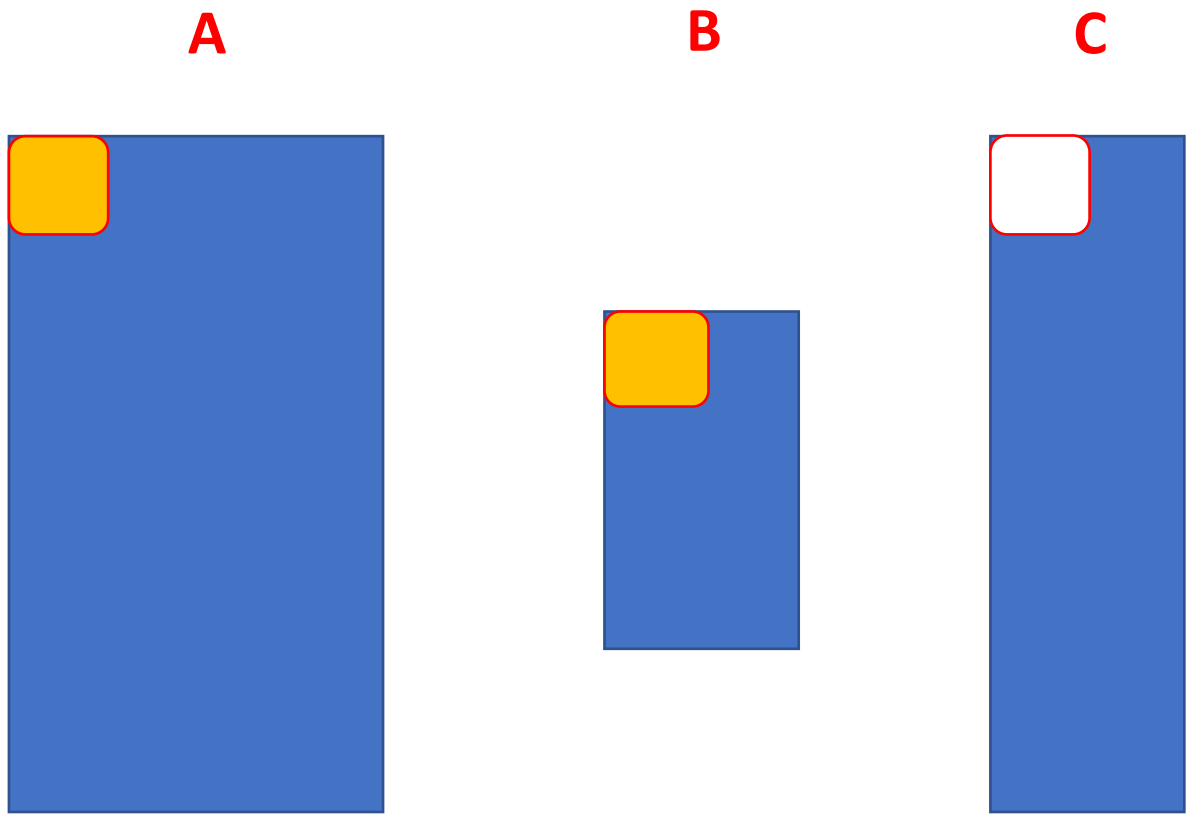


Figure 4. Systolic data flow of the Matrix Multiply Unit. Software has the illusion that each 256B input is read at once, and they instantly update one location of each of 256 accumulator RAMs.

- Partial sums are accumulated while flowing downwards and are kept in the partial sum buffer of its column

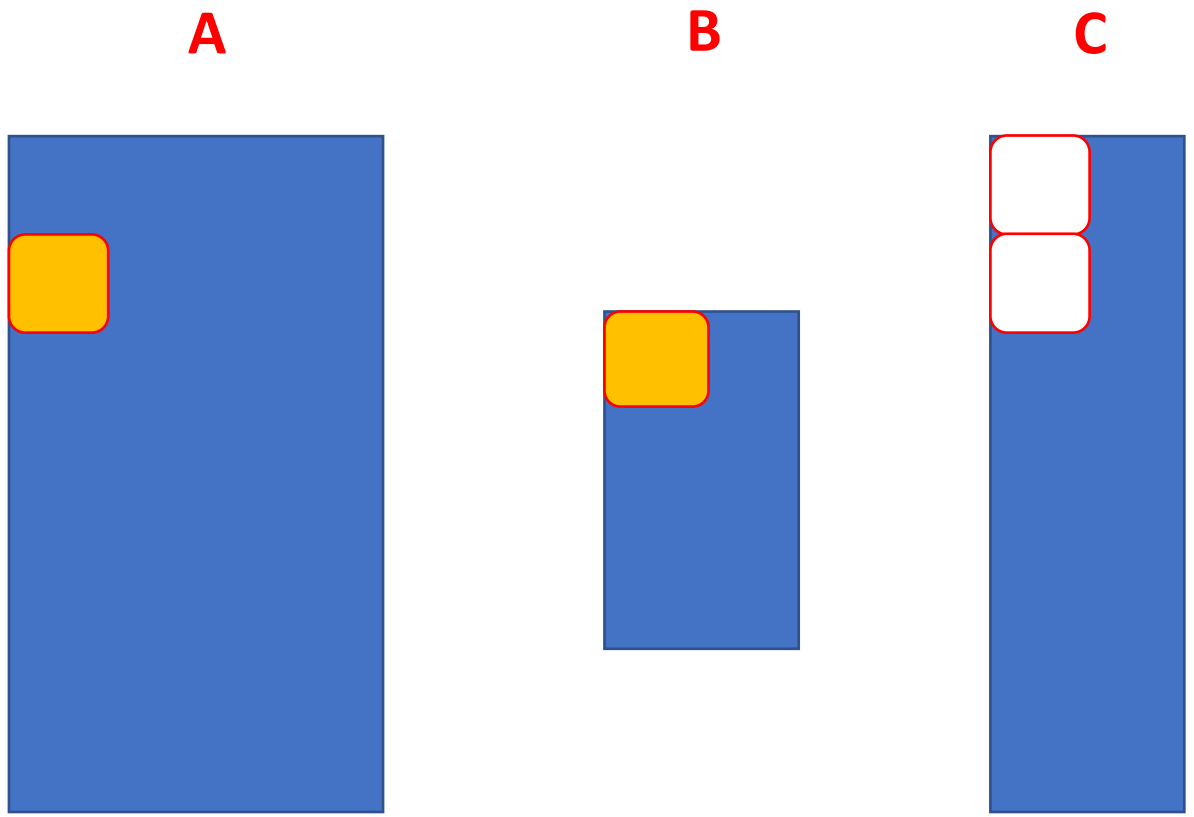
Keeping, i.e. Reusing A Tile in the Systolic Array

- Assume B's tile is kept in the systolic array

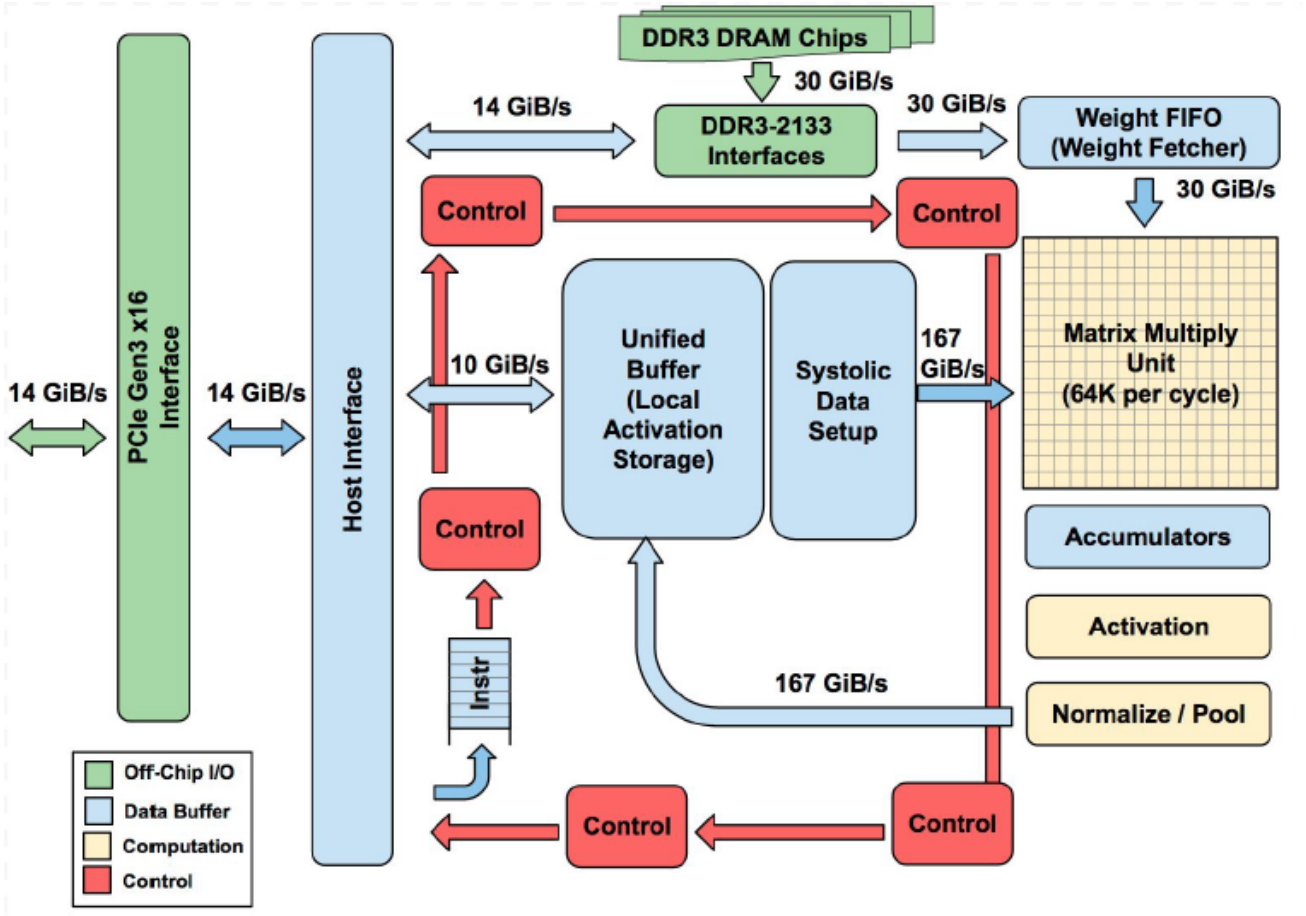


Keeping, i.e. Reusing A Tile in the Systolic Array

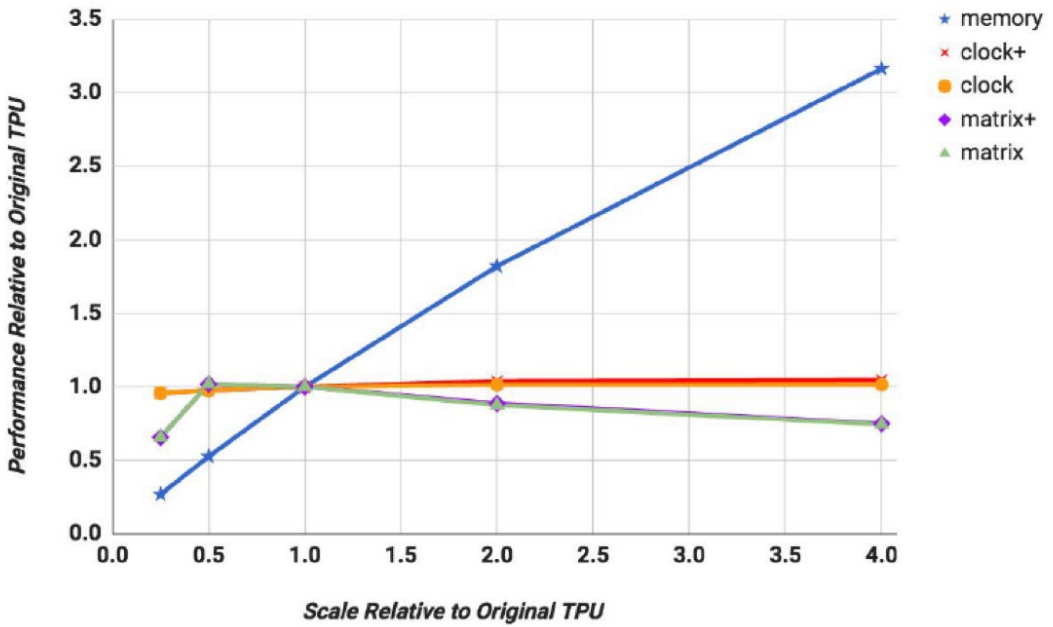
- For the new A's tile, the systolic array performs computation
- The new partial sums are calculated in the partial sum buffer



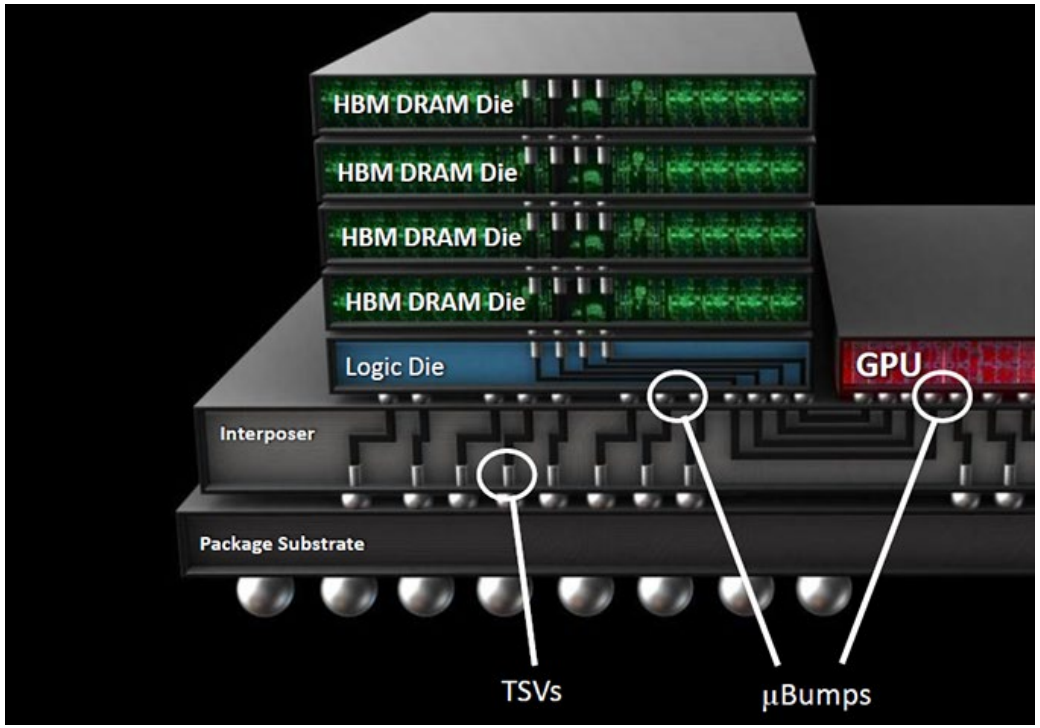
Problem in TPU v1: High Steady State Memory BW for Weights in Tiling of M*M



- Steady state memory bandwidth for weights
  - 256B/cycle * 0.7GHz = 179GB/s
- DDR3 bandwidth, 30GB/s



Solution? High Bandwidth Memory

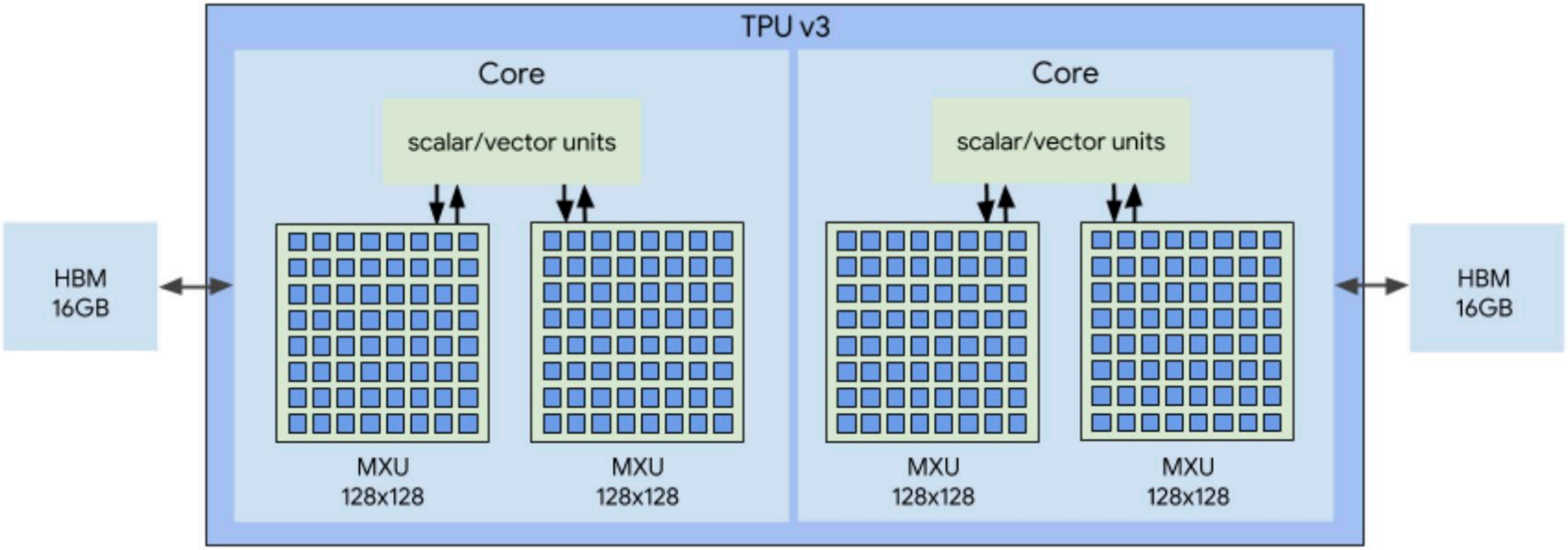


Item	GDDR6	HBM2
DRAM density	16 Gb (per chip)	64 Gb (per stack)
# Channels / DRAM package	2 channels	8 channels
# Bits in a channel	16 bits	128 bits
Speed	16 Gb/s	2.4 Gb/s
Overall bandwidth	64 GB/s	307 GB/s
Power efficiency		Better than GDDR6
Cost	Lower cost than HBM2	
Packaging process	Traditional DRAM on PCB	Uses a 2.5D Interposer for connectivity between the host and the HBM2 DRAMs

[Google, 2018]

TPUv3 Uses Two High Bandwidth Memory (HBM) Chips

- More instances of smaller matrix unit
- 16-bit bfloat (brain floating point)



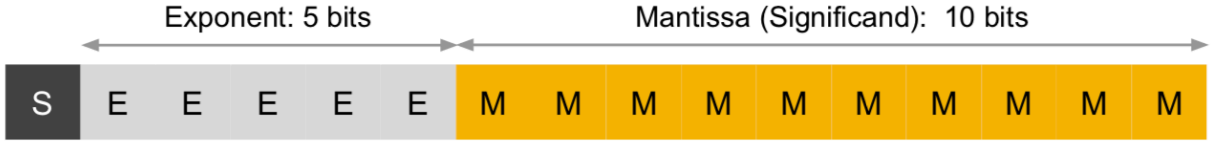
32 and 16 bit Floating Point Representations

float32: Single-precision IEEE Floating Point Format

Range: ~1e^{-38} to ~3e^{38}

float16: Half-precision IEEE Floating Point Format

Range: ~5.96e^{-8} to 65504

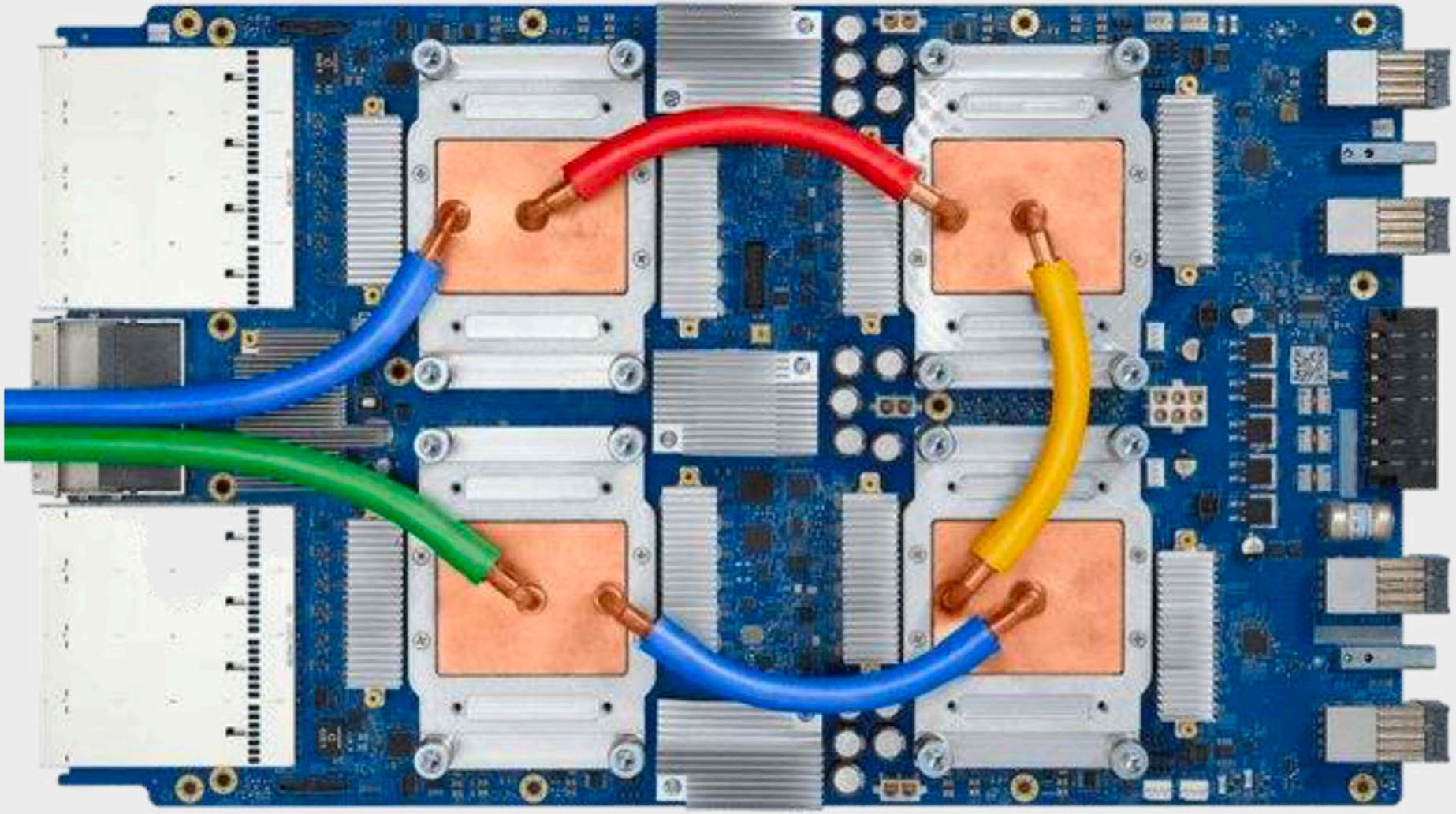


bfloat16: Brain Floating Point Format

Range: ~1e^{-38} to ~3e^{38}

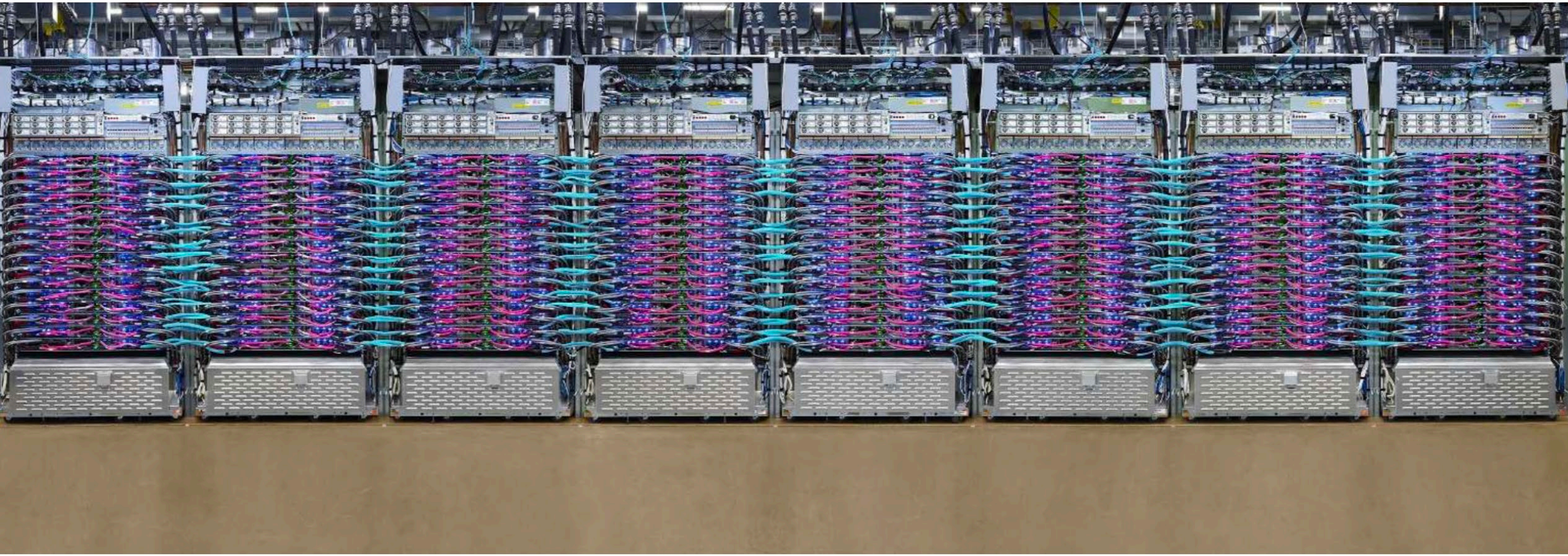
Google TPU
Intel Spring Crest
Habana, …

Cloud TPU v3 Board



- 420 teraflops of computation, 128 GB of HBM memory
- Liquid cooling

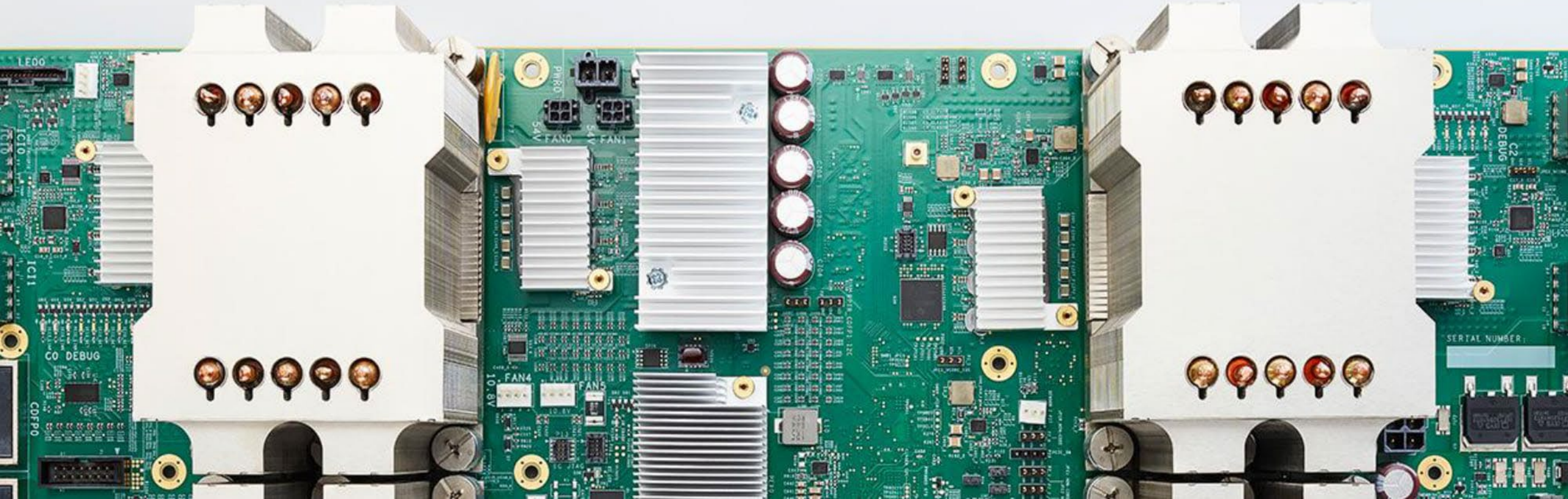
Cloud TPU v3 Pod

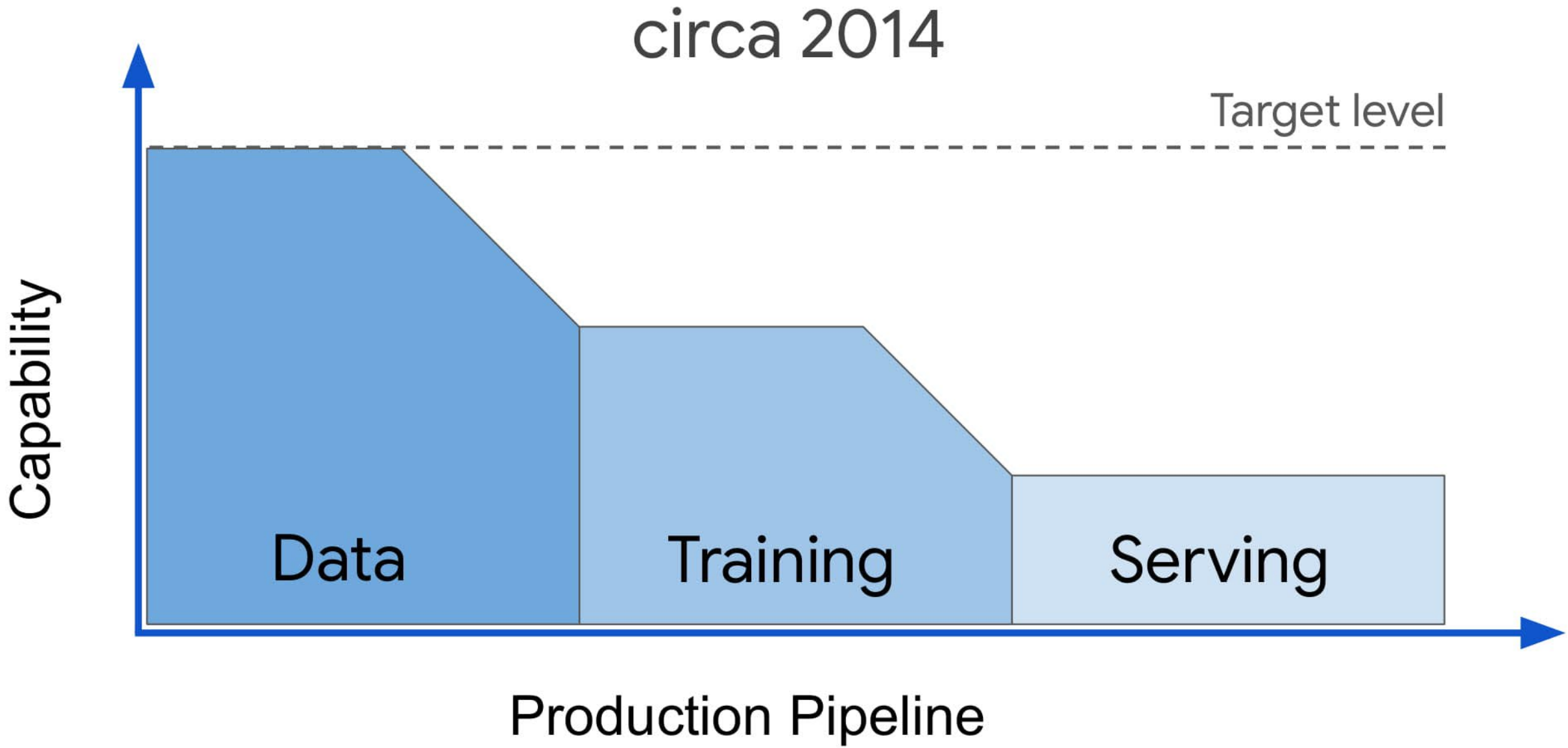


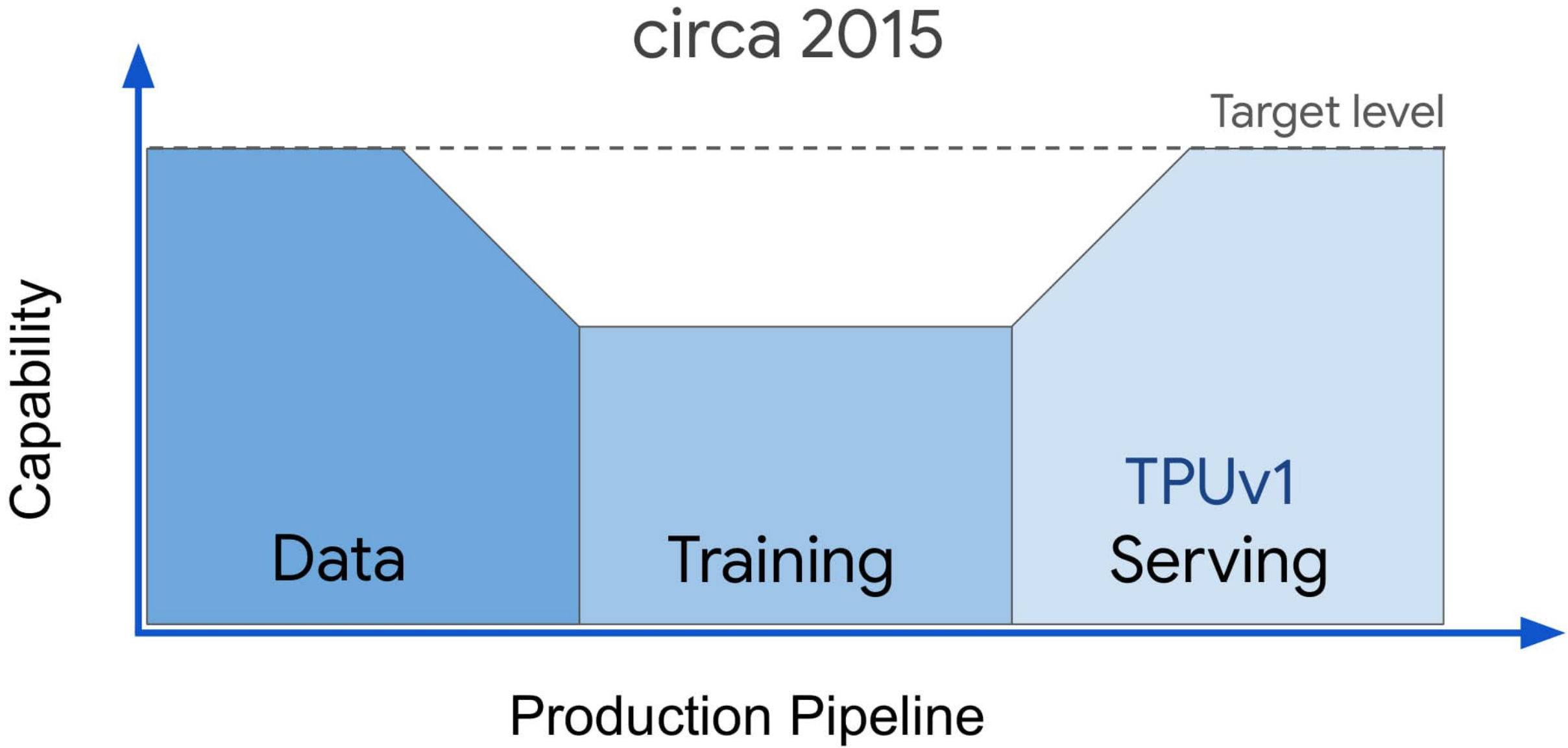
- >100 pflops
- 32TB HBM

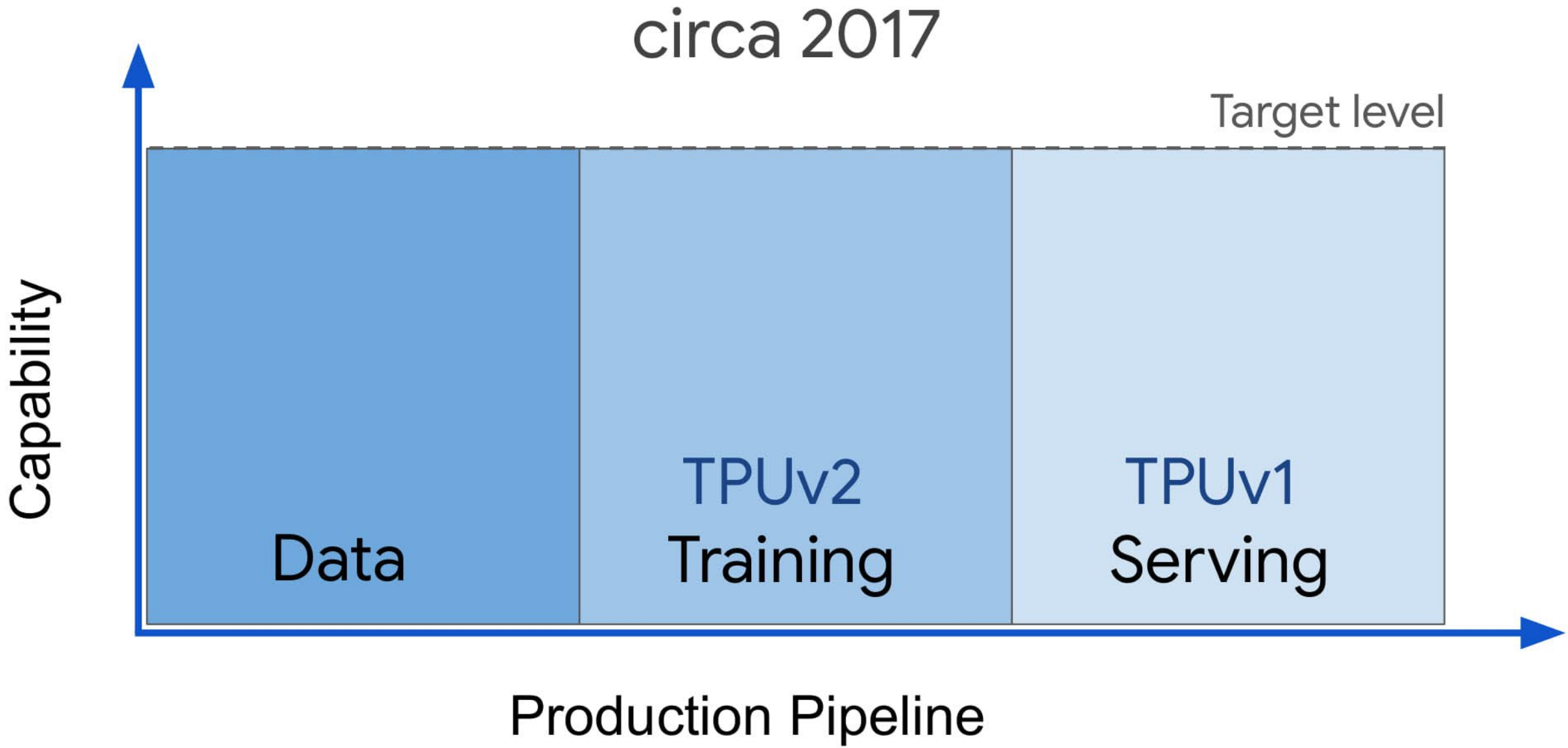
Google's Training Chips Revealed: TPUv2 and TPUv3

Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman P. Jouppi, and David Patterson

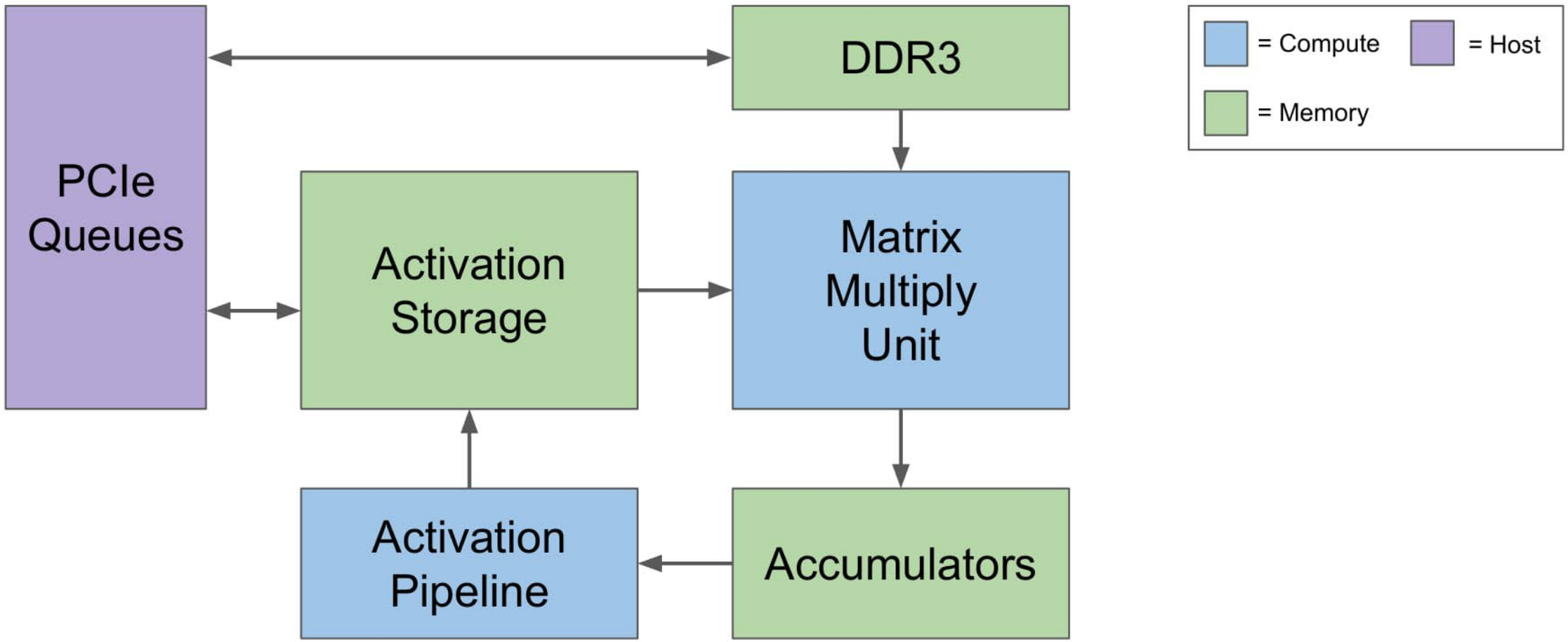



circa 2014
Target level
Capability
Data
Training
Serving
Production Pipeline

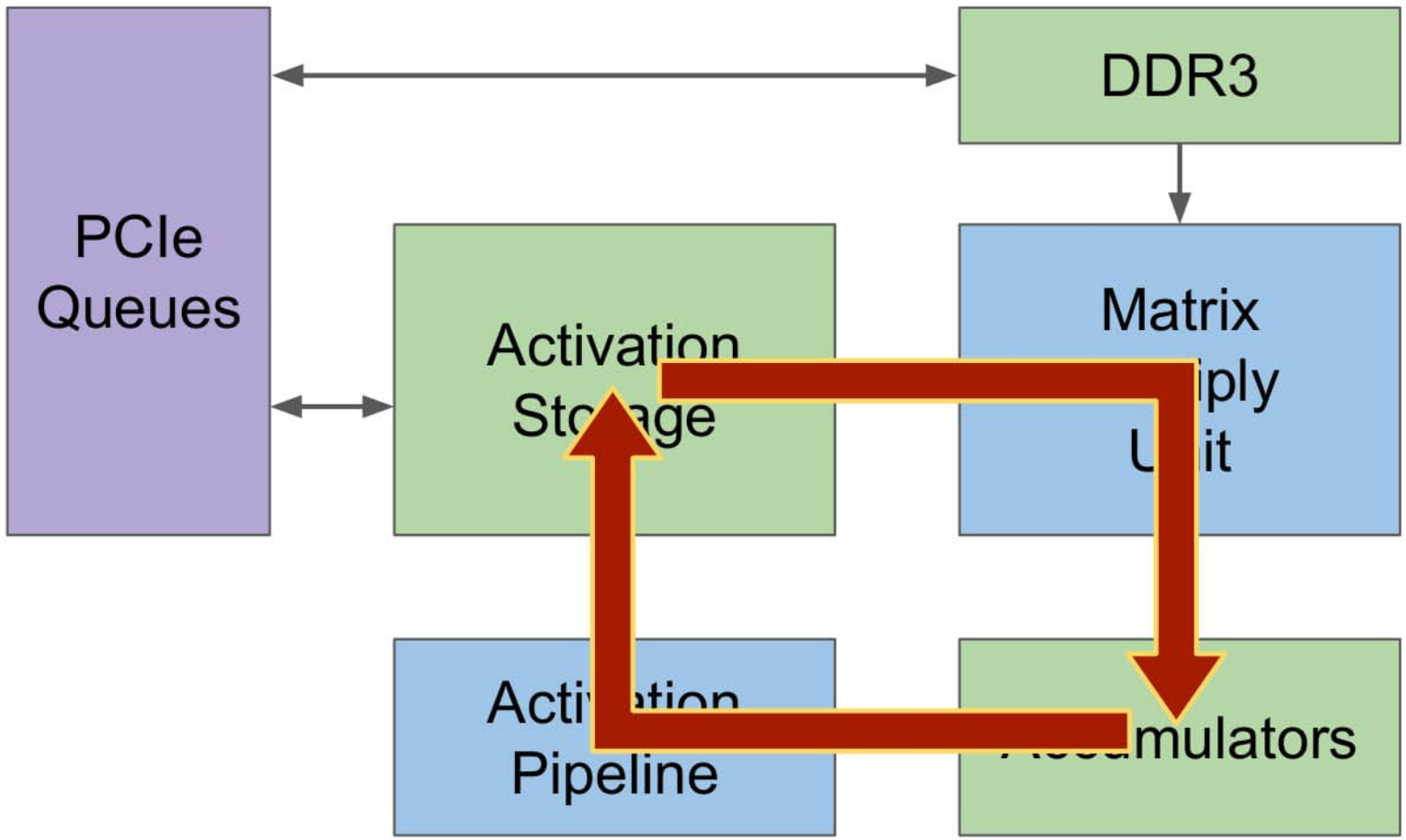

circa 2015
Target level
Capability
Data
Training
TPUv1
Serving
Production Pipeline


circa 2017
Target level
Capability
Data
TPUv2
Training
TPUv1
Serving
Production Pipeline

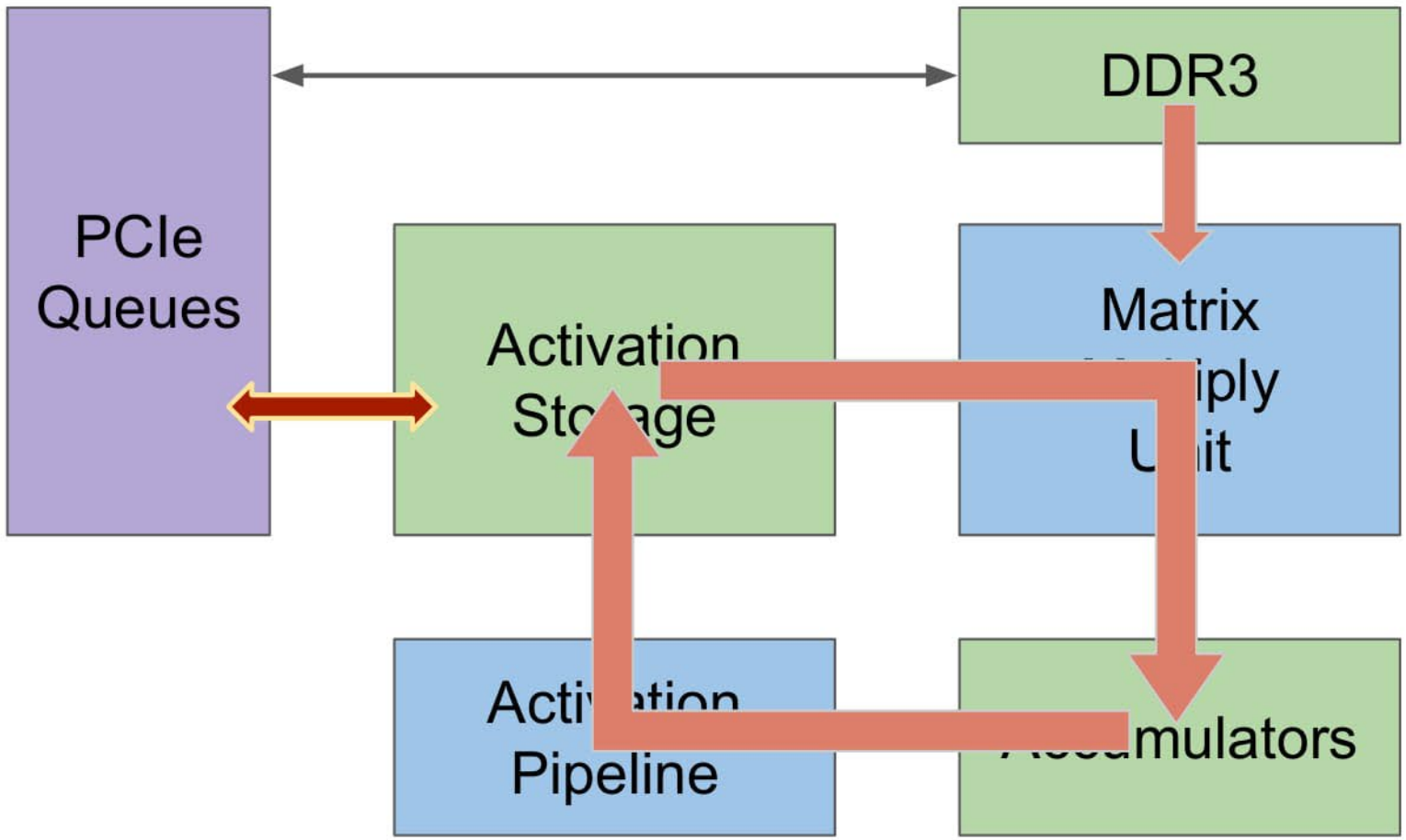
TPUv1 Recap


PCIe Queues
DDR3
Activation Storage
Matrix Multiply Unit
Activation Pipeline
Accumulators
= Compute
= Host
= Memory

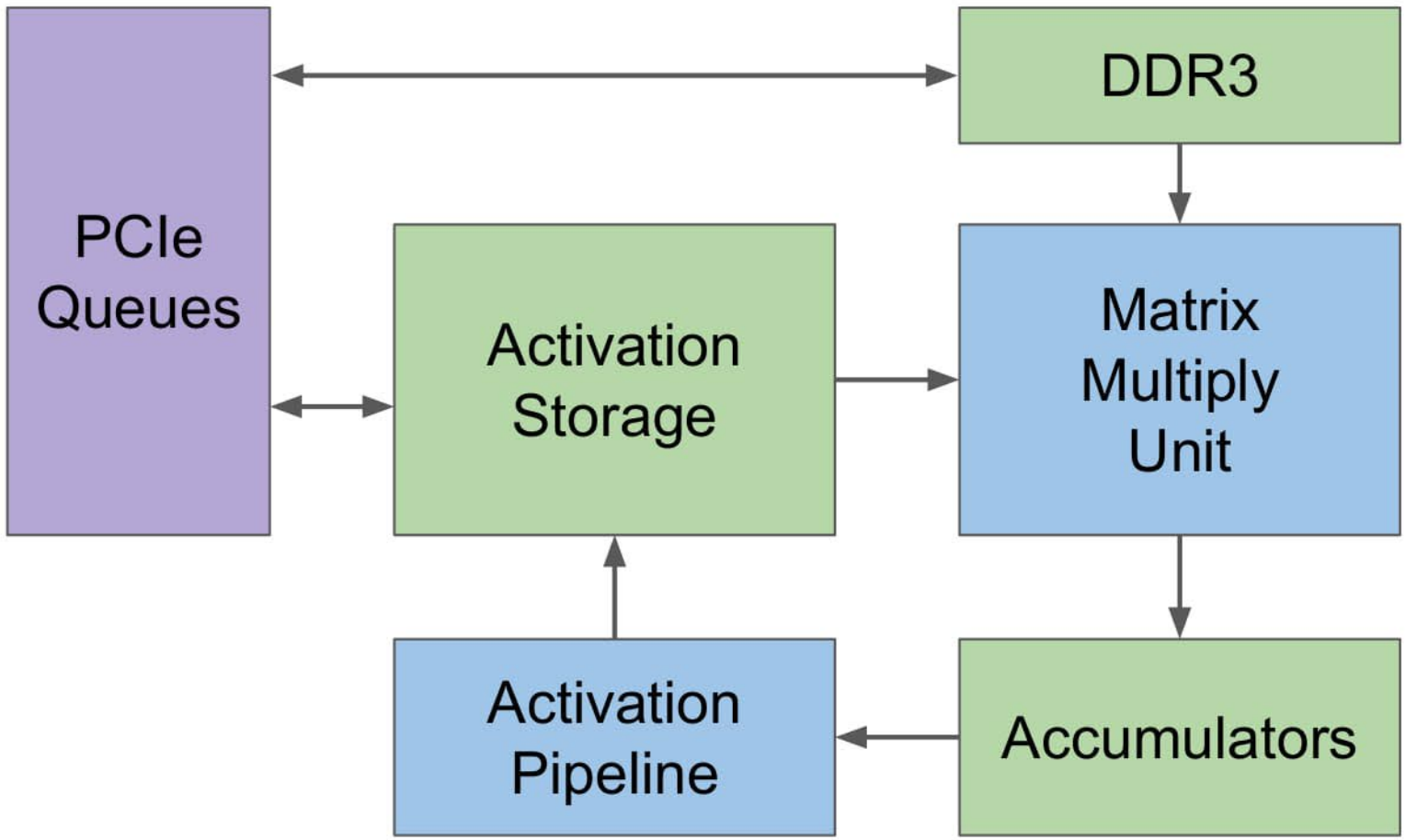
TPUv1 Recap


PCIe
Queues
DDR3
Activation
Matrix
Pipeline
Accumulators

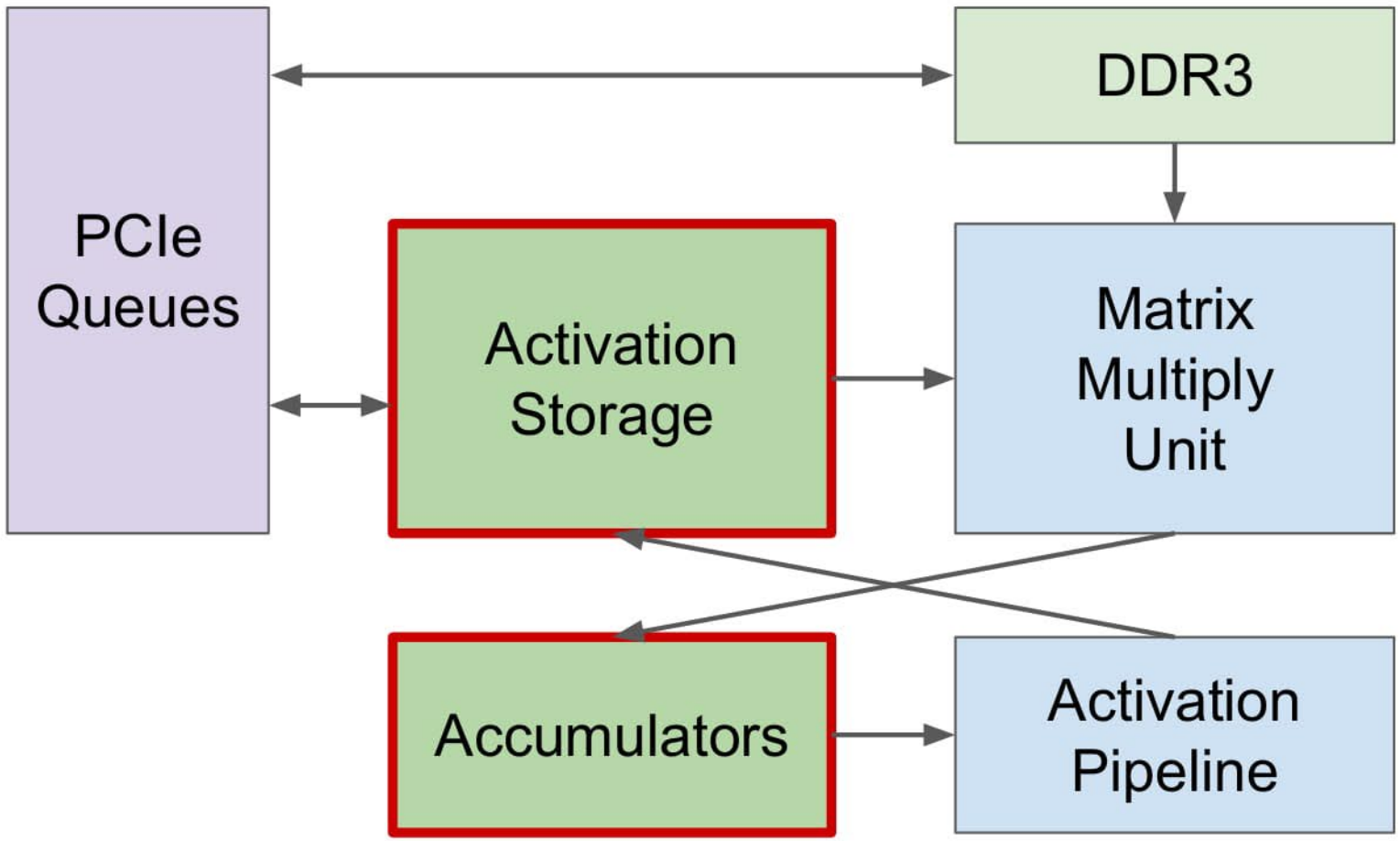
TPUv1 Recap


PCIe
Queues
DDR3
Activation
Matrix
Pipeline
Accumulators

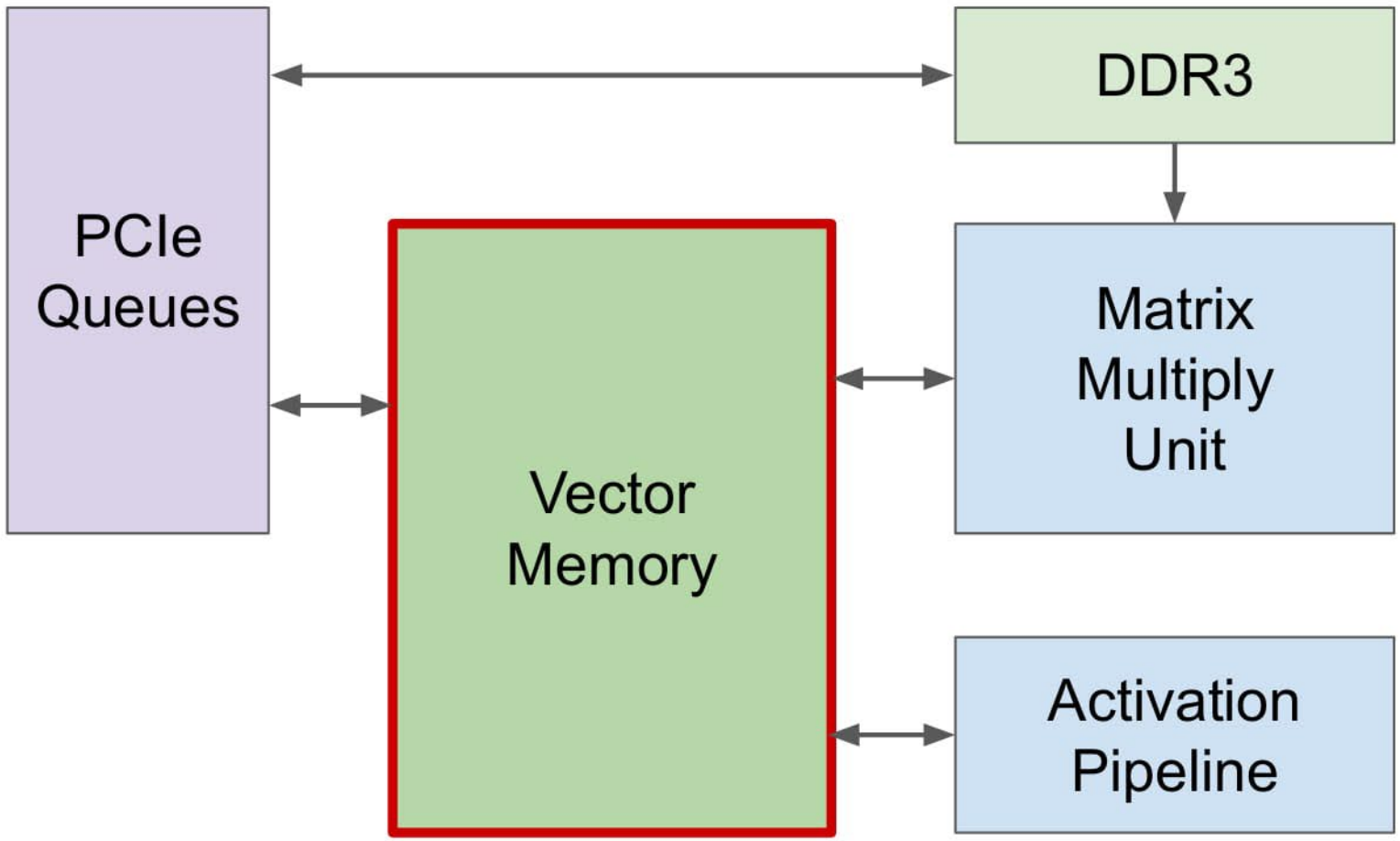
TPUv2 Changes



TPUv2 Changes

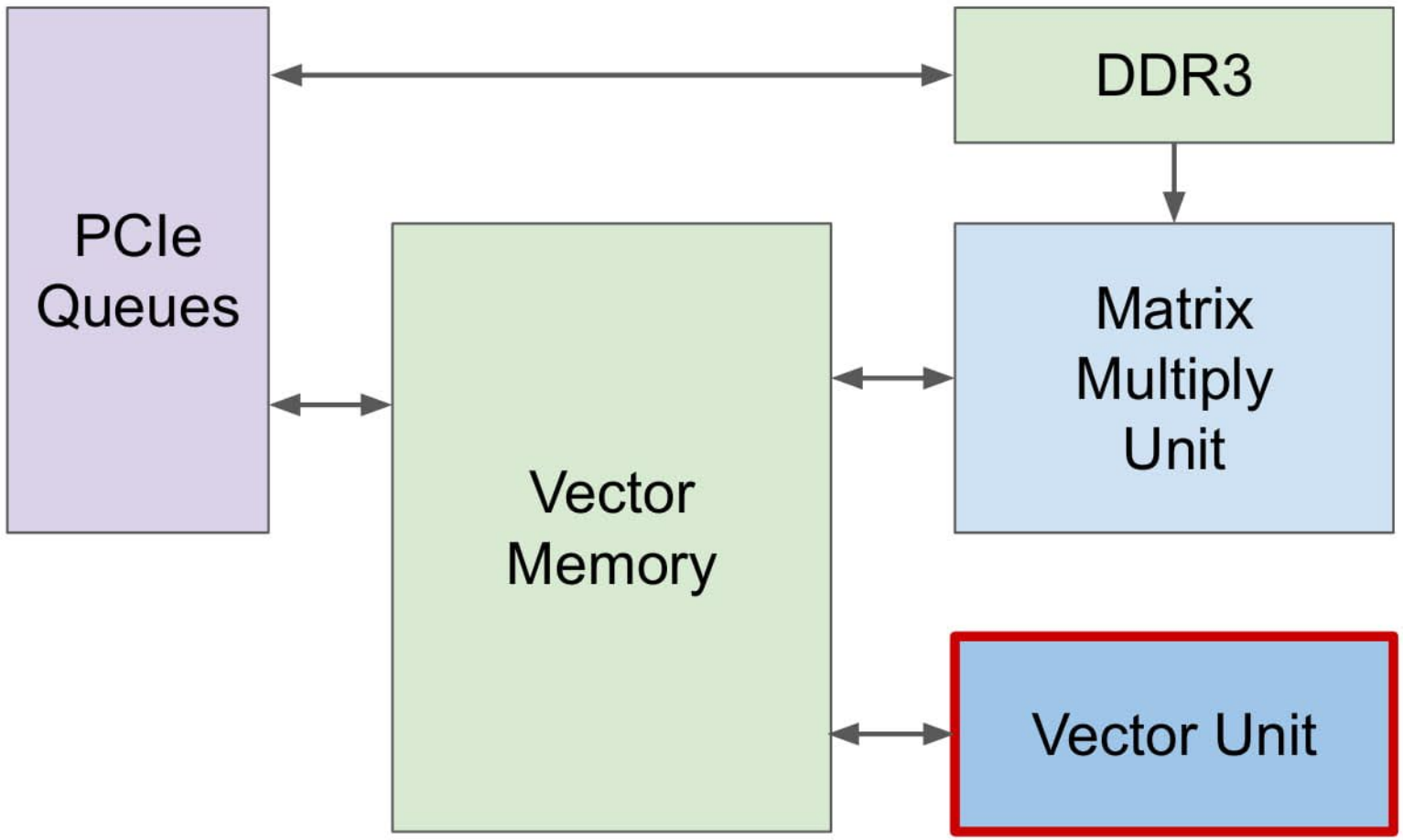


TPUv2 Changes



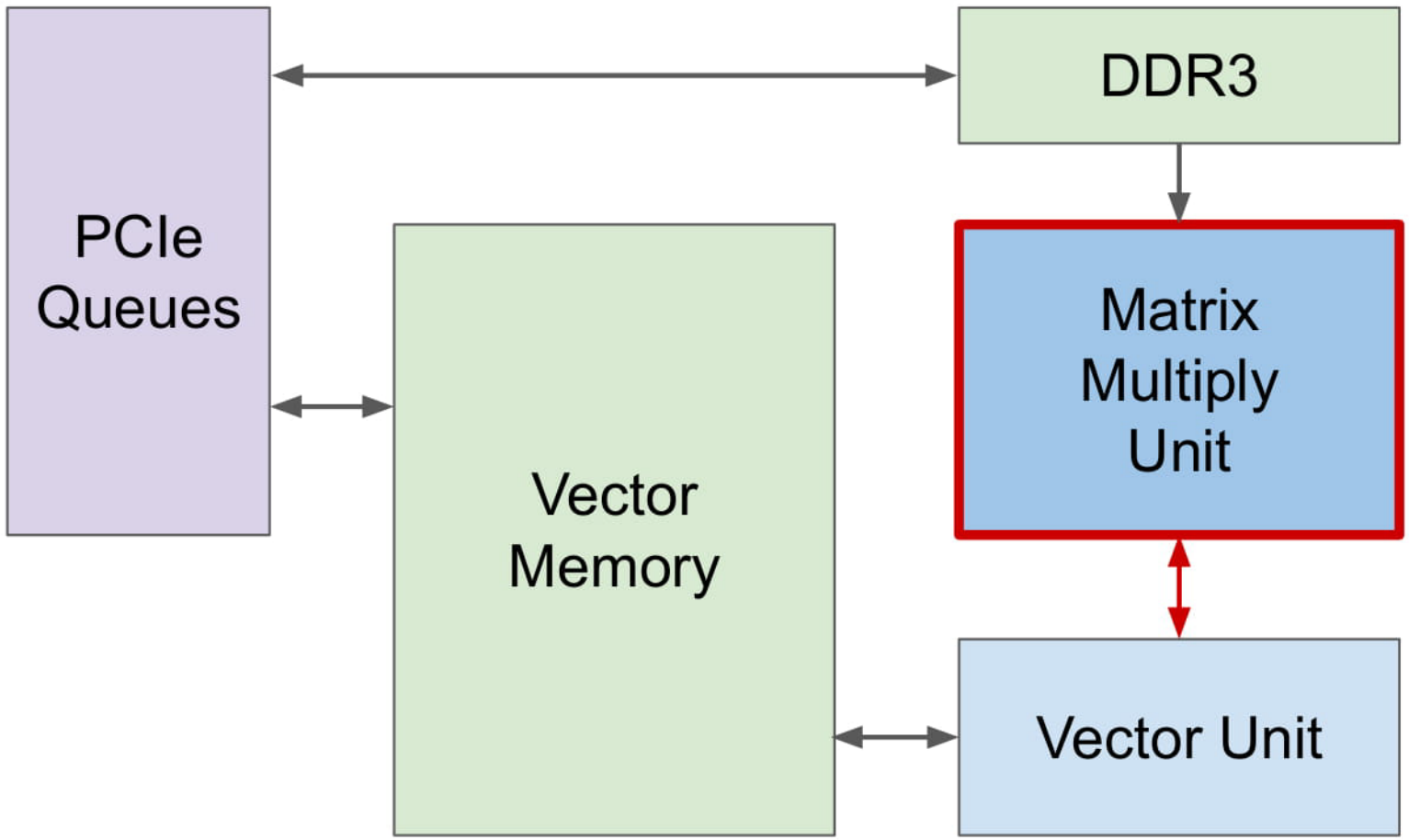
Single vector memory instead of buffers between fixed function units.

TPUv2 Changes



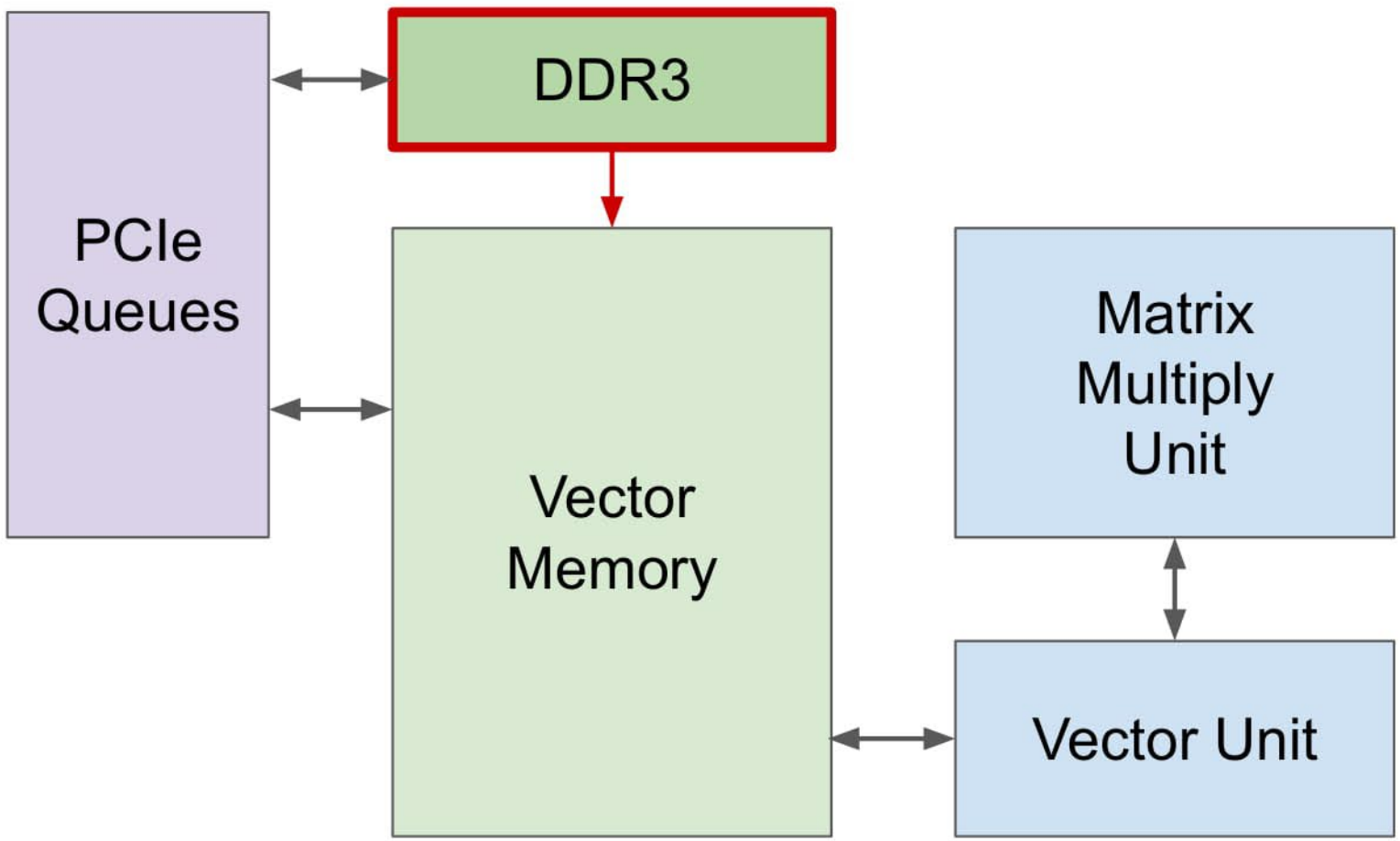
General purpose vector unit instead of a fixed function activation pipeline.

TPUv2 Changes



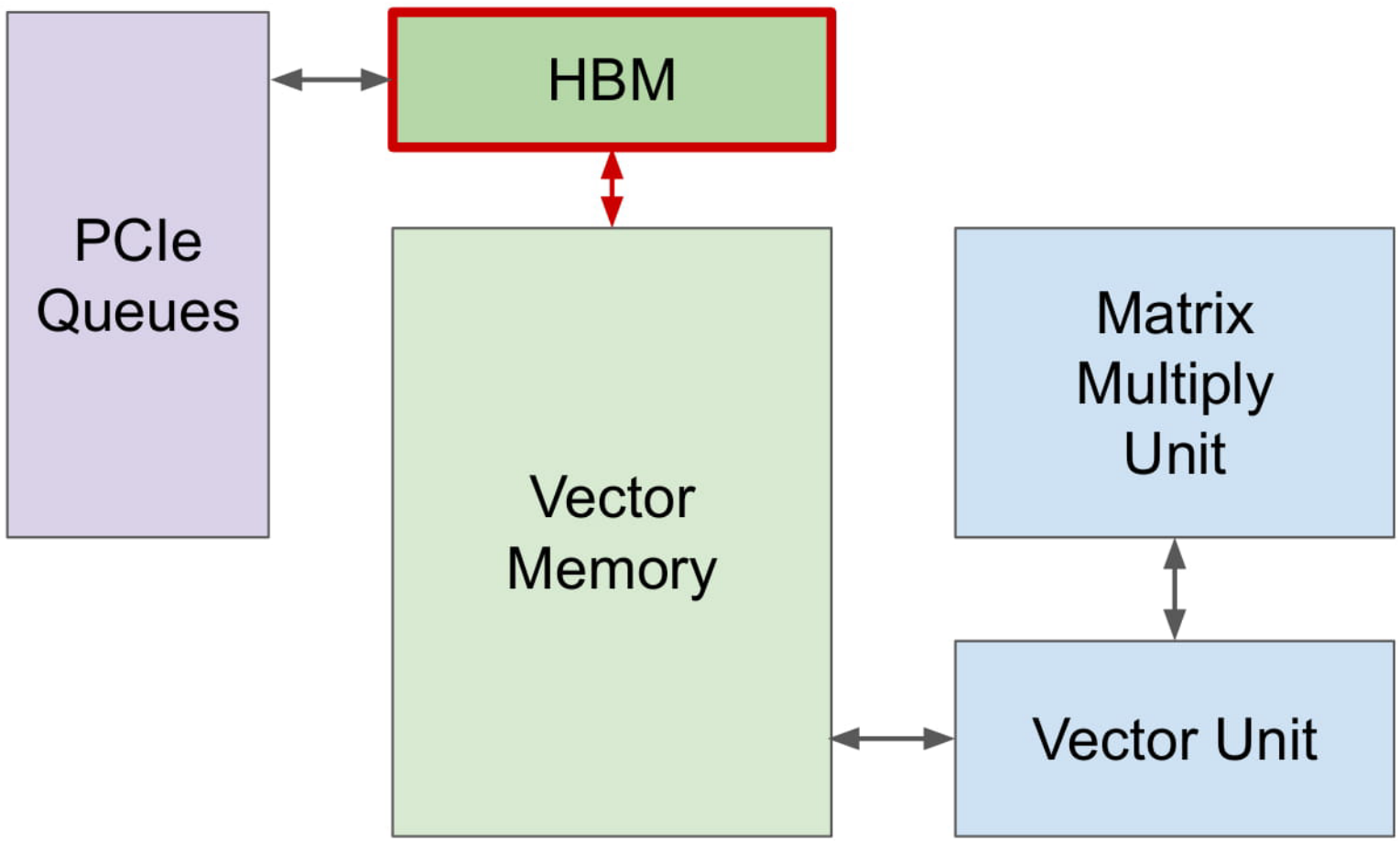
Connect matrix unit as an offload for the vector unit

TPUv2 Changes



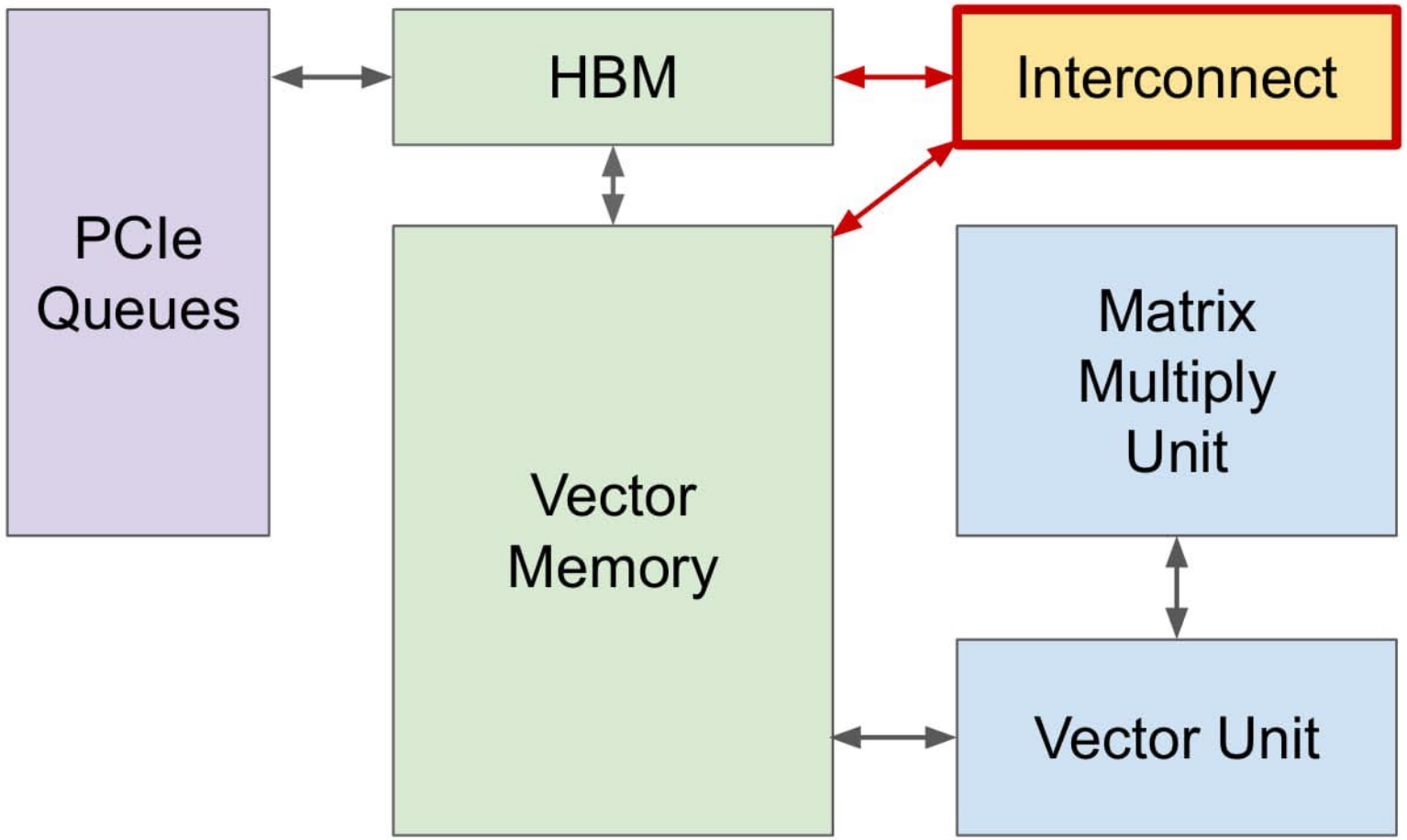
Connect DRAM into the memory system instead of directly into the matrix unit

TPUv2 Changes

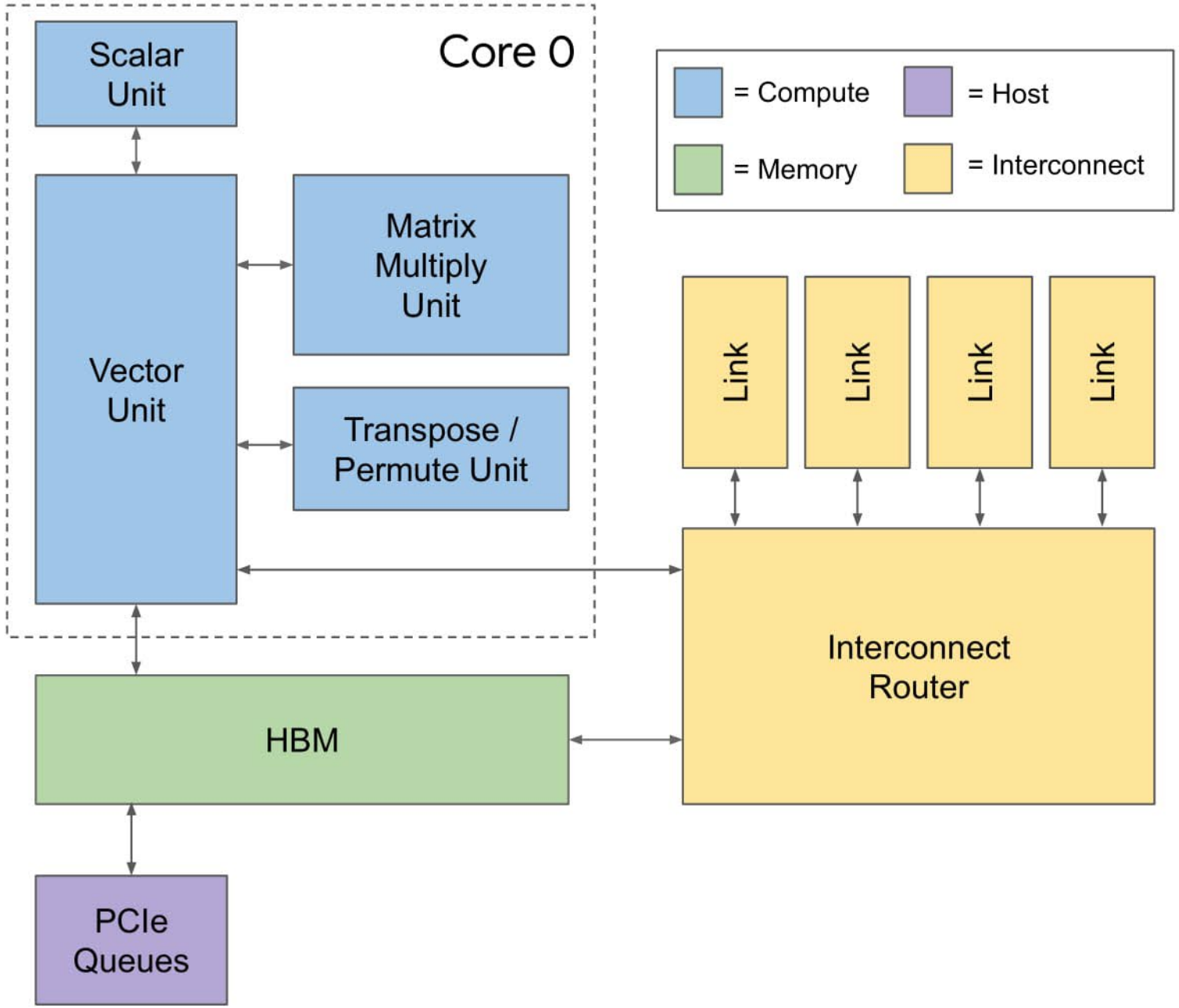


Move to HBM for bandwidth

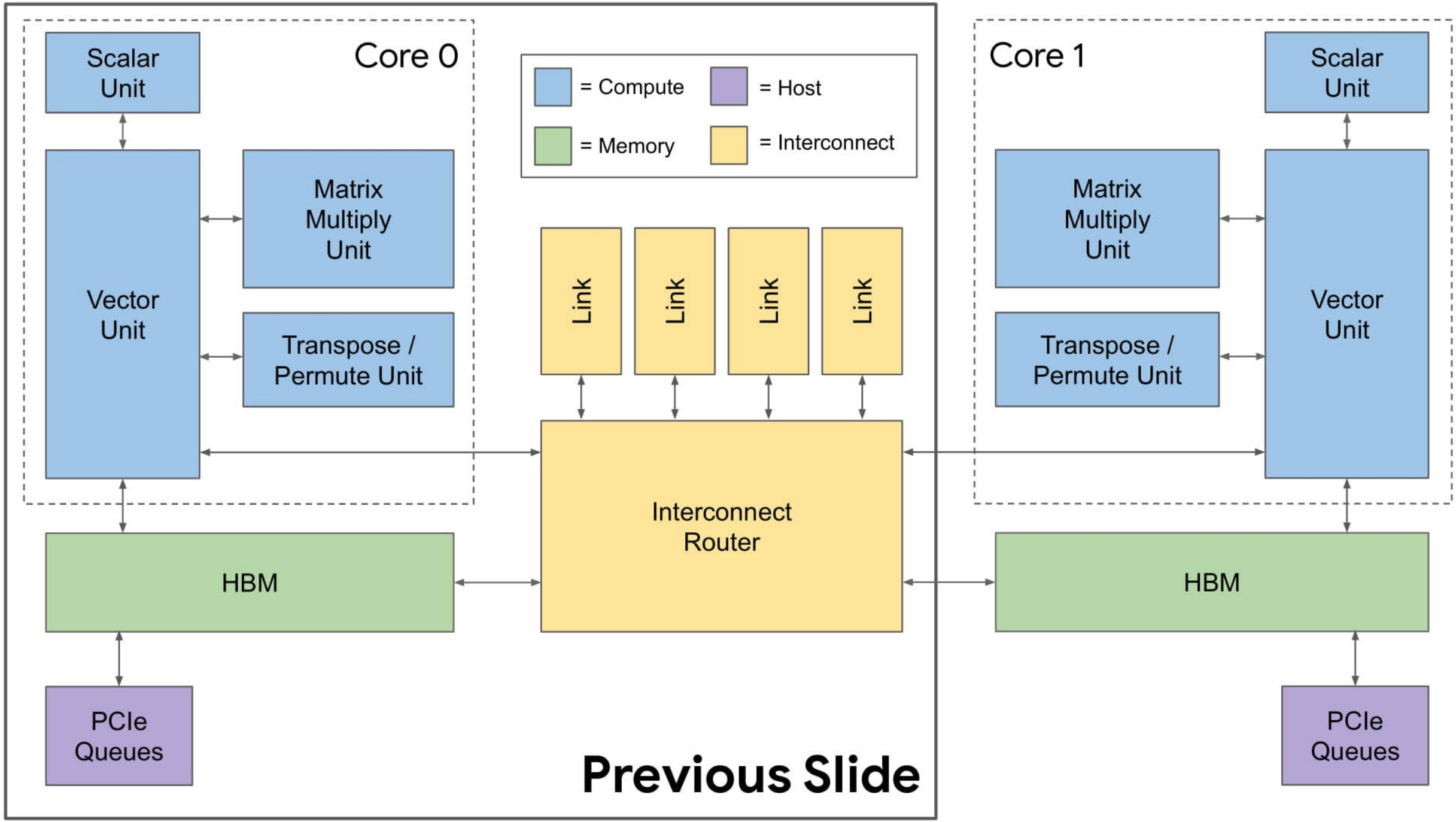
TPUv2 Changes



Add interconnect for high-bandwidth scaling

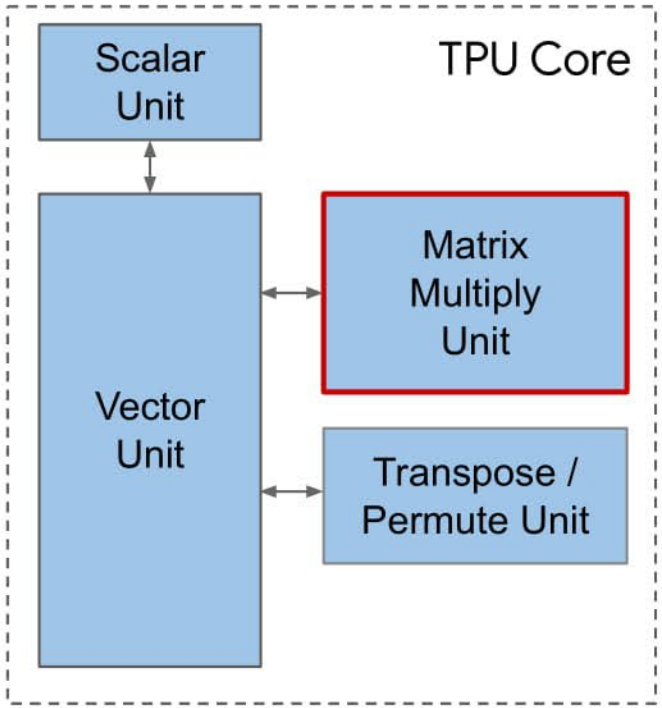


Redrawn with more detail...

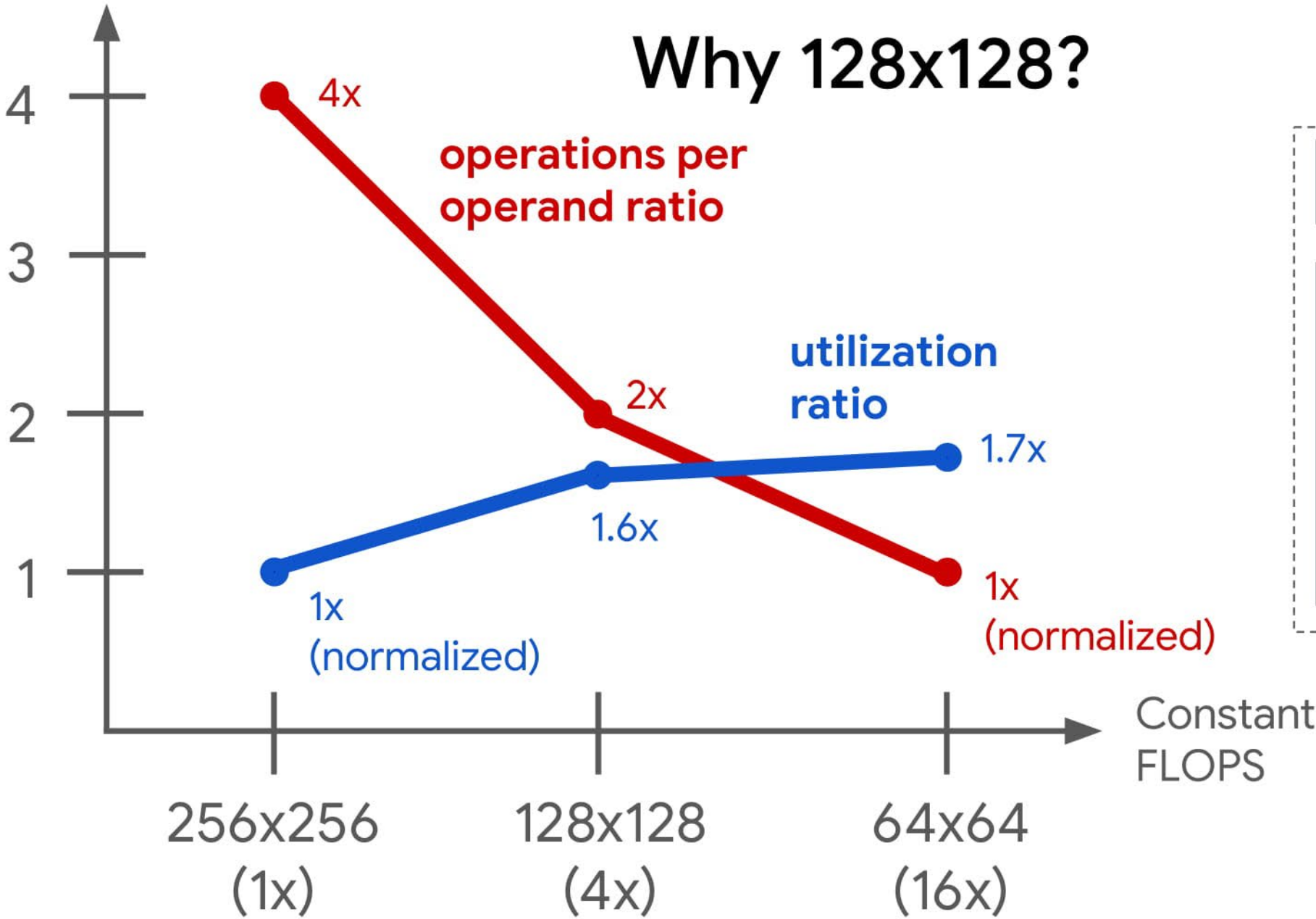

Core 0
Scalar Unit
Vector Unit
Matrix Multiply Unit
Transpose / Permute Unit
= Compute
= Host
= Memory
= Interconnect
Link
Link
Link
Link
Interconnect Router
HBM
PCIe Queues
Core 1
Scalar Unit
Matrix Multiply Unit
Transpose / Permute Unit
Vector Unit
HBM
PCIe Queues
Previous Slide

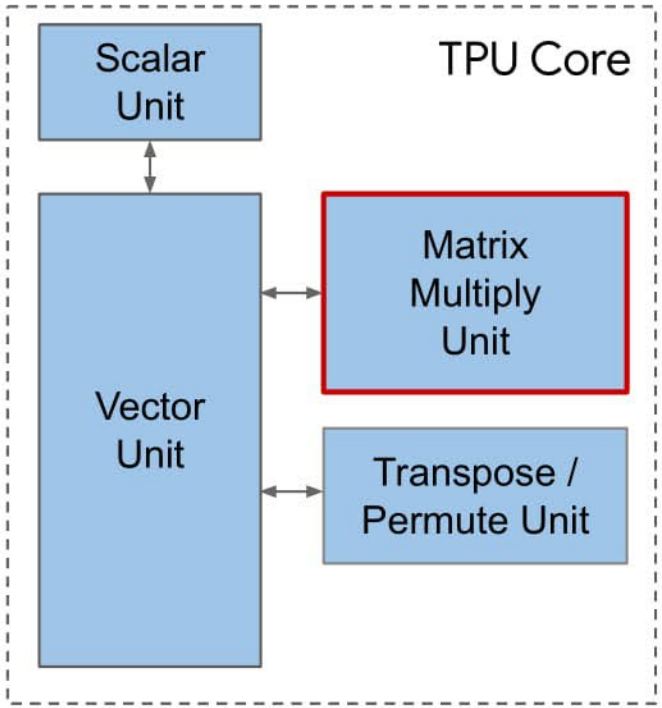
TPU Core: Matrix Multiply Unit

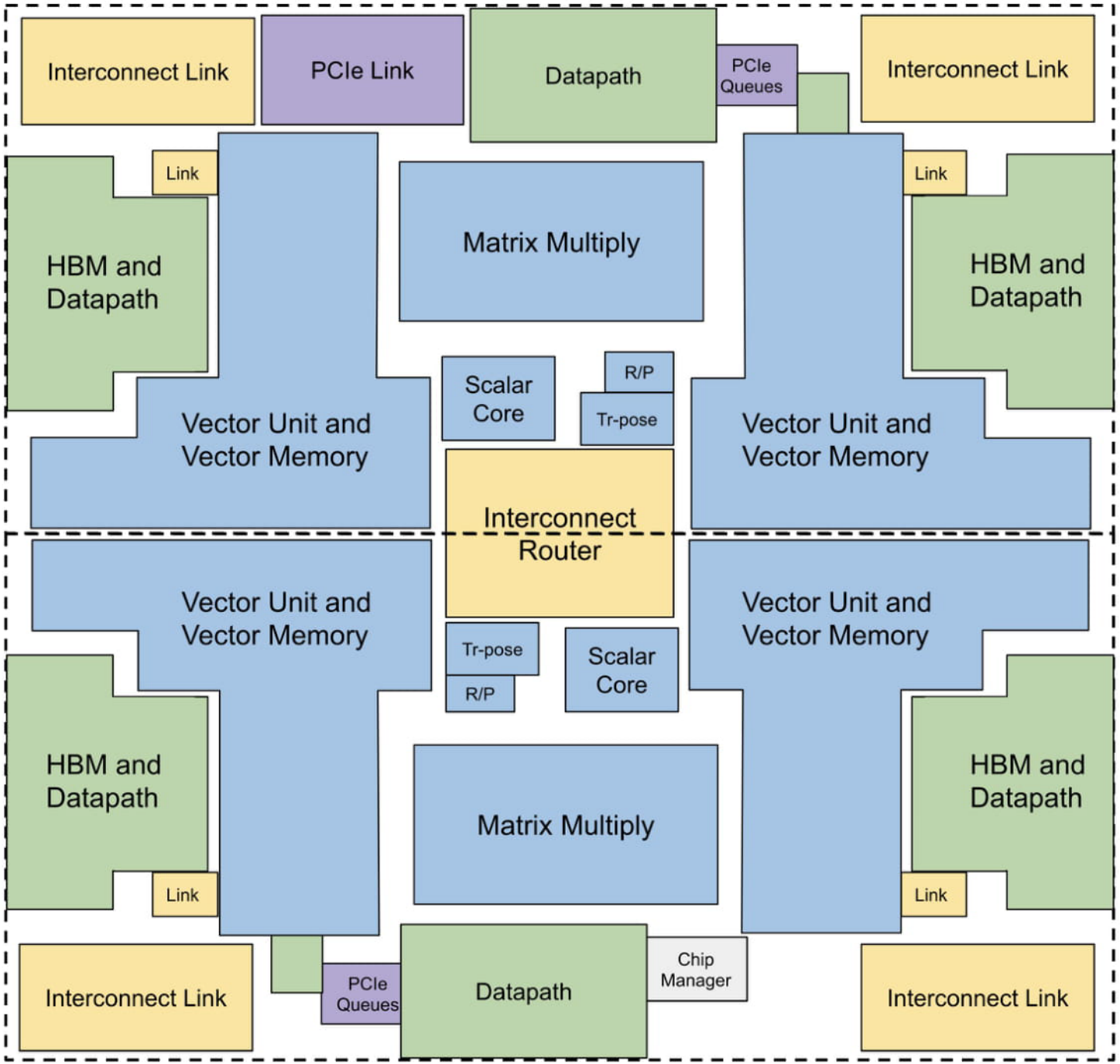
- 128 x 128 systolic array
  - Streaming LHS and results
  - Stationary RHS (w/ optional transpose)
- Numerics
  - bfloat16 multiply
    - {s, e, m} = {1, 8, 7}
    - The original!
  - float32 accumulation



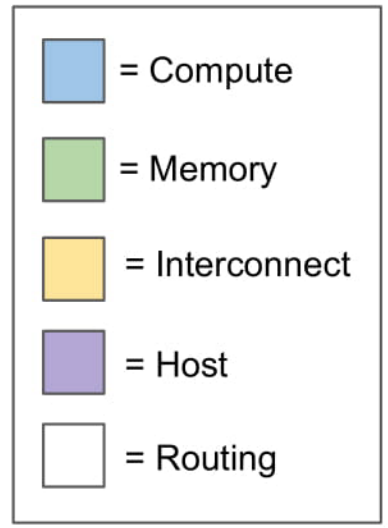
Why 128x128?


4
3
2
1
4x
operations per
operand ratio
2x
1x
(normalized)
utilization
ratio
1.7x
1.6x
1x
(normalized)
Constant
FLOPS
256x256
(1x)
128x128
(4x)
64x64
(16x)


TPU Core
Scalar
Unit
Vector
Unit
Matrix
Multiply
Unit
Transpose /
Permute Unit

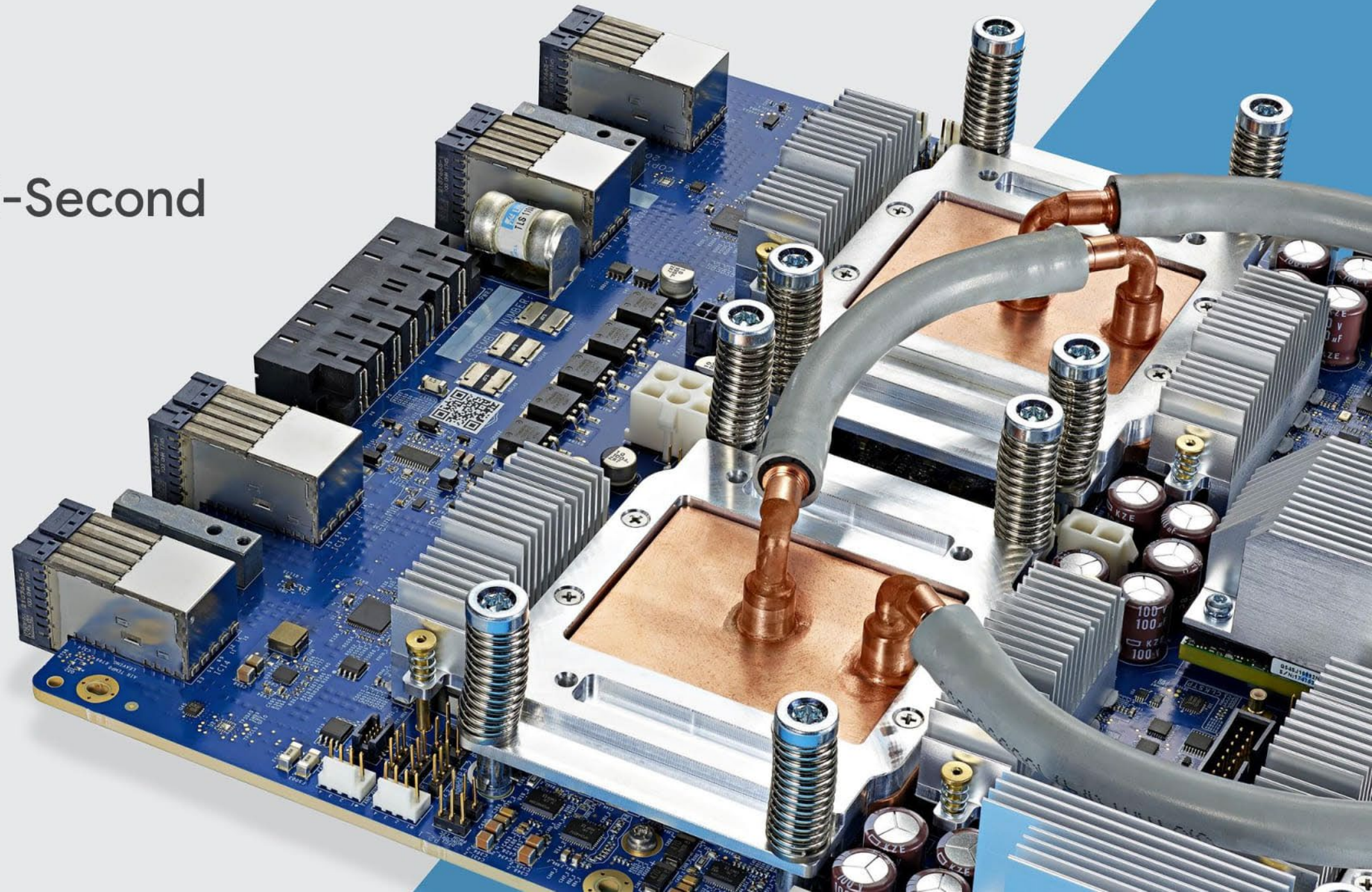

Interconnect Link
PCIe Link
Datapath
PCIe Queues
Interconnect Link
Link
HBM and Datapath
Matrix Multiply
Link
HBM and Datapath
Vector Unit and Vector Memory
Scalar Core
R/P
Tr-pose
Vector Unit and Vector Memory
Interconnect Router
Vector Unit and Vector Memory
Tr-pose
R/P
Scalar Core
Vector Unit and Vector Memory
HBM and Datapath
Matrix Multiply
Link
HBM and Datapath
Link
Interconnect Link
PCIe Queues
Datapath
Chip Manager
Interconnect Link

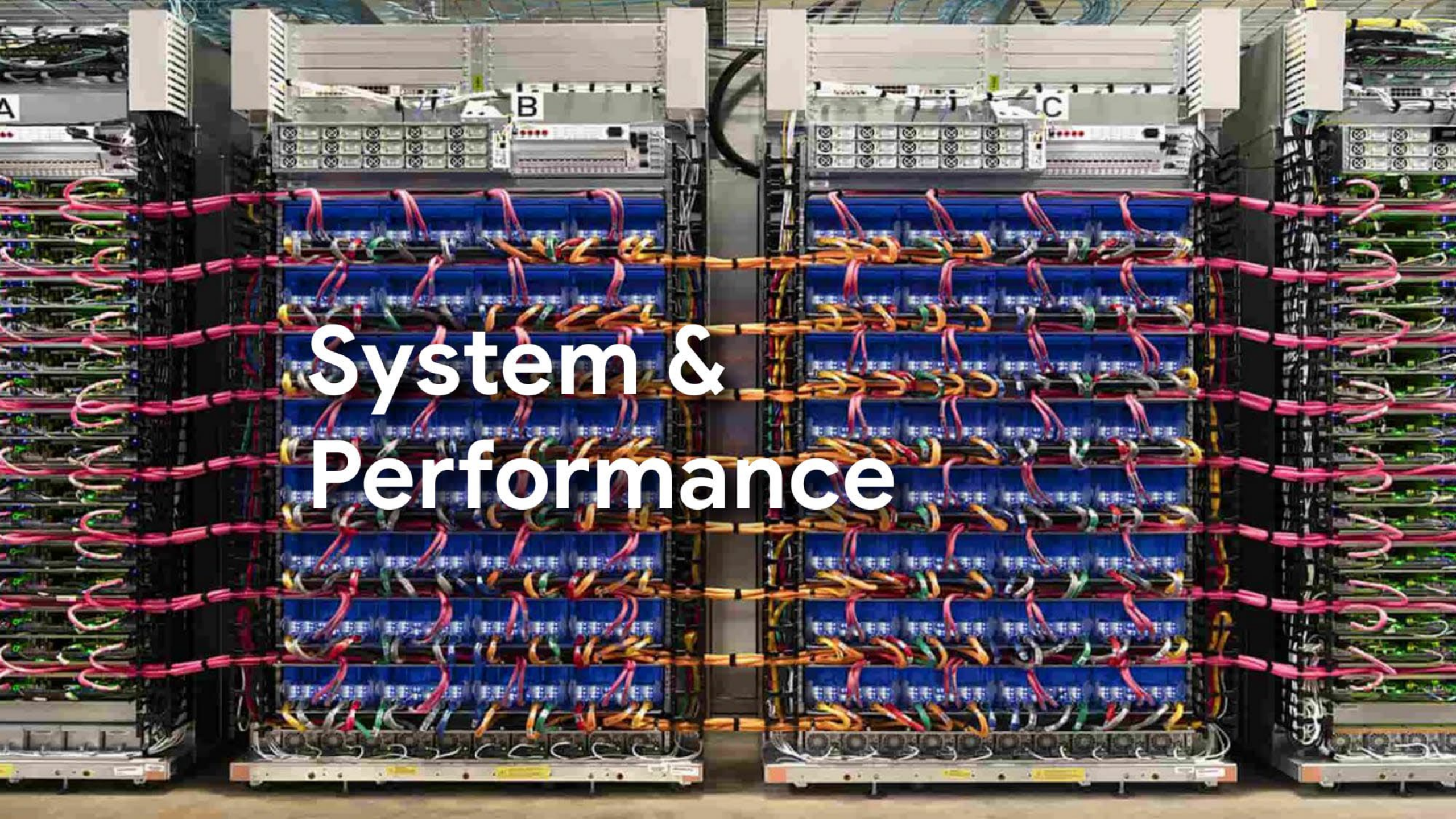
Floorplan


= Compute
= Memory
= Interconnect
= Host
= Routing

TPUv3

The Anti-Second System





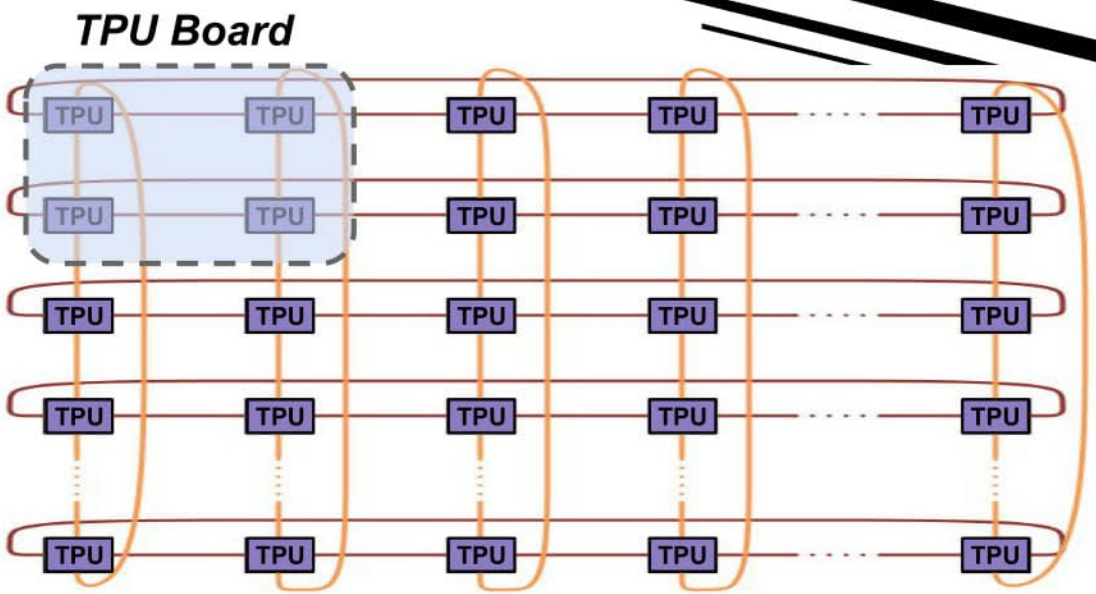
System & Performance

Supercomputer with dedicated interconnect

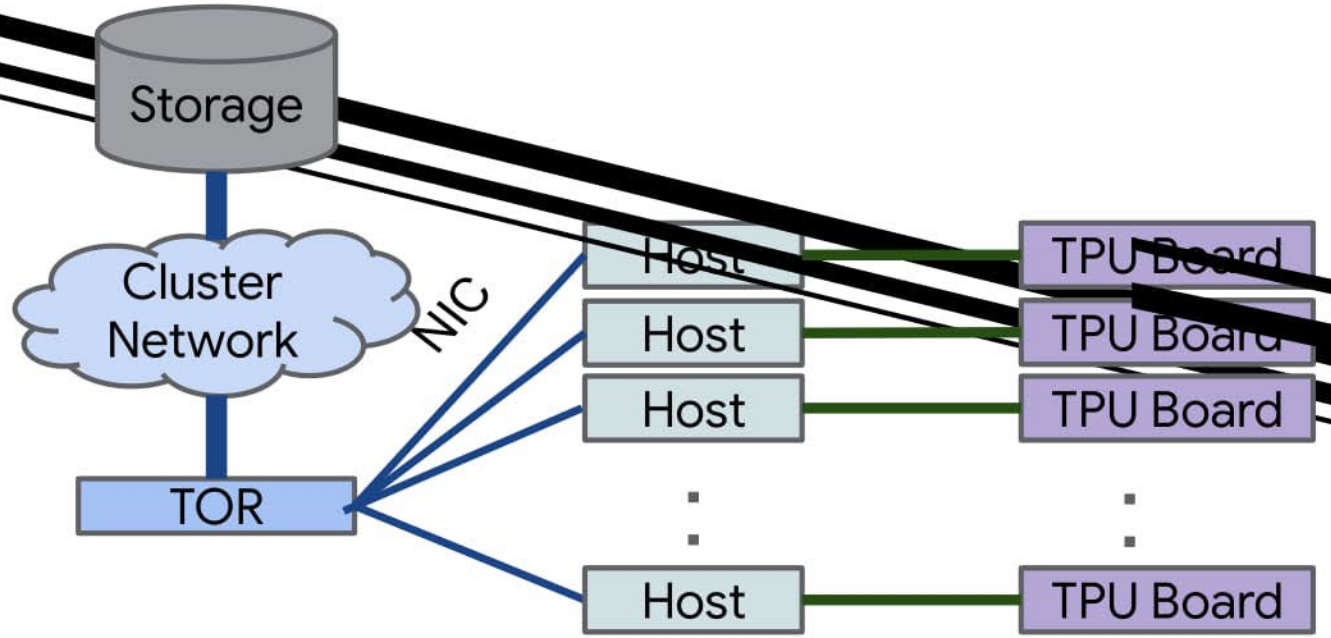
- TPUv1: single-chip system—built as **coprocessor** to a CPU
  - Works well for inference

- TPUv2, v3: ML **Supercomputer**
  - Multi-chip scaling critical for practical training times
    - Single TPUv2 chip would take 60 - 400 days for production workloads

TPU Training Pod Architecture



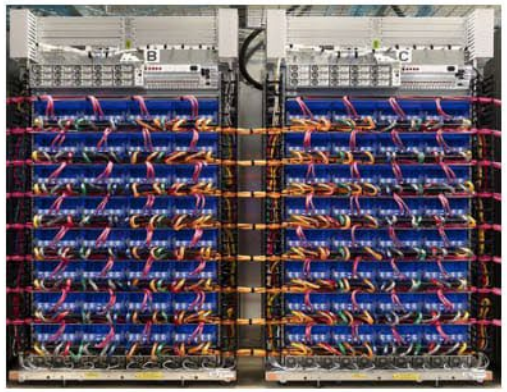
TPUs interconnected in 2D Torus



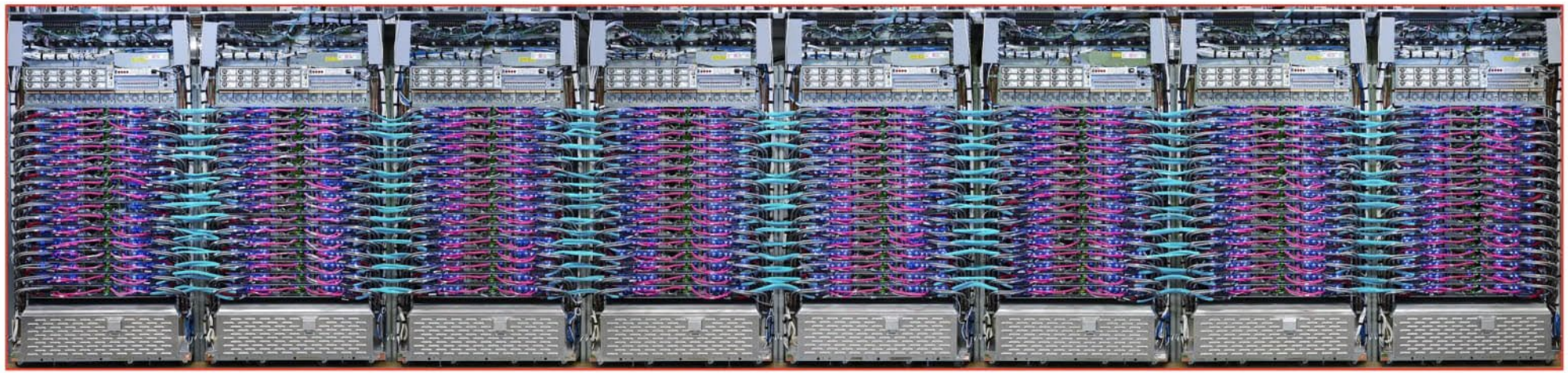
Dedicated network for
synchronous parallel training

Supercomputer with dedicated interconnect

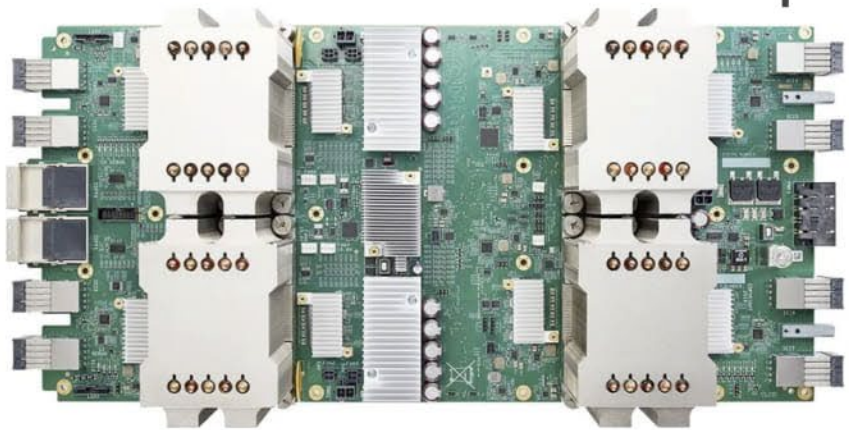
TPUv2 supercomputer (256 chips)



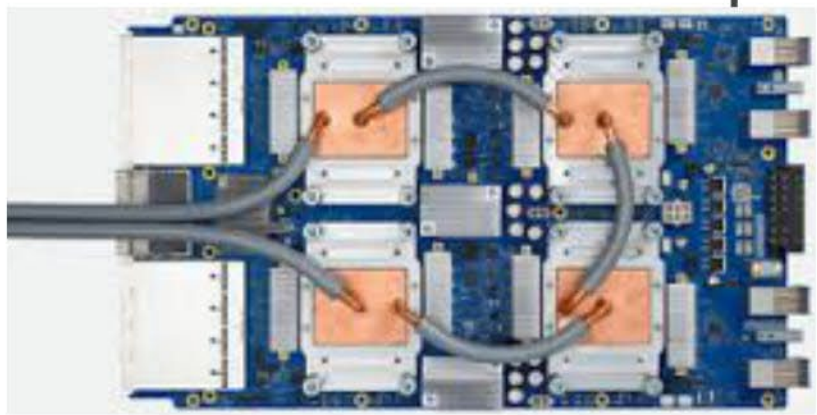
TPUv3 supercomputer (1024 chips)



TPUv2 boards = 4 chips

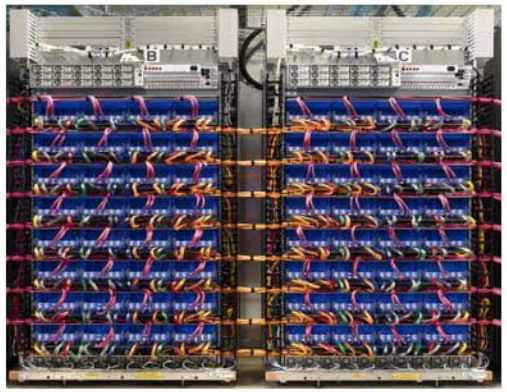


TPUv3 boards = 4 chips



Supercomputer with dedicated interconnect

TPUv2 supercomputer (256 chips)



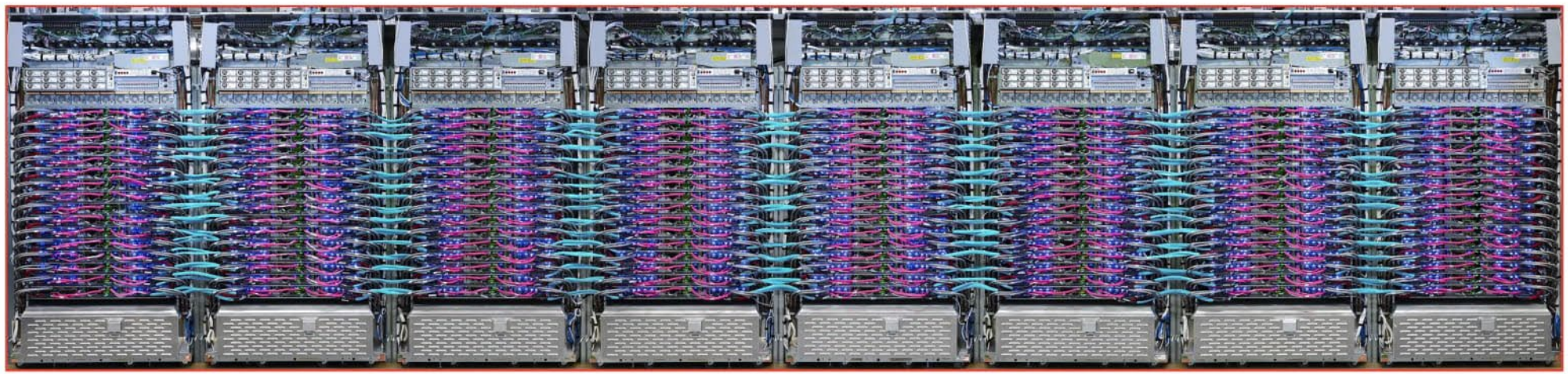
11.5 petaflops

4 TB HBM

2-D torus

256 chips

TPUv3 supercomputer (1024 chips)



> 100 petaflops

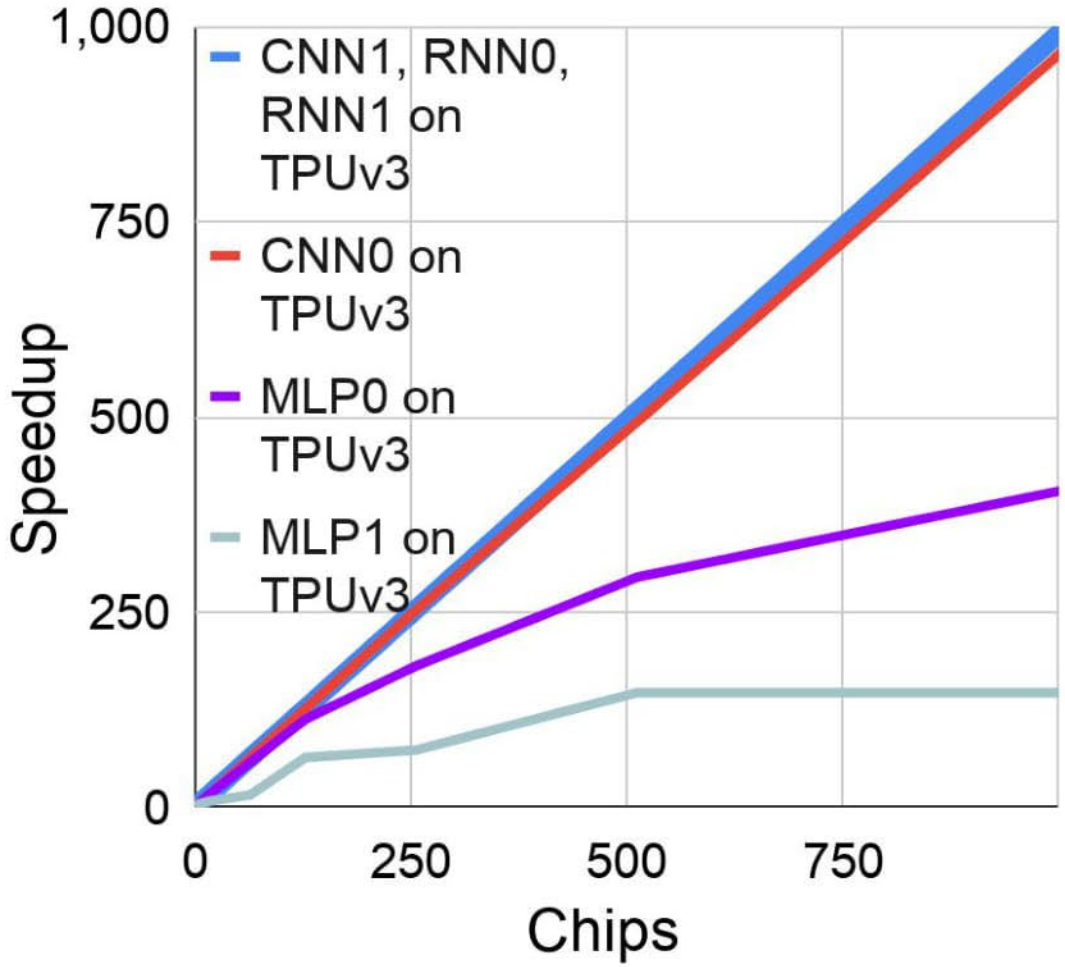
32 TB HBM

Liquid cooled

New chip + larger-scale system

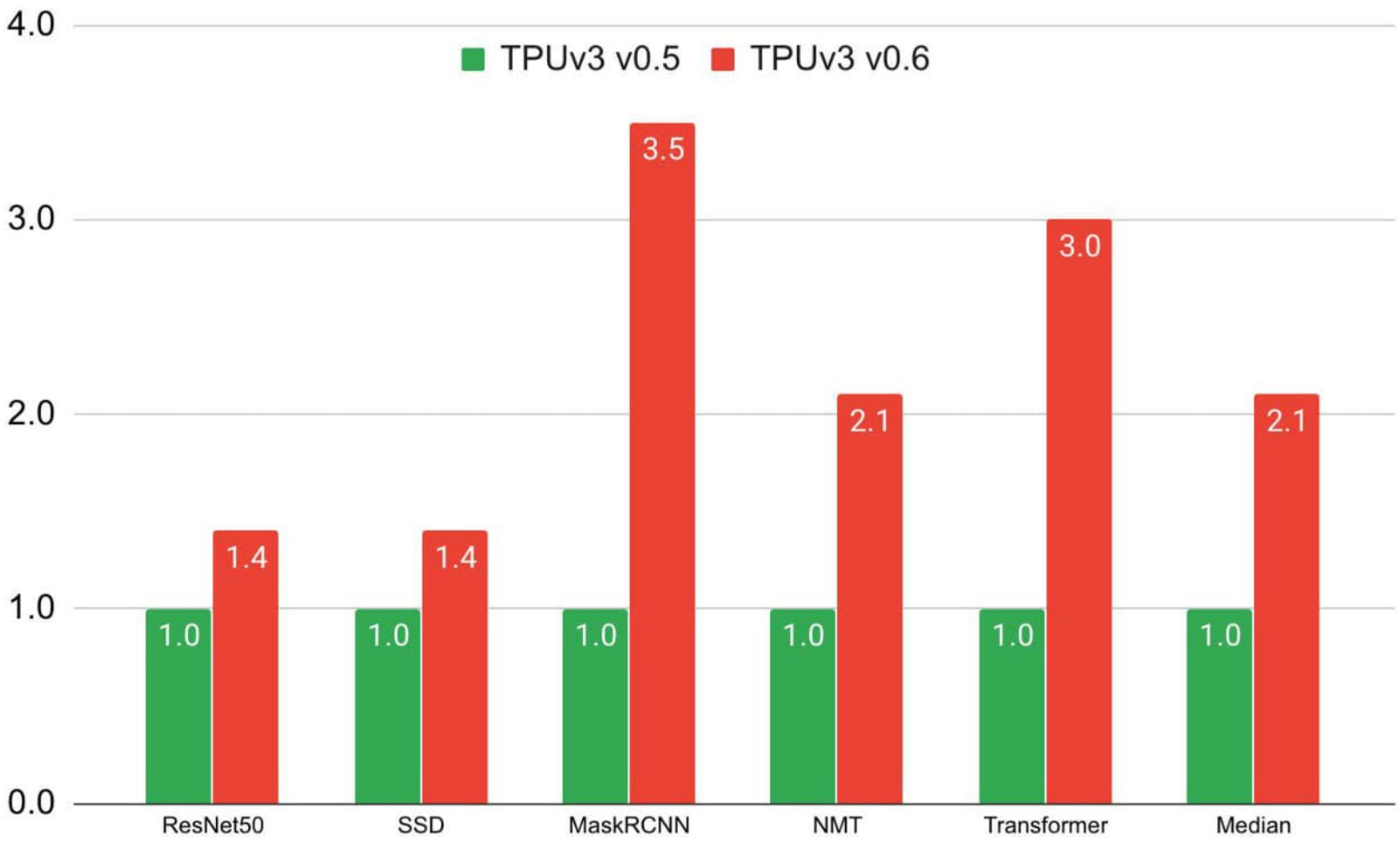
1024 chips

TPUv3 Supercomputer Scaling: 6 Production Apps



- **MLP0** & **MLP1**
  - 40% & 14% of perfect linear scaling
- **CNN0**
  - 96% of perfect linear scaling!
- **CNN1, RNN0, RNN1**
  - **3 production apps** run at **99%** of perfect linear scaling at 1024 chips!

Speedup: MLPerf v0.5 (11/2018) - v0.6 (5/2019)



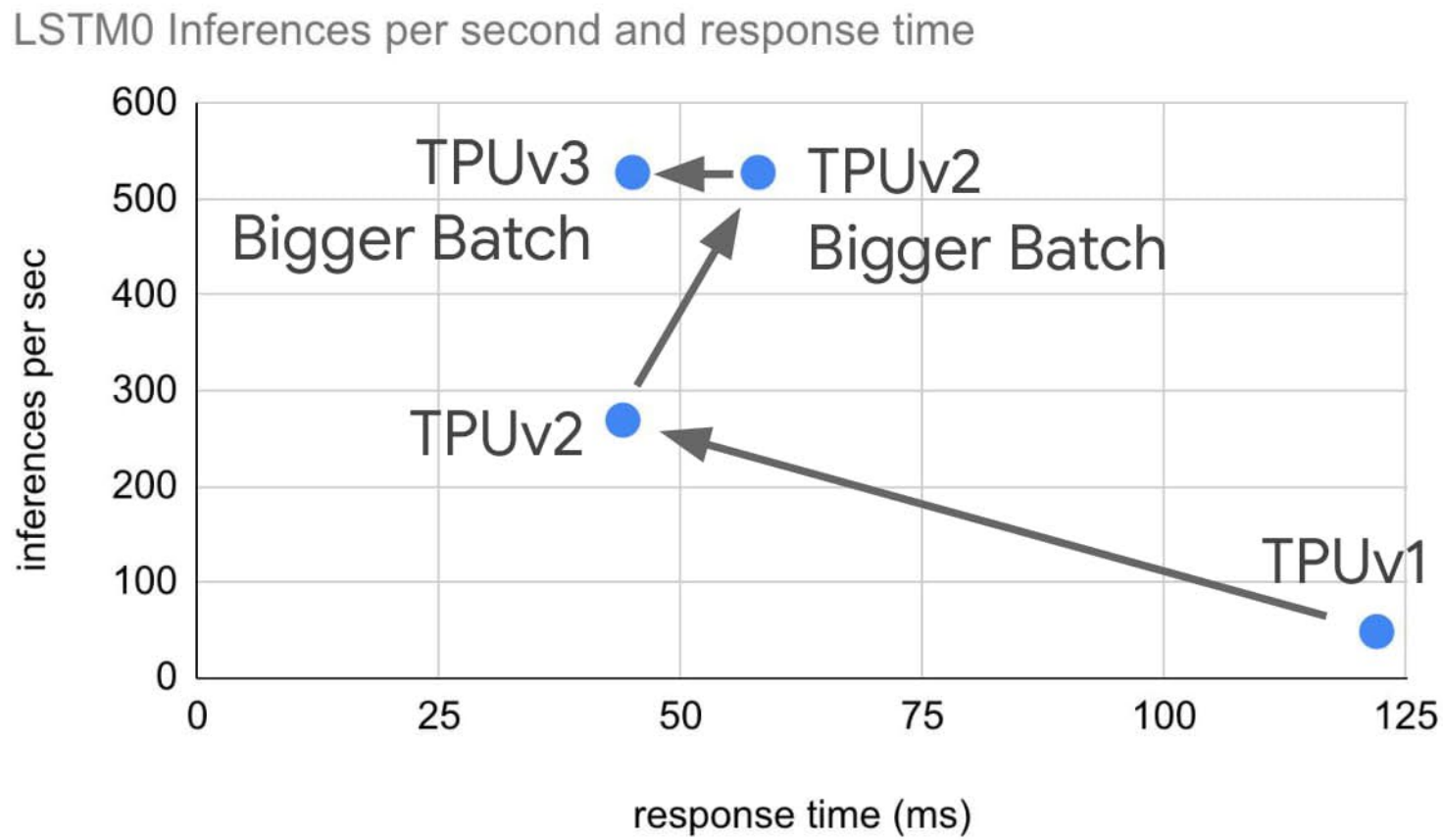
Production apps also sped up:

- CNN0 1.8x (more bfloat16 use)
- MLP0 1.6x (better partitioning and placement of embeddings)

Performance enables larger models for improved accuracy

Inference: TPUv2/v3 vs TPUv1

- Training chips can also do inference (looks like forward pass)
- Bfloat16 numerics in TPUv2/v3 vs int8 in TPUv1



Tesla FSD hardware 3.0

Tesla Trains One of the Largest Neural Networks

- https://www.youtube.com/watch?feature=youtu.be&v=oBklltKXtDE&app=desktop

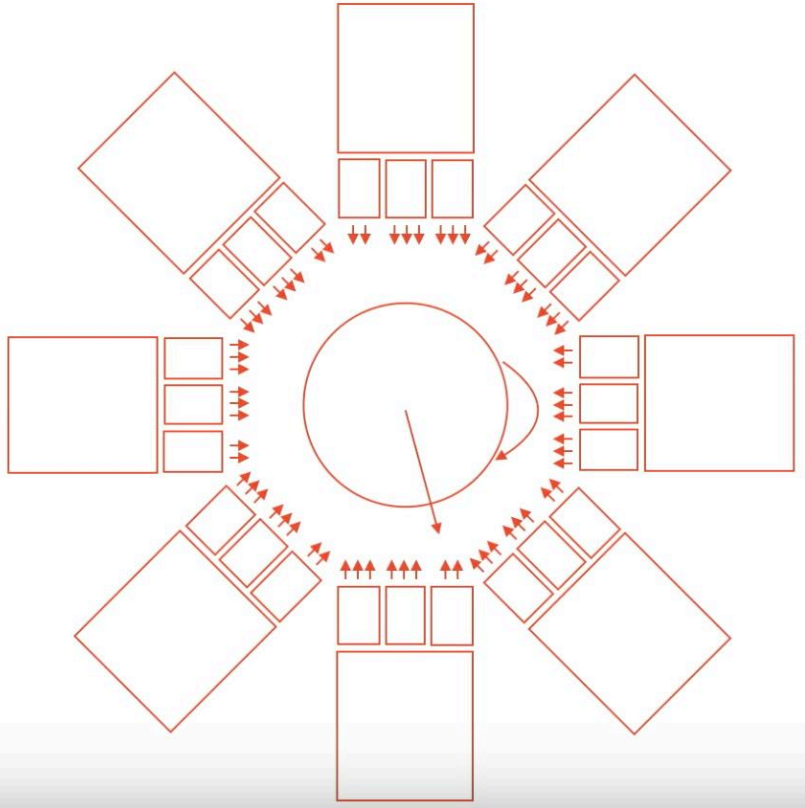
[Tesla, 2019]

Tesla HydraNet

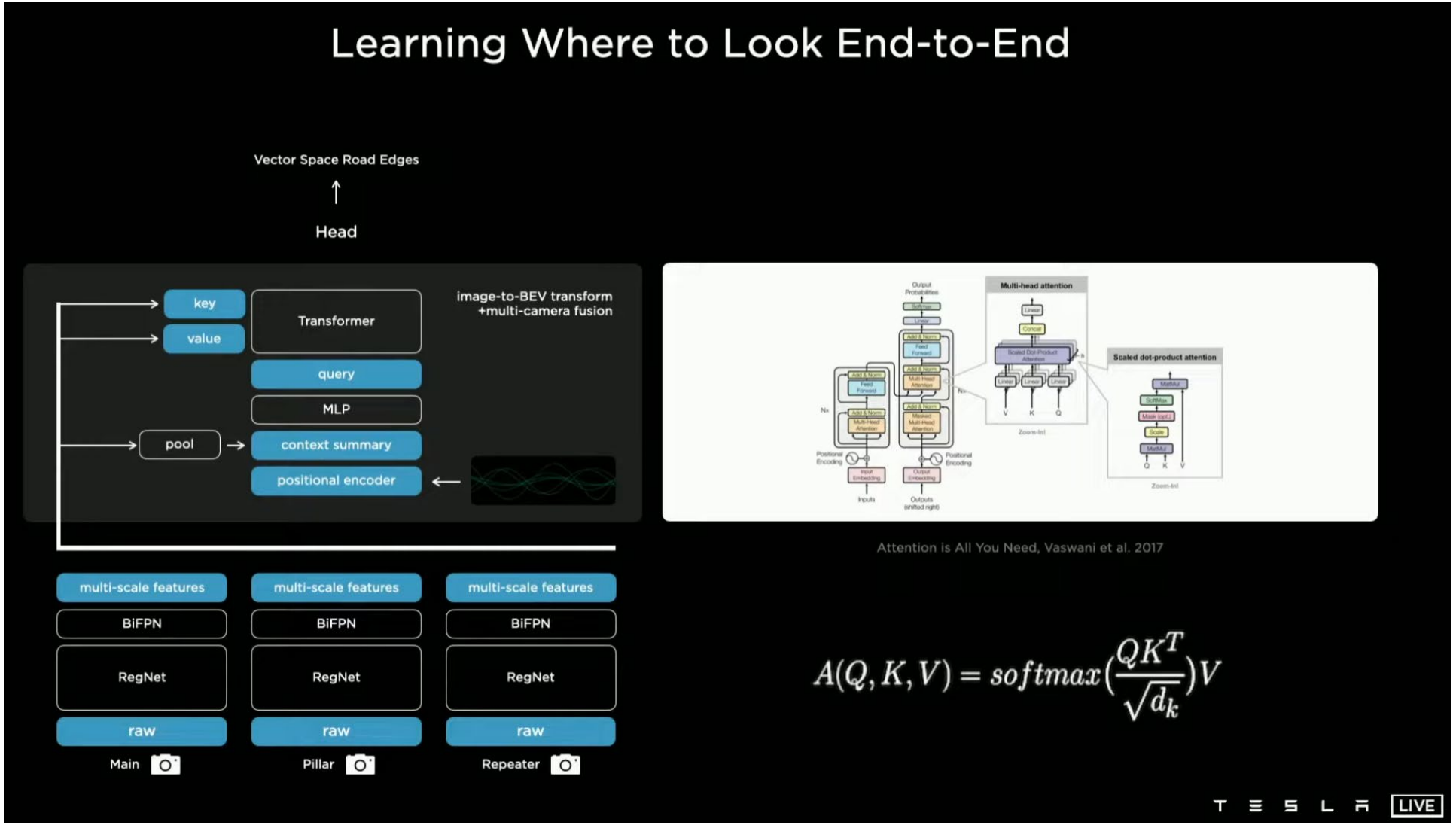
- One of the world-largest CNN models
- Processing 4,096 HD images /second

e.g.:
8 cameras, 16 time steps, batch size 32:
8 x 16 x 32 = **4096 images**
in a single forward pass

Requires a combination of data parallel and model parallel training

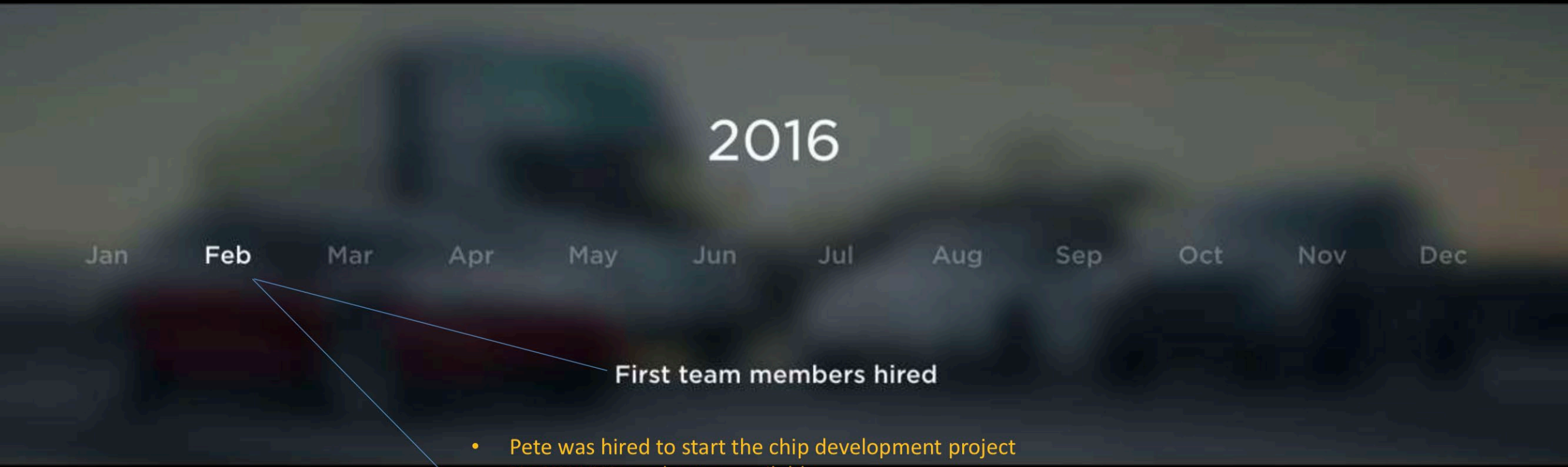


Tesla HydraNet



[Tesla, 2021]

TIMELINE



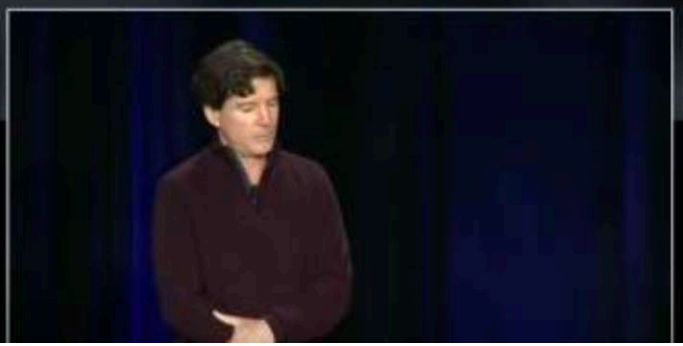
- Pete was hired to start the chip development project
- No NN HW accelerator available
- Custom design HW started by Pete & team in Tesla
- CPU and GPU IPs were licensed from other companies

TIMELINE

2018

Jan Feb Mar Apr May Jun **Jul** Aug Sep Oct Nov Dec

Production starts w/ Qualification



Aug 2017: released the design for manufacturing
Dec 2017: got the chip back; testing was done; made few changes
April 2018: Designed finished

TESLA LIVE

TIMELINE

2019

Jan Feb Mar **Apr** May Jun Jul Aug Sep Oct Nov Dec

Model 3 production start w/ new computer integrated in the car

TESLA LIVE

GOALS

Focus exclusively on Tesla requirements

<100W:
- Power, power supply, and thermal solution add cost
- Retrofit existing cars

Lower part cost to enable redundancy

At least 50 TOPS of neural network performance

Batch size of one

GPU compute for post processing

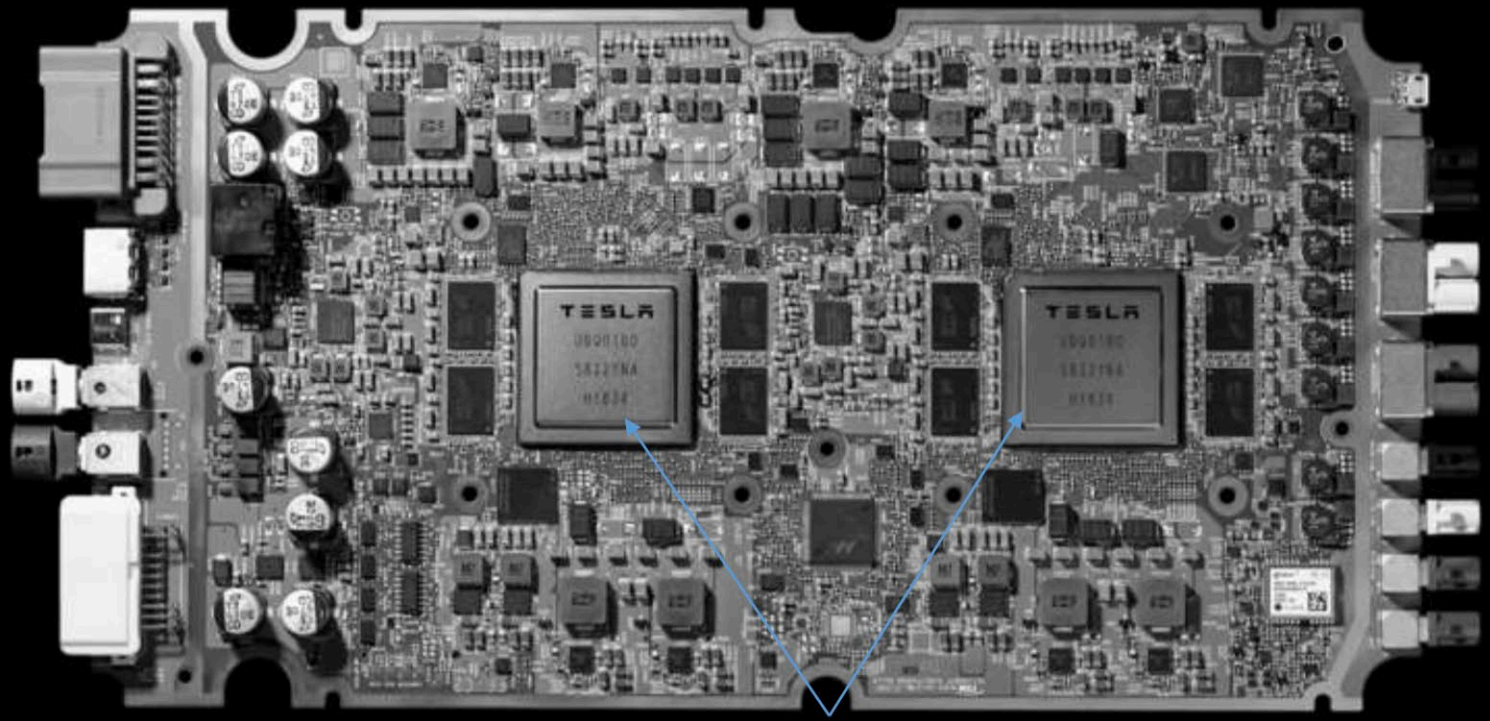
Security

Safety

TESLA LIVE

FULL SELF-DRIVING COMPUTER

Power supply and control

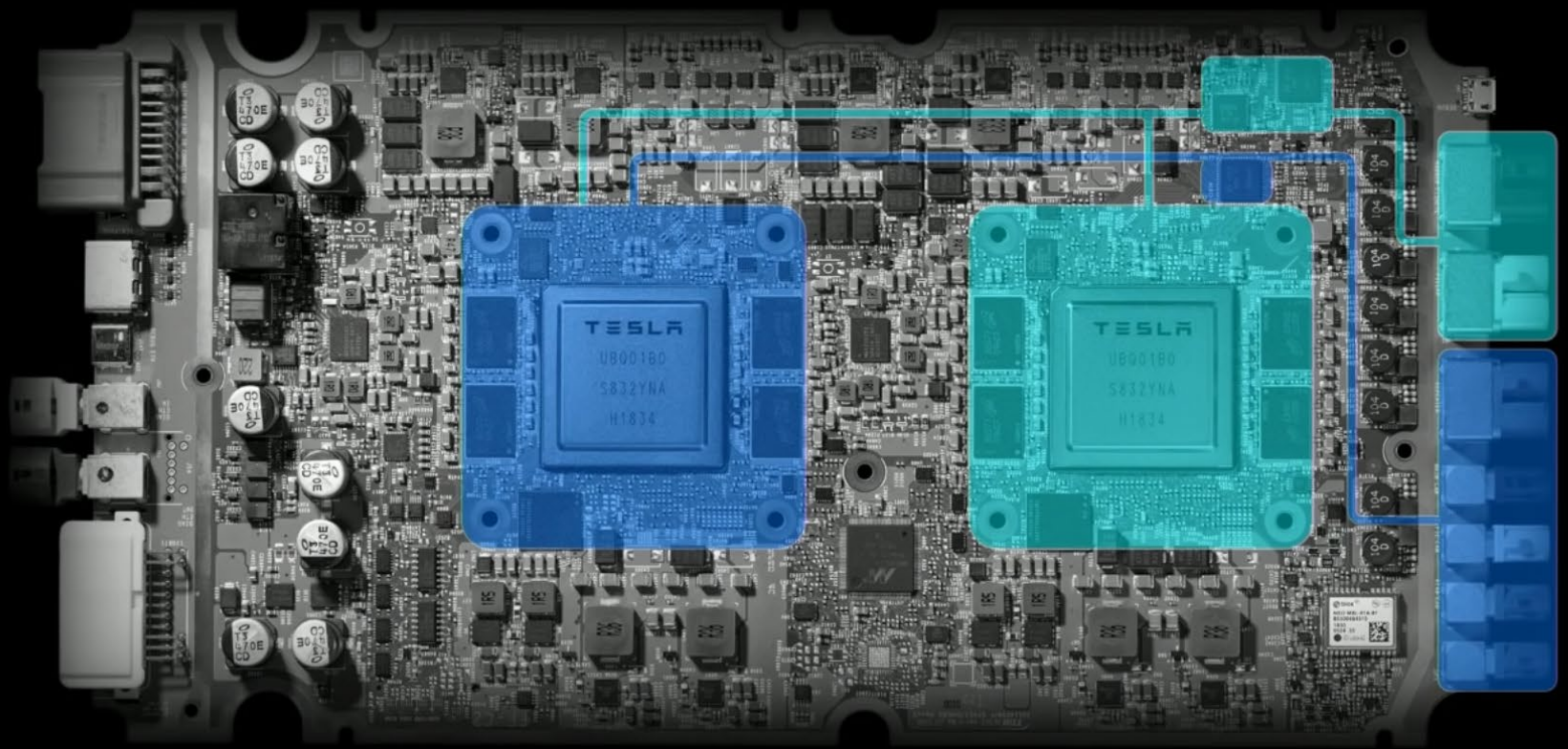


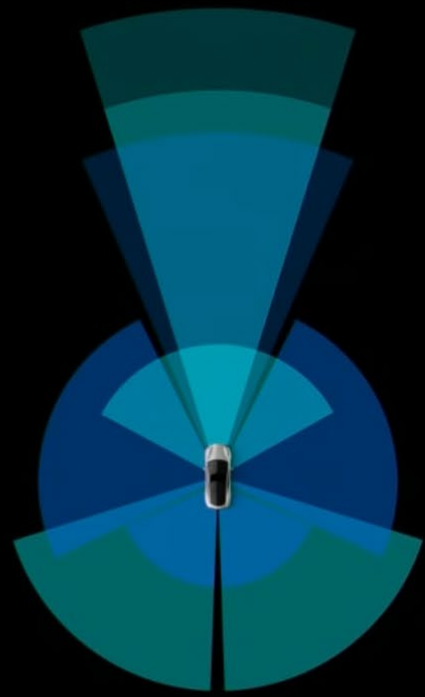
Video connection

Two independent computing chips (SoCs): calls FSD computer
[FSD: Full Self-Driving]

TESLA LIVE

PERCEIVE

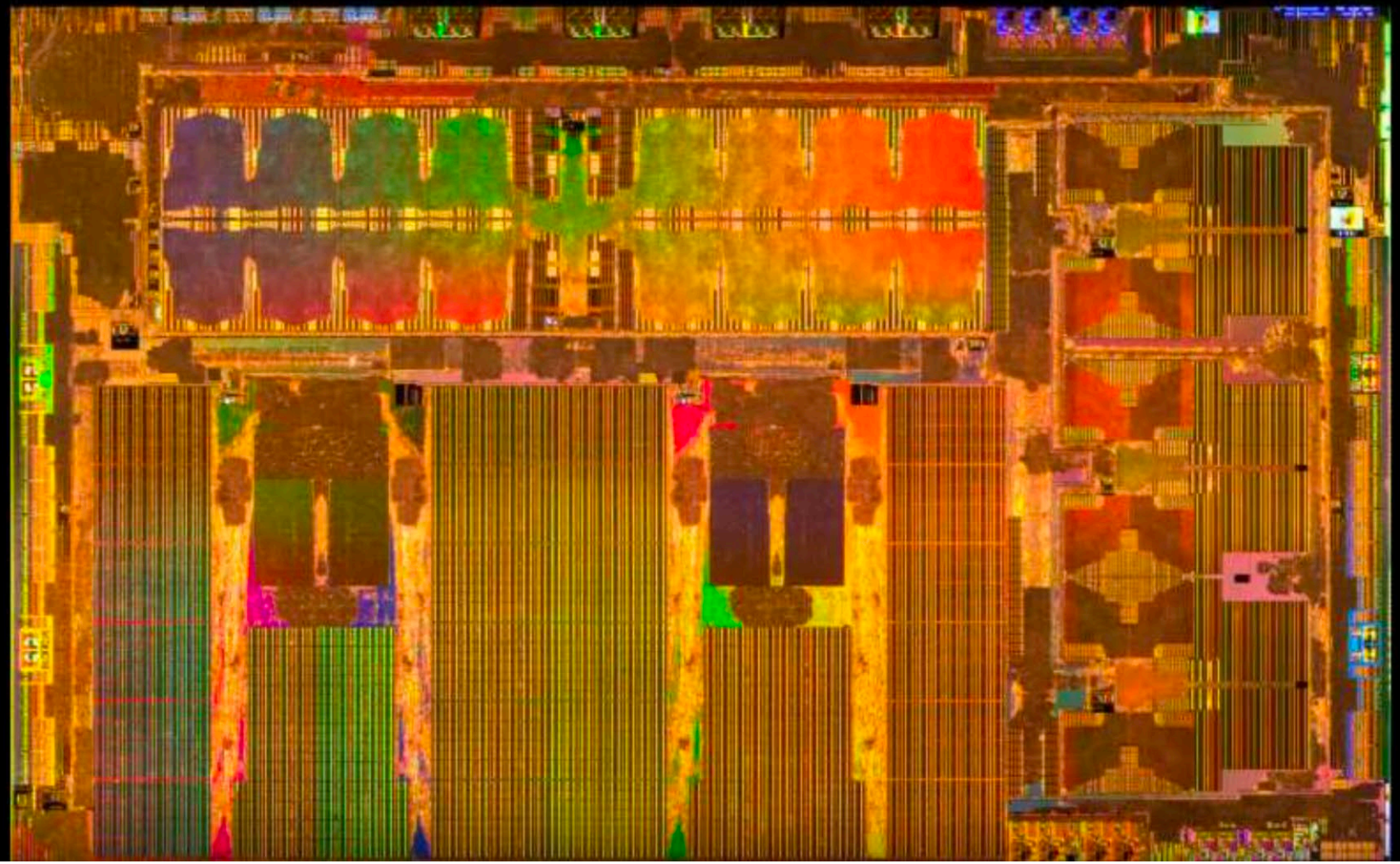




TESLA LIVE

Radar – GPS – Maps – IMU – Ultrasonic – Wheel Ticks – Steering Angle

FSD CHIP TOUR



14nm FinFET CMOS

260 mm²

250 million gates

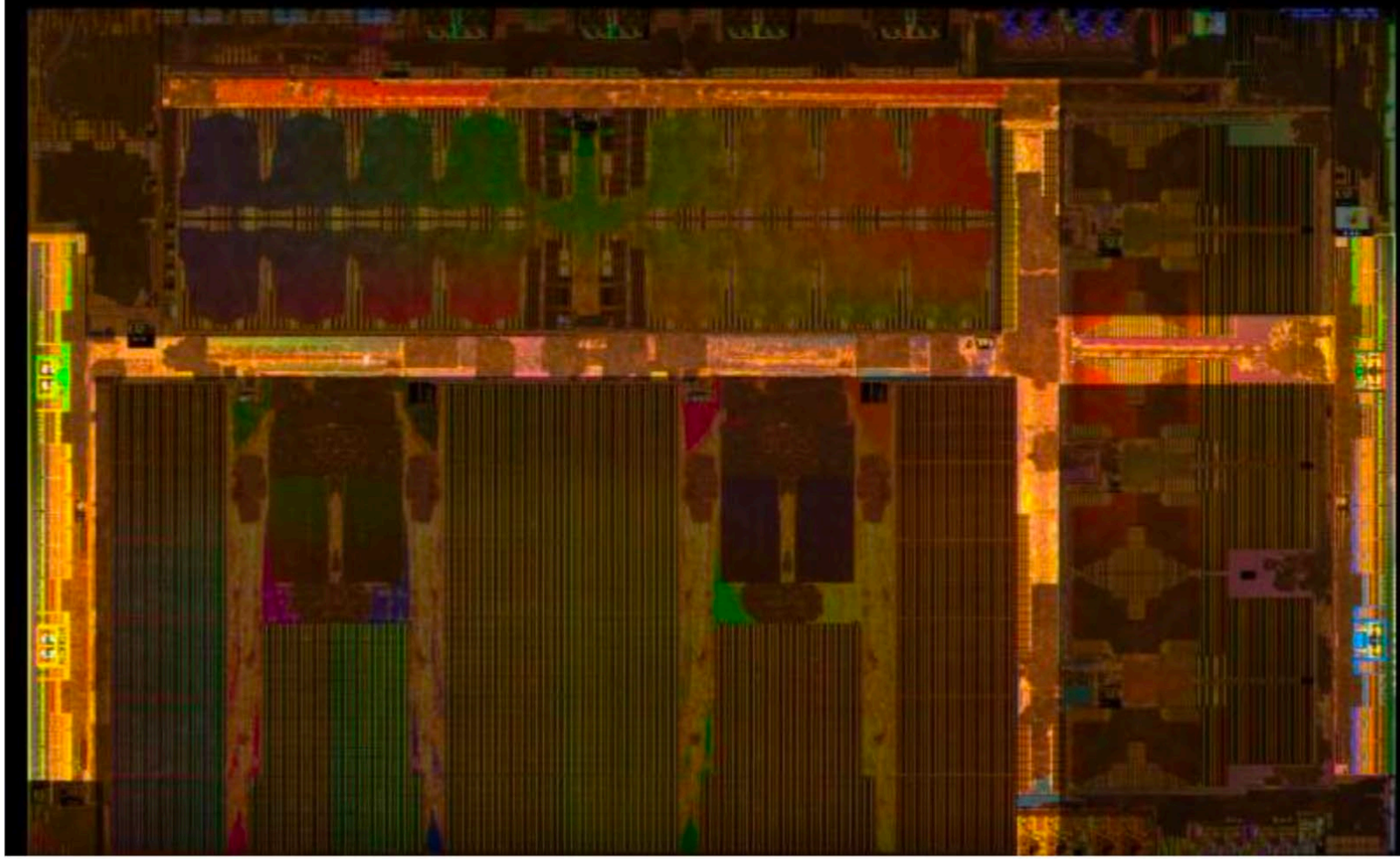
6 billion transistors

AEC Q100

[Typical cellphone chip: 100 mm2;
GPU chip: 600 – 800 mm2]

TESLA LIVE

LPDDR4 INTERFACE



On Chip Network

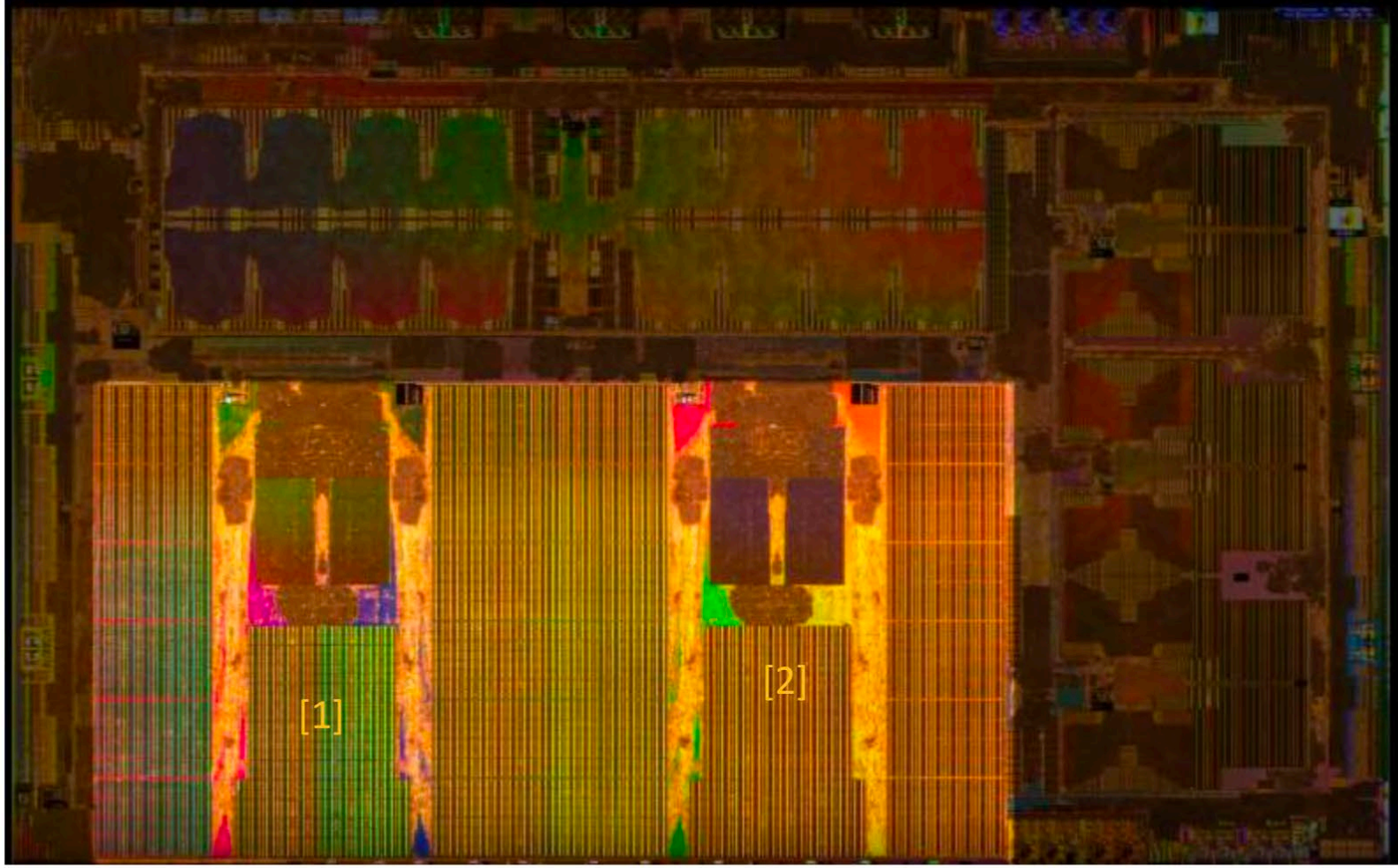
LPDDR4 DRAM

128b @ 4266 Gb/s

68 GB/s peak bandwidth

TESLA LIVE

NEURAL NETWORK PROCESSOR



32MB SRAM

96x96 Mul/Add array

ReLU hardware

Pooling hardware

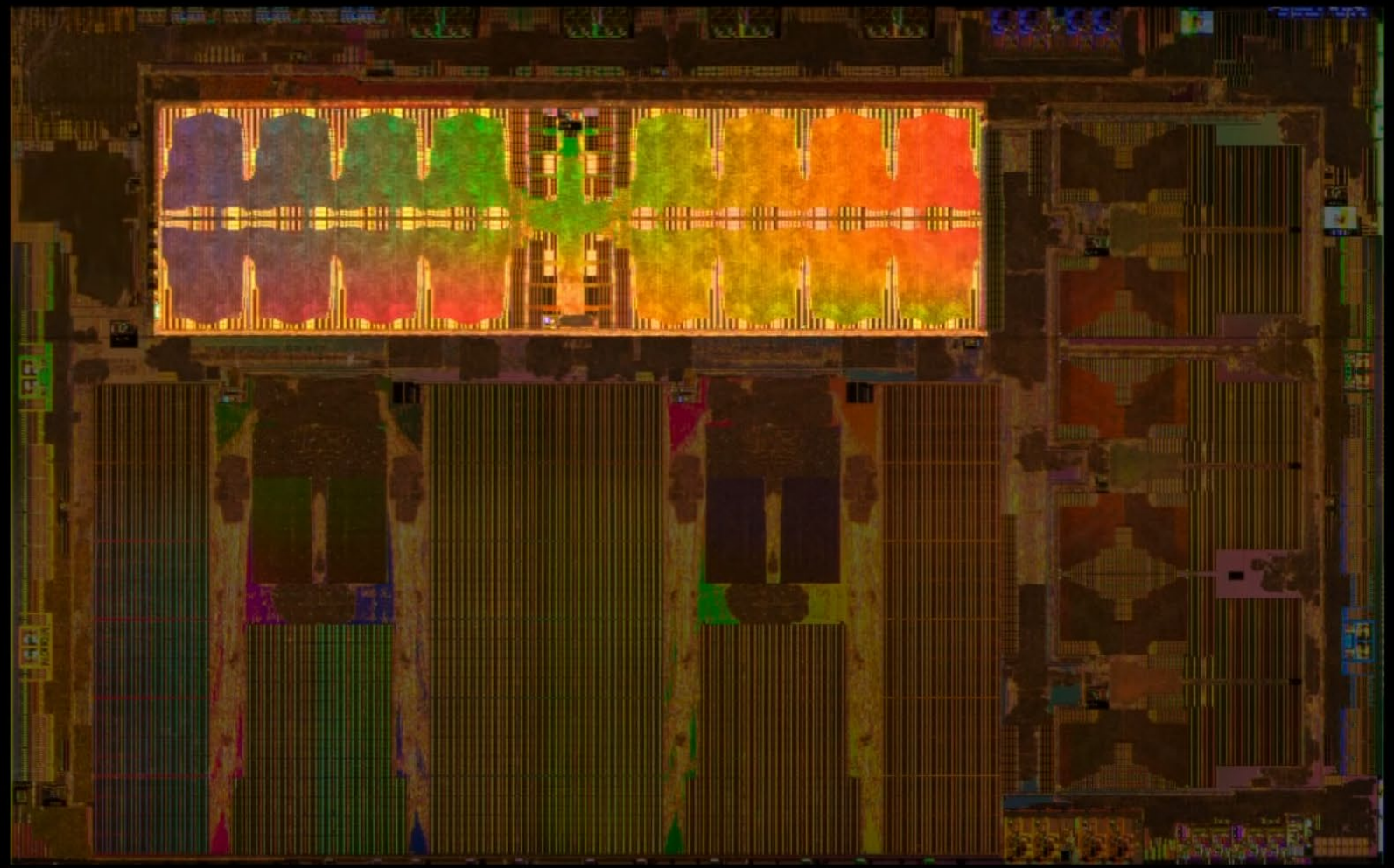
Separate dedicated HW

36 TOPS @ 2 GHz

2 per chip, 72 TOPS total

TESLA LIVE

GPU



1 Ghz

600 GFLOPS

FP32

FP16

TESLA LIVE

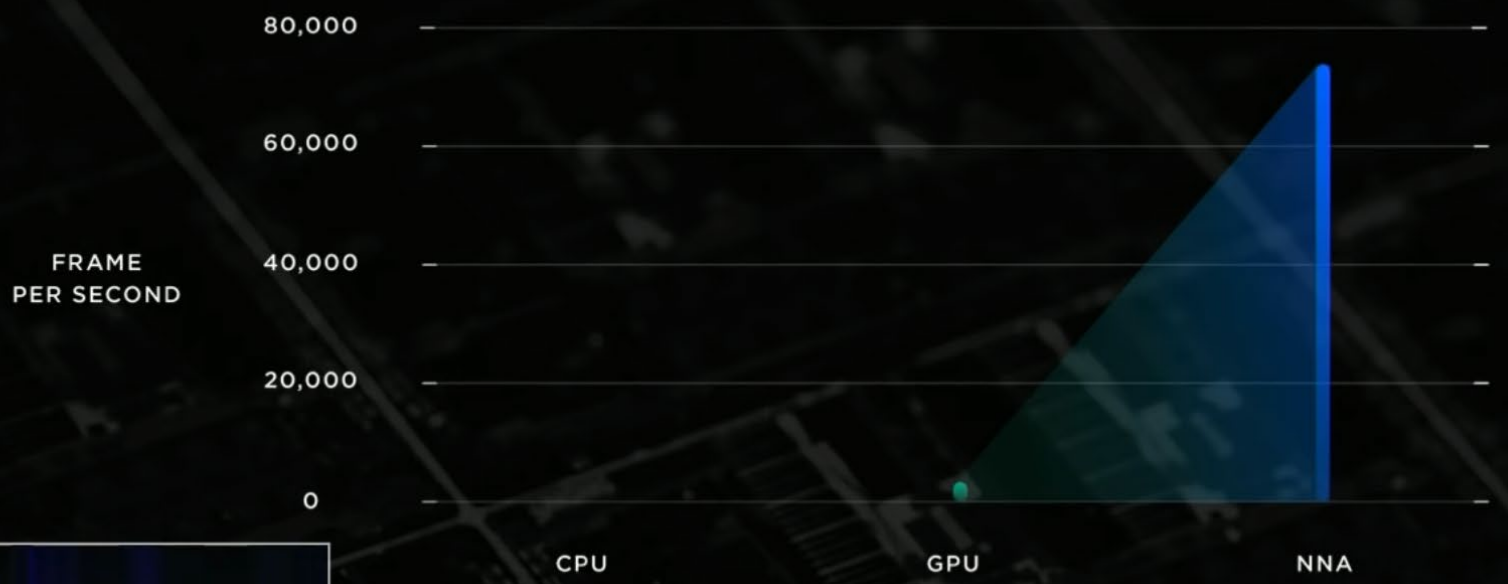
MAIN PROCESSOR

12 ARM A72 64b CPUs

2.2 GHz

TESLA LIVE

PERSPECTIVE (GOPS)

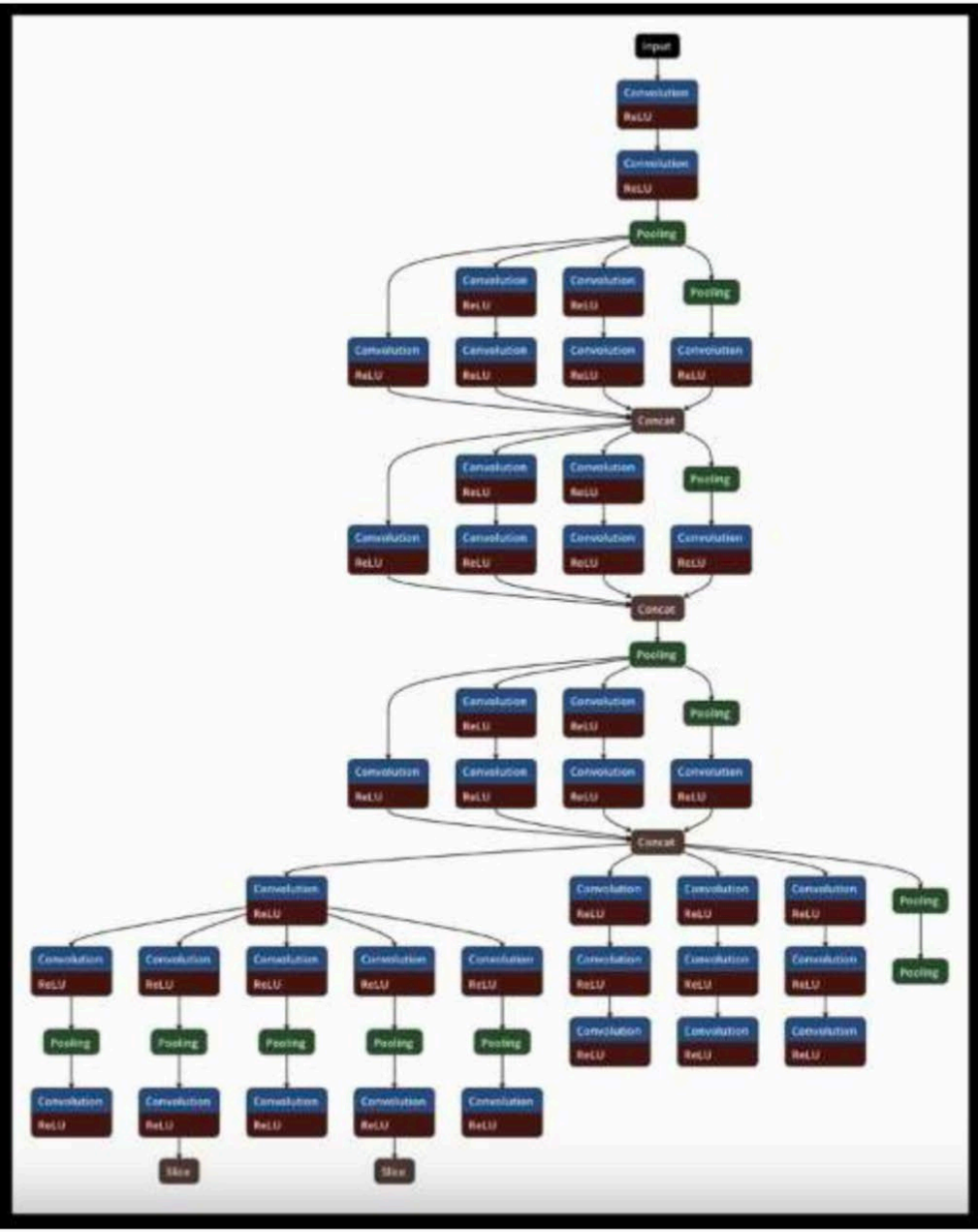


- 1.5 frame/s
- 17 frame/s
- 2100 frame/s

Frame per second for 35 GOP network: 2100

TESLA LIVE

- Best chip in world developed by Tesla
- All Tesla cars have this computer onboard
  - Switched from NVIDIA solution to this solution about a month ago
  - Model 3: 10days ago
- All Tesla car produced right now will have the HW necessary onboard → only requirement is SW update



Operation	MOPS	%
Convolution	34275	98.1
Deconvolution	576	1.6
ReLU	123	0.1
Pooling	13	0.2

99.7% of operations are multiply add

- Pooling & RELU are being done in dedicated HW resources
- Temp/intermediate data are stored in on-chip SRAM

TESLA LIVE

ARITHMETIC ENERGY

Scheme for minimizing requirement of chip power

Integer	Energy
Add	
8 bit	0.03pJ
32 bit	0.10pJ
Mult	
8 bit	0.20pJ
32 bit	3.00pJ

FP	Energy
FAdd	
16 bit	0.40pJ
32 bit	0.90pJ
FMult	
16 bit	1.00pJ
32 bit	4.00pJ

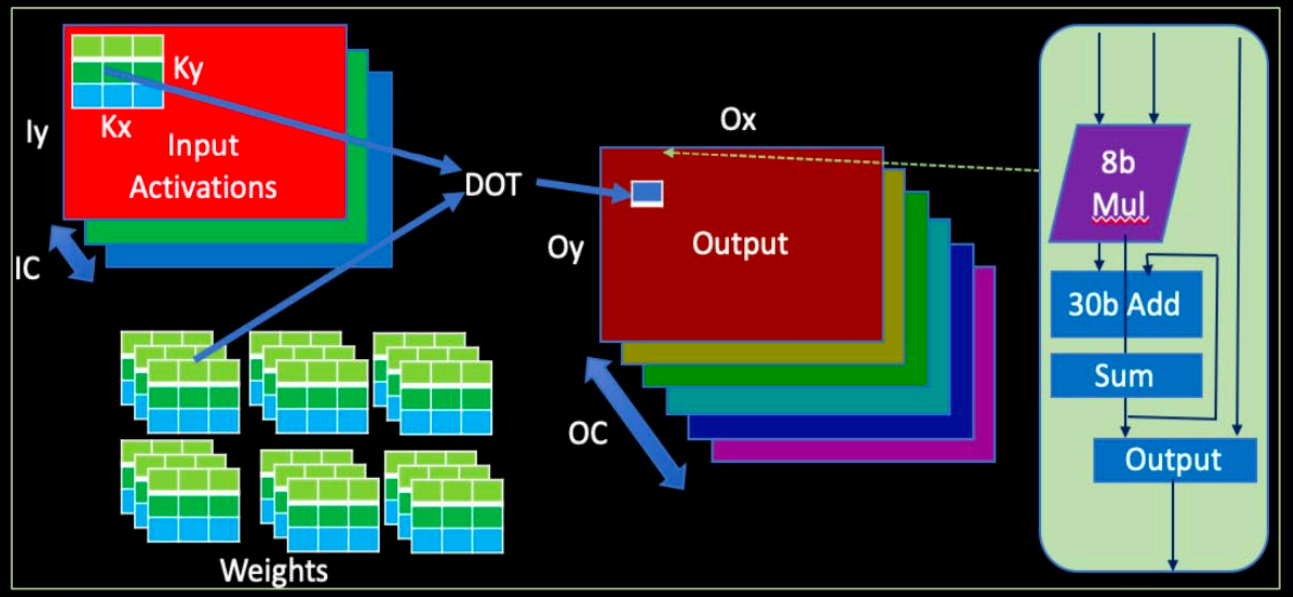
Memory	Energy
SRAM	(64 bit)
32KB	20pJ
DRAM	2000pJ

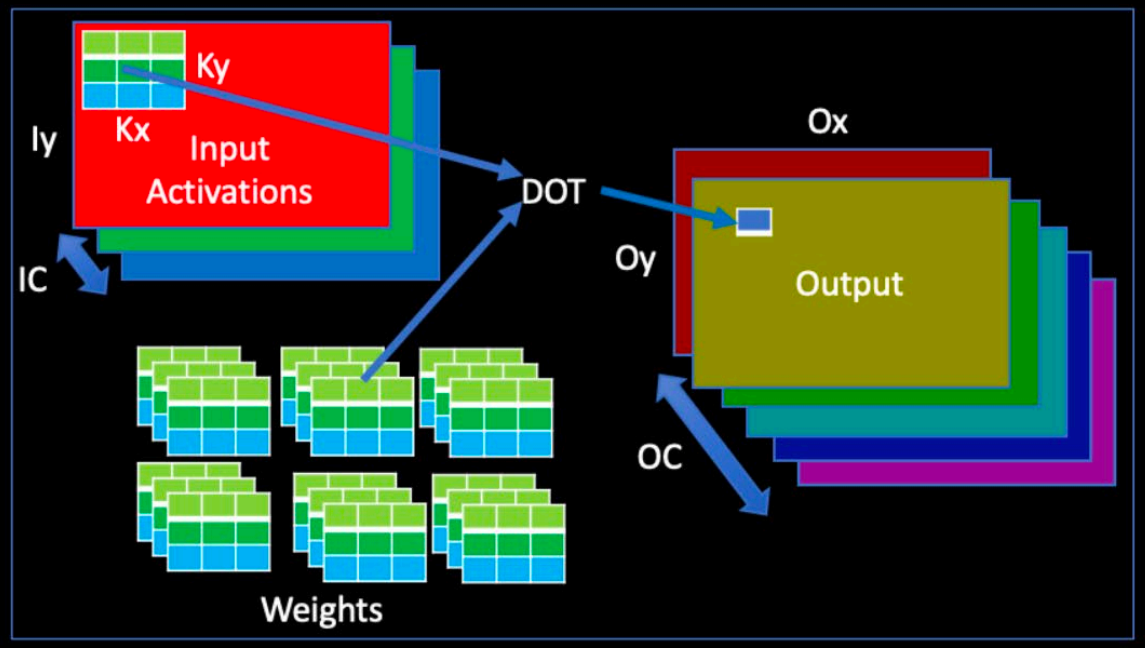
These are Ref. info taken from ISSCC 2014 paper (Horowitz)

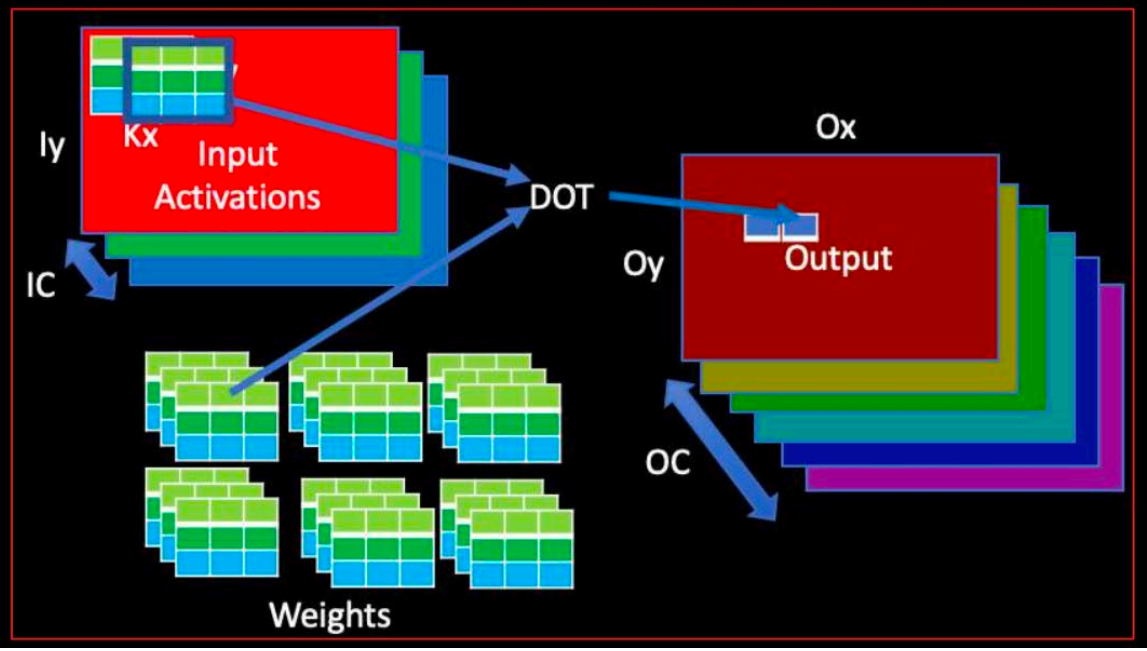
CONVOLUTION REFACTORED FLOW

- Merge Output X & Output Y to create larger input to process
- Process OutputX.Y and Output Channel 96 at a time.

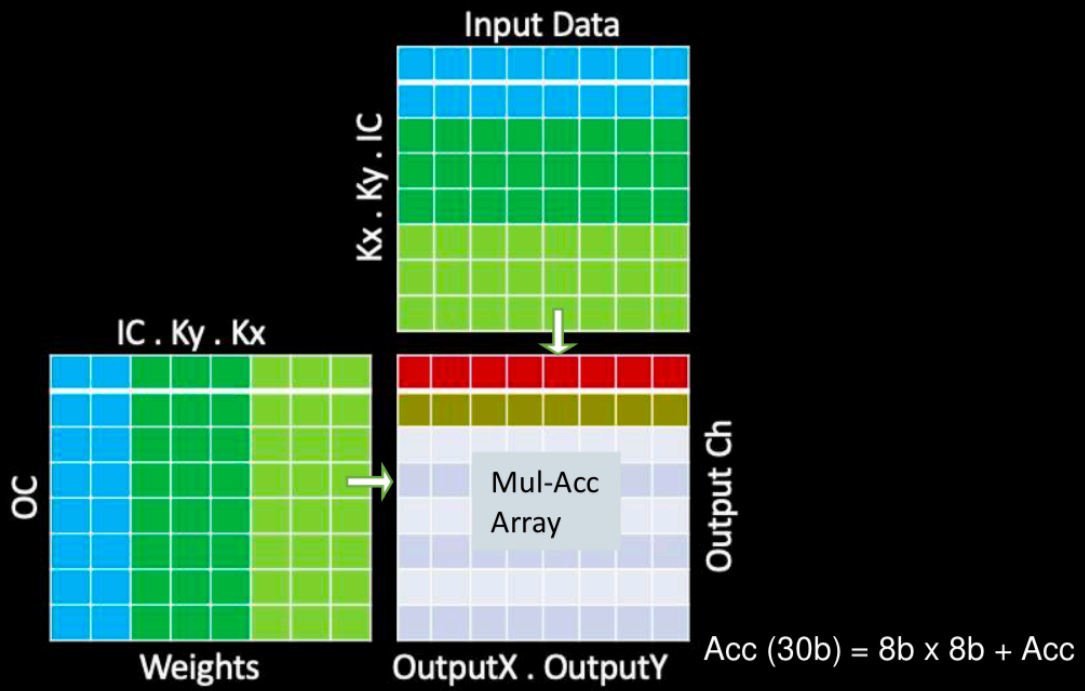
1. For each Image
2. For (Output X * Output Y), step 96
3. For each Output Channel, step 96
4. For each Input Channel
5. For each Input Y within KernelY
6. For each Input X within KernelX



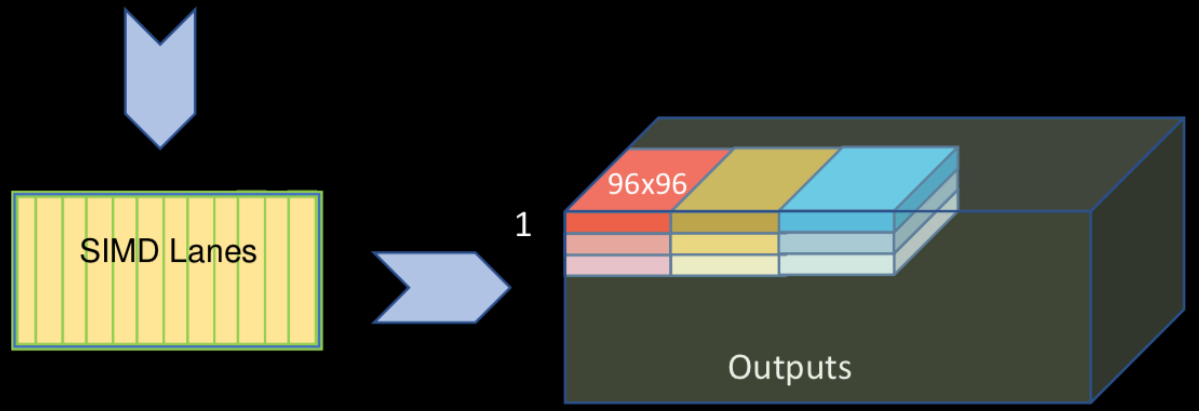




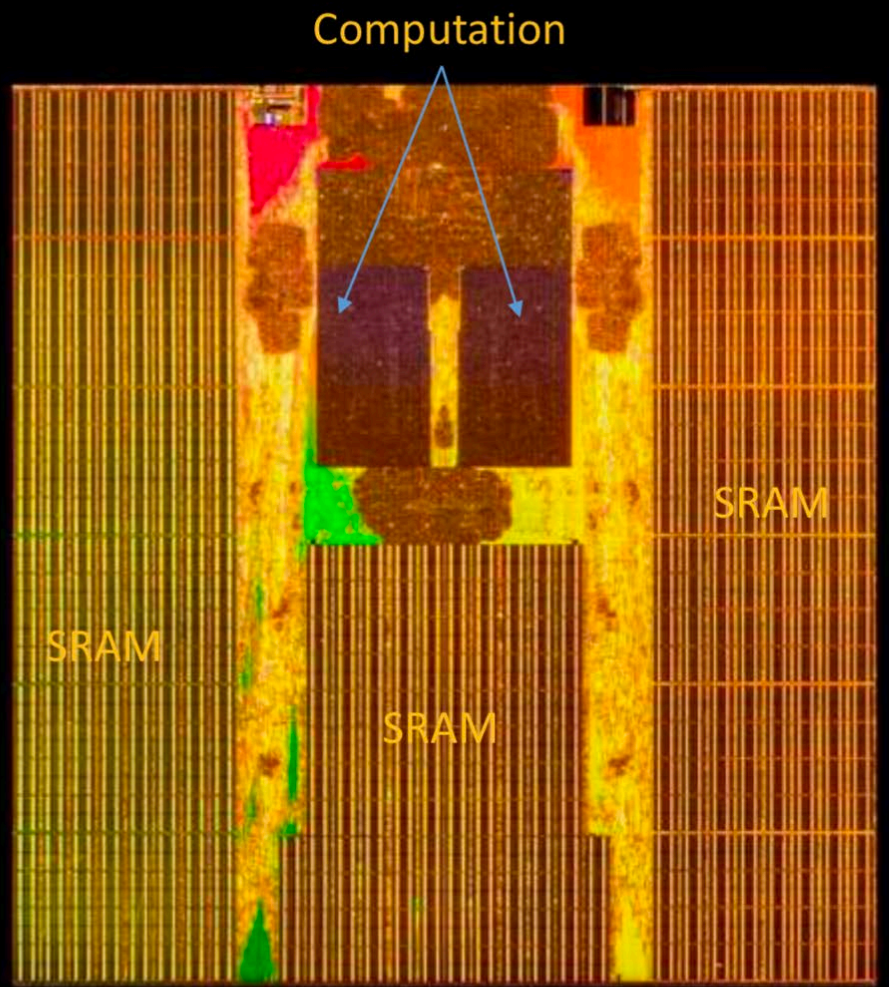
OUR COMPUTE SCHEME



- Maximum Data sharing (Reduced SRAM and DRAM activity)
- Minimized Data shifting power
- Further Power reduction with smarter shifting
- Increased compute Bandwidth Utilization



COMPUTE - MULTIPLY - ADD



32 MB SRAM

Every Clock

Activation data is read into the formatter

Weight data is read

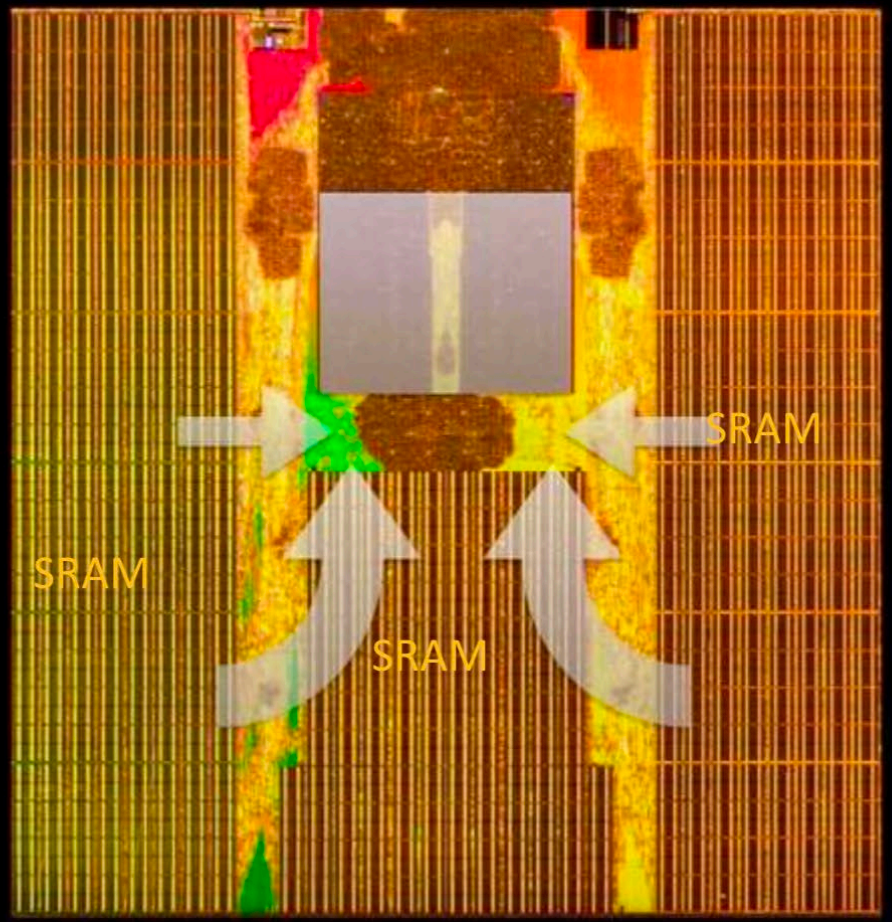
9216 multiply adds

2 GHz

36.8 TOPS

TESLA LIVE

COMPUTE - MULTIPLY - ADD



Every Clock

Activation data is read into the formatter

Weight data is read

9216 multiply adds

2 GHz

36.8 TOPS

- Reads 256B activations/single clock cycle from SRAM
- 128B weight data from SRAM/cycle from SRAM
- Combine 96x96 (9216 MAC/clock) in computing systems

TESLA LIVE

POST-PROCESSING

Every Clock

Results are shifted out of the array

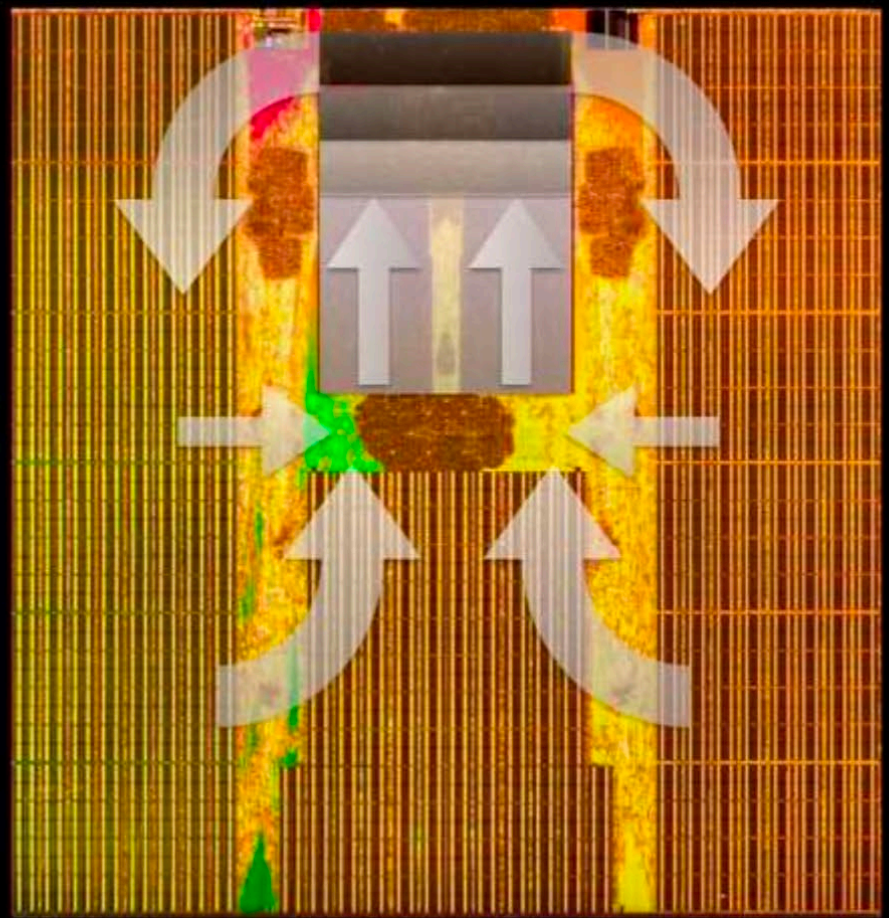
Dedicated ReLU unit

Optional Pooling unit

Write Buffer

Write Result

REPEAT UNTIL DONE



Bandwidth matters

256B Read

128B Read

128B Write

2 GHz

1 TB/sec SRAM Bandwidth per engine

2TBps for whole chip

TESLA LIVE

SMALL INSTRUCTION SET

DMA Read

DMA Write

Convolution

Deconvolution

Inner-product

Scale

Eltwise

Stop

TESLA LIVE