# In-Datacenter Performance Analysis of a Tensor Processing Unit

Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, Doe Hyun Yoon

Presented By: John Hsu

# Paper #1 — Framing the Contribution

*Jouppi et al., ISCA 2017 — "In-Datacenter Performance Analysis of a Tensor Processing Unit"*

**WHY THIS CHIP EXISTS**

### Inference-only, production-scale

In 2013 Google projected that DNN speech recognition alone would require doubling its entire datacenter. CPUs and GPUs were too expensive and too power-hungry for always-on inference serving at that scale. A custom ASIC was the only viable path.

**THE CORE ARCHITECTURAL BET**

### Determinism over throughput

User-facing services require strict latency guarantees. CPUs and GPUs optimize average throughput but introduce timing variability. TPUs eliminate these sources of variance and are designed for consistent 99th-percentile latency.

**THE SELF-ADMITTED WEAKNESS**

### Memory bandwidth left on the table

The TPU used DDR3 memory (34 GB/s), though the paper showed that GDDR5 already used in the K80 GPU could have tripled throughput. This reflected a schedule-driven compromise: the 15-month deadline took priority over the higher-performance option.

### The Headline Numbers — and What They Hide

**15–30×** faster than K80 GPU (weighted mean: 29×)

**30–80×** better TOPS/Watt vs GPU or CPU

**~23%** mean MAC utilization on MLP/LSTM workloads

*Note: 23% utilization is not a failure — it's the roofline story.*

Source: Jouppi et al. 2017

# Neural Networks — What You Need to Know for This Paper

## Why INT8? The Arithmetic Case for Quantization

| | |
|---|---|
| TRAINING | float32 (gradients are tiny — need precision) |
| | *quantization ↓* |
| INFERENCE | **int8** ← 4× narrower |

### Hardware savings: int8 vs fp16

| | |
|---|---|
| Multiply — Energy | 6× less |
| Multiply — Area | 6× less |
| Add — Energy | 13× less |
| Add — Area | 38× less |

**Payoff:** int8 lets the TPU pack **25× more MACs** than the K80 GPU into a smaller, lower-power die. This is not magic — it is arithmetic.

## The Workload Mix — The Most Provocative Table in the Paper

| NN Type | Ops / Weight Byte (Operational Intensity) | % of TPU Fleet |
|---|---|---|
| **MLP** *MLP0, MLP1* | 168–200 | **61%** |
| **LSTM** *LSTM0, LSTM1* | 64–96 | **29%** |
| **CNN** *CNN0, CNN1* | 1750–2888 | **5%** |

**Ops/Weight Byte** = how many operations per byte loaded from DRAM. **Low → memory-bound. High → compute-bound.**

### The Inconvenient Finding

In 2016, ~15% of computer architecture papers studied CNN accelerators. Google's actual production workload is 90% MLPs and LSTMs — the memory-bound ones. CNNs are just 5%. The research community was optimizing for the wrong workload.

### Why small batch sizes?

Larger batches improve efficiency by amortizing weight-loading costs, increasing effective ops/byte. But user-facing applications cannot wait to accumulate large batches, so latency constraints force small batches, making memory bandwidth the primary bottleneck.

*Source: Jouppi et al. 2017 — Table 1*

# TPU Origin, Architecture & Die Floor Plan

**2006**

GPUs, FPGAs & ASICs evaluated — concluded spare CPU capacity was 'good enough'

**2013**

Voice search projection: 3 min/day DNN speech → datacenter must double. Custom ASIC approved.

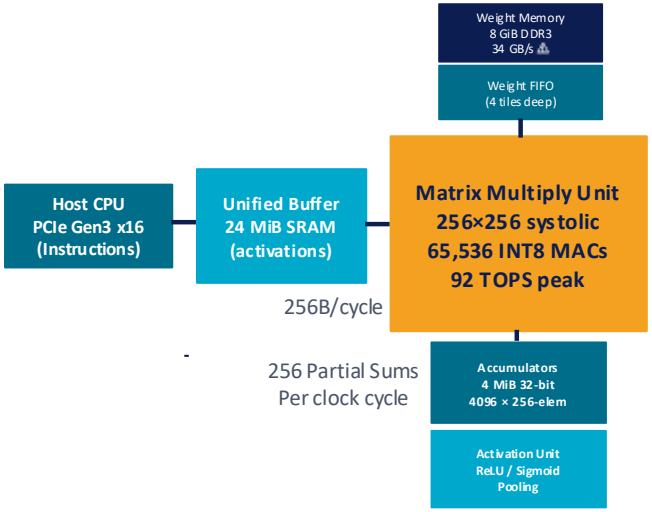
**+15 mo.**

TPU designed, verified, built & deployed in datacenters. Normal chip cycle: 3–5 years.

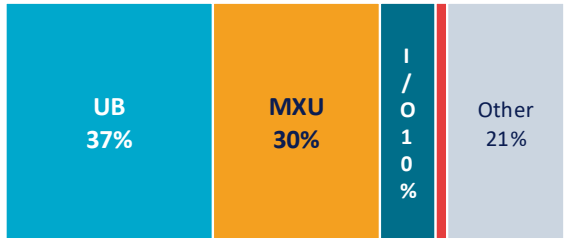
**Key decision:** PCIe coprocessor (not CPU-integrated) — plugs into existing servers like a GPU, zero deployment risk. Host CPU sends instructions; TPU executes. Closer to an FPU than a GPU.

*Data flow: UB → MXU → Accumulators → Activation → UB | Weights: Weight DRAM → FIFO → MXU*

## Block Diagram — Data Flow




## Die Floor Plan — Where Silicon Goes




Data buffers (UB+Acc) — 37%

I/O — 10%

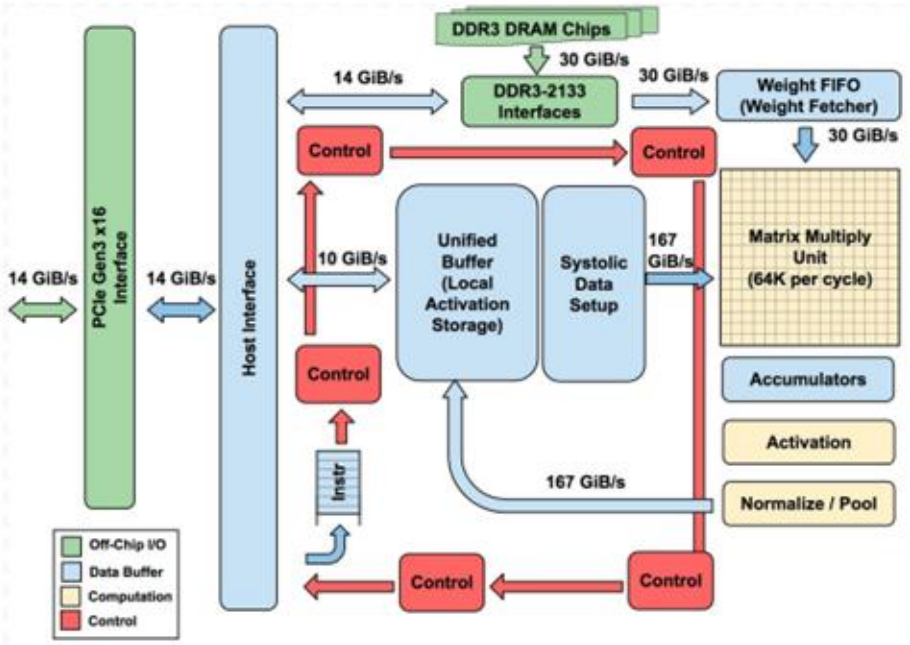
Compute (MXU+Activation) — 30%

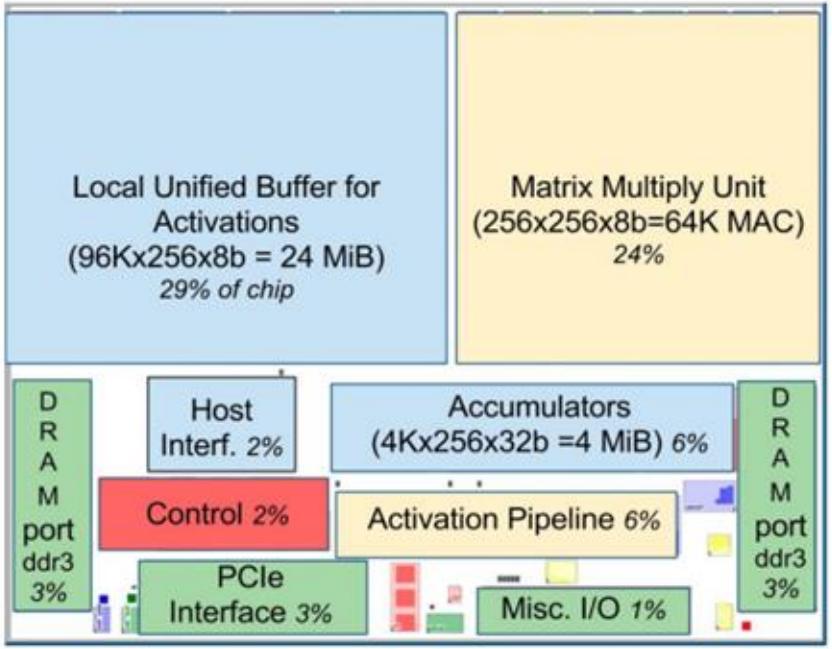
**Control — only 2% ← vs ~20% in a CPU**

A CPU spends ~50% of die on caches, ~20% on control. The TPU spends **2% on control** — nearly everything is datapath. This is what domain-specificity buys.

*Source: Jouppi et al. 2017 — Figures 1 & 2*

# TPU Paper Architecture Diagrams



# Systolic Execution, CISC Instructions & Platform Comparison

## Systolic Execution — How the MXU Saves Energy


① Weights preloaded from top, stay stationary — no re-fetching

② Activations flow left→right, each reused 256 times across the row

③ Results ripple as a diagonal wavefront — each cell: weight × activation → accumulate

**Why this saves energy:** Each activation is read from SRAM **once** then reused 256×. Reading large SRAM costs far more power than arithmetic — systolic reduces reads from $O(N^2)$ to $O(N)$.

## 5 Key CISC Instructions

| | |
|---|---|
| **Read_Host_Memory** | CPU memory → Unified Buffer |
| **Read_Weights** | Weight DRAM → FIFO (decoupled, issues early) |
| **MatrixMultiply/Convolve** | B×256 in → B×256 out, B pipelined cycles (12 bytes) |
| **Activate** | ReLU/Sigmoid/Pooling: Accumulators → UB |
| **Write_Host_Memory** | Unified Buffer → CPU memory |

*CISC rationale: PCIe is slow — wide instructions do lots of work per send. MatrixMultiply keeps MXU busy B cycles, hiding PCIe latency inside long MAC chains.*

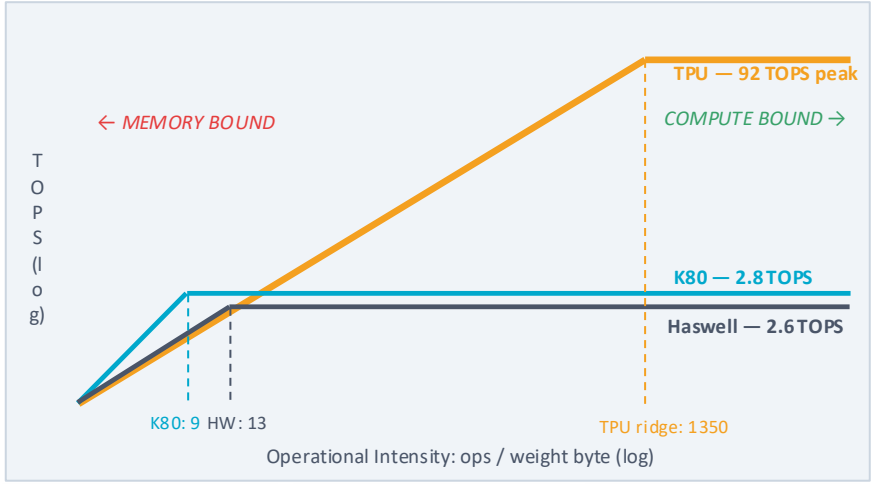
## Platform Comparison — Table 2

| | Haswell CPU | K80 GPU | TPU |
|---|---|---|---|
| **Peak TOPS** | 2.6 | 2.8 ⚠ | 92 |
| **Memory BW** | ~50 GB/s | 160 GB/s | 34 GB/s ⚠ |
| **Memory Type** | DDR4 | GDDR5 | DDR3 ⚠ |
| **MACs** | ~512 | ~2,600 | 65,536 |
| **TDP** | 145W | 150W | 40W |
| **Die size** | ~662mm² | ~561mm² | ≤331mm² |

*Source: Jouppi et al. 2017 — Figures 3, 4 & Table 2*

# The Roofline Model — Diagnosing Where Performance Goes

## What the Roofline Model Shows




### Y-axis: TOPS

Performance ceiling. The flat top is peak compute. Hardware cannot exceed this.

### X-axis: Ops/Byte

Operational intensity. How many operations per byte loaded from weight memory.

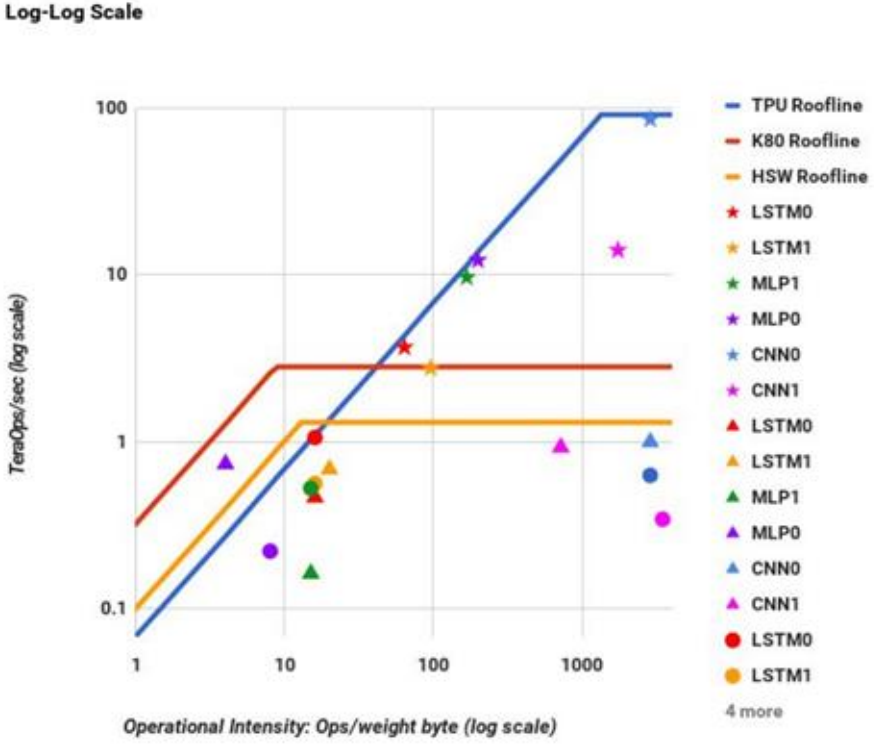
### Ridge Point

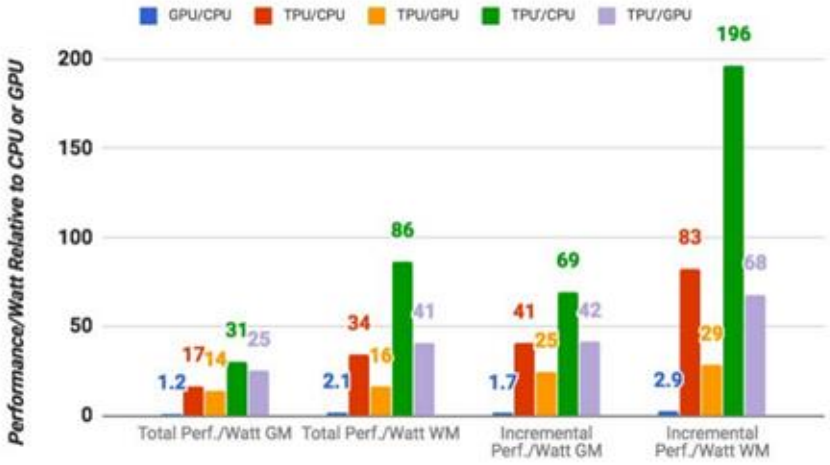
Where memory and compute ceilings meet. TPU's ridge (1350) is 150× right of K80's (9).

## Figure 5 — TPU Roofline: Where the Six Apps Land


*Source: Jouppi et al. 2017 — Figures 5, 6, 7*

# Log-Log Scale & Performance/Watt



# CNN1 Anomaly (Table 3) & The 99th-Percentile Problem (Table 4)

## Table 3 — Why CNN1 Only Achieves 14.1 TOPS Despite High OI

Approximate cycle breakdown (from Table 3 performance counters):


### ① Shallow Feature Depth — Wasted MACs

Even when the MXU is active, only ~50% of 65,536 MACs hold useful weights. CNN1 has layers with fewer than 256 output channels — the other half of the 256×256 array runs idle with dummy weights. The array is sized for maximum density, not minimum layer size.

### ② FC Layers Inside a 'Compute-Bound' App — Memory Stalls

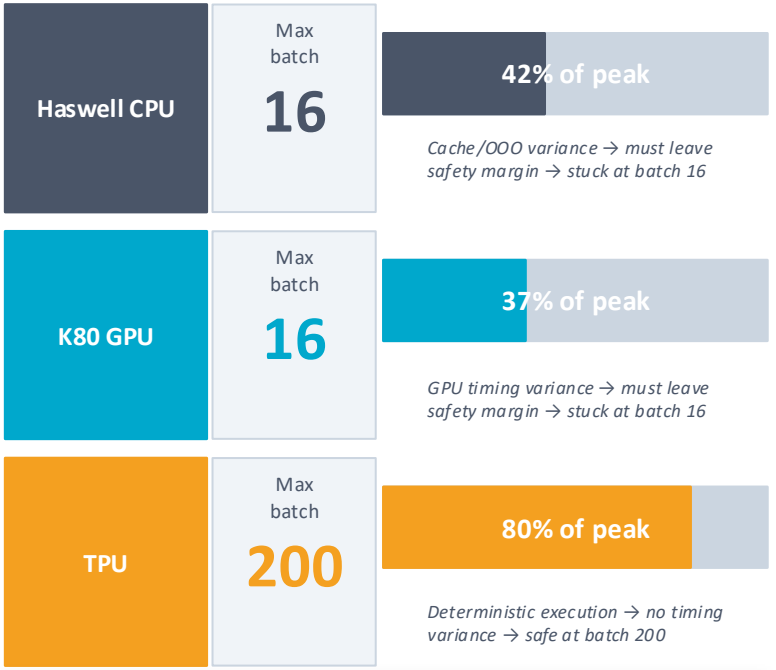
CNN1 contains 4 fully connected layers running at just 32 ops/byte — far left of the ridge point. During these layers, the MXU stalls waiting for weights from DDR3. ~35% of all cycles are spent in this weight-fetch wait state despite the overall app having OI of 1750.

### ③ RAW Hazards — Pipeline Stalls at Layer Boundaries

Read-After-Write hazards: layer N's activations must fully write to the UB before layer N+1's MatrixMultiply can read from it. The 4-stage pipeline stalls waiting for the dependency to clear. Performance counters show 23% of cycles stalled for RAW hazards.

## Table 4 — The 99th-Percentile Deadline Cripples GPUs & CPUs

MLP0 has a hard **7 ms** 99th-percentile deadline. Larger batch sizes raise efficiency (more ops/byte) but add latency. Each platform is constrained to the largest batch that still meets 7 ms.




K80 is only 1.1× faster than CPU — its peak throughput advantage is entirely wasted by timing variance forcing batch size 16. TPU's determinism lets it use batch 200 and reach 80% of peak.

*Source: Jouppi et al. 2017 — Tables 3 & 4*

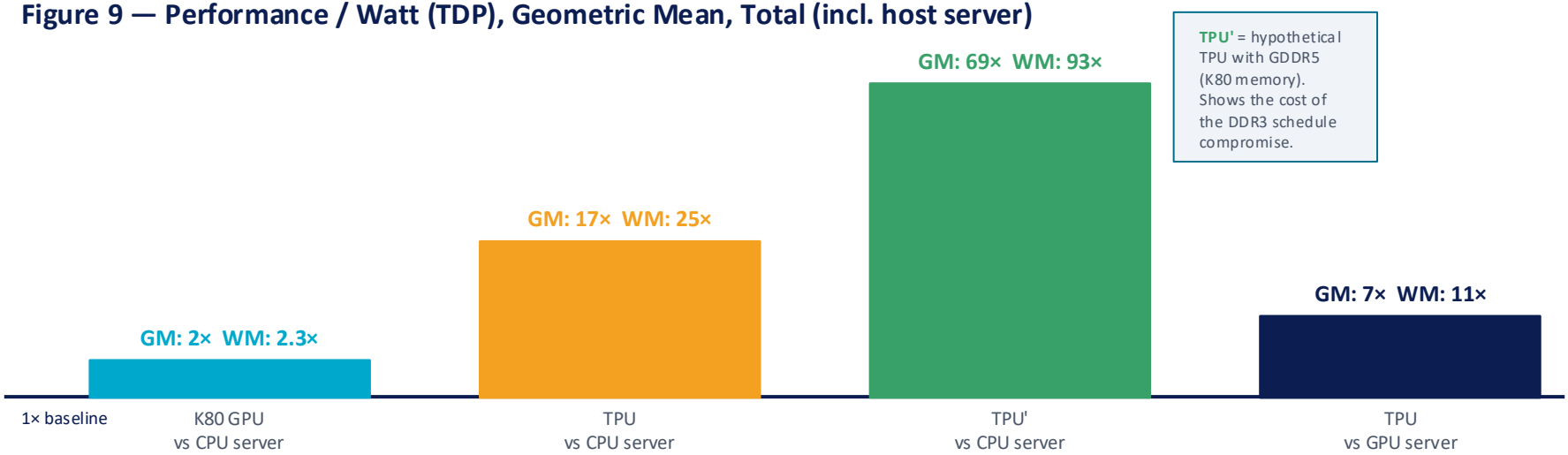
# Bottom Line: Inference Performance & Power Efficiency

## Table 6 — Performance Relative to Haswell CPU (per die, including host overhead)

| | MLP0 | MLP1 | LSTM0 | LSTM1 | CNN0 | CNN1 | GM | WM |
|---|---|---|---|---|---|---|---|---|
| K80 GPU | 1.3× | 0.8× | 1.5× | 1.1× | 4.7× | 1.2× | **1.1×** | **1.9×** |
| TPU | 70× | 26× | 10× | 15× | 36× | 7× | **14.5×** | **29.2×** |

**GM** (Geometric Mean) = all six apps weighted equally — used when the mix is unknown. **WM** (Weighted Mean) = actual production mix (61% MLP, 29% LSTM, 5% CNN). K80 rises from 1.1× to 1.9×. TPU rises from 14.5× to 29.2×. MLP/LSTM make up 90% of real load — where the TPU's large-batch deterministic execution advantage is greatest.

## Figure 9 — Performance / Watt (TDP), Geometric Mean, Total (incl. host server)




*Source: Jouppi et al. 2017 — Tables 5, 6 & Figure 9*

# TCO, Performance/Watt & Energy Proportionality

## Why Power = TCO Proxy

### Two Ways to Measure Performance/Watt:

**TOTAL**

Host server power included.
"What does inference actually cost to run in a real datacenter?"

**INCREMENTAL**

Host server power excluded.
"Given I already have a host server, how efficient is the accelerator card itself?"

### Performance/Watt Results (vs Haswell CPU):

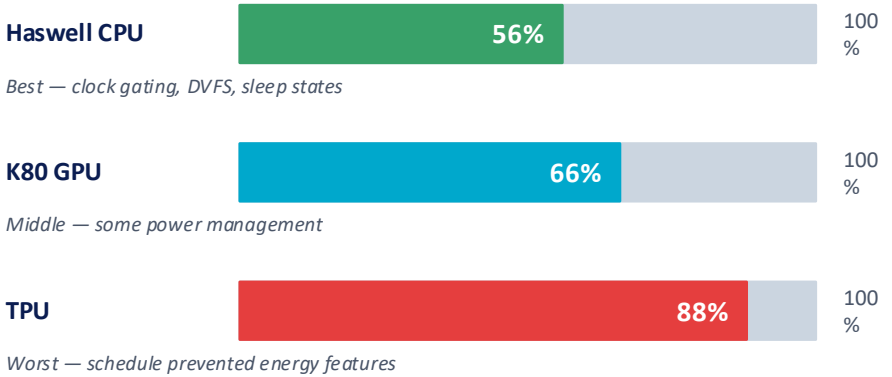
| | Total | Incremental |
|---|---|---|
| K80 GPU | 1.2–2.1× | 1.7–2.9× |
| TPU | 17–34× | 41–83× |
| TPU vs K80 | 14–16× | 25–29× |

**The business case:** "The incremental performance/Watt was our company's justification for a custom ASIC." At 41–83× incremental efficiency, adding a TPU card to an existing server gives dramatically more work per extra watt than any off-the-shelf alternative.

## Energy Proportionality — The TPU's Admitted Weakness

**Energy proportionality**: power drawn should scale with work done. A perfectly proportional chip at 10% load would draw 10% of full power. TDP provisions infrastructure; actual consumption pays the electricity bill.

**Power consumed at 10% workload vs full load:** ← *Ideal: 10% load = 10% power*




**Despite poor proportionality:**

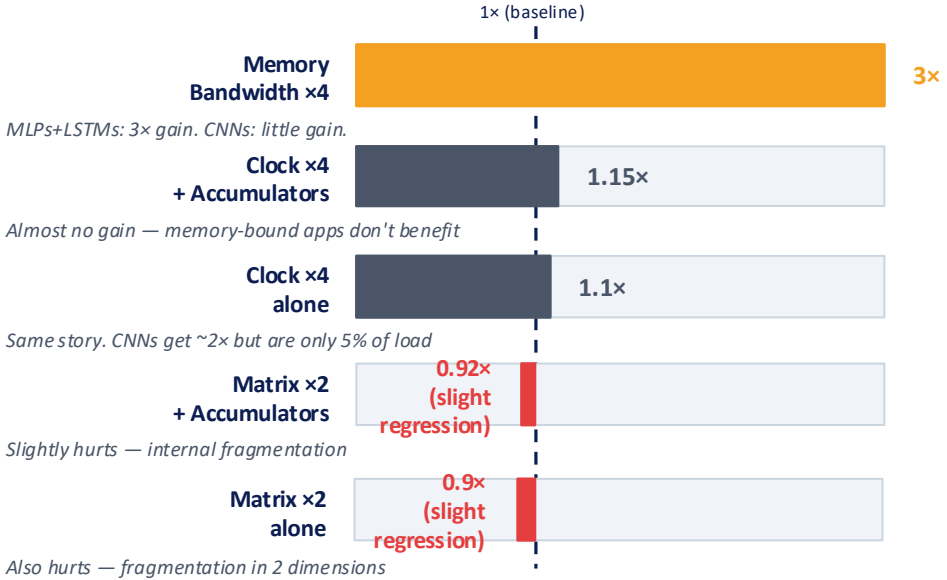
Haswell server + 4 TPUs = **<20% more power** than Haswell alone, but **80× faster on CNN0.** At high utilization — which Google optimizes for — the TPU's low absolute power and massive throughput overwhelm the proportionality weakness.

*Source: Jouppi et al. 2017 — Sections 5 & 6, Figure 9 & 10*

# Evaluation of Alternative TPU Designs — What Actually Moves Performance

## Figure 11 — Sensitivity Analysis: Varying 5 Parameters

Performance model validated at **~8% average error** vs hardware counters (Table 7) — trustworthy enough to draw conclusions from hypotheticals.




**Why bigger MXU hurts — Internal Fragmentation (LSTM1, 600×600 matrix):**
256×256 MXU: 9 tiles × 256 cycles = ~18 µs | 512×512 MXU: 4 tiles × 512 cycles = ~32 µs (slower — fewer steps but each takes 2× longer, wasted padding fills 2D array)

## TPU' — What 15 More Months Could Have Built

**Clock: 700 → 1050 MHz (+50%)** **≈ 0×**

More aggressive logic synthesis. But memory-bound workloads dominate — a faster clock just means the MXU waits faster. Real gain: negligible.

**DDR3 → GDDR5 memory (+5× BW)** **3.9× WM**

Shifts ridge point: 1350 → 250 ops/byte. MLPs and LSTMs suddenly reach the compute ceiling. Requires doubling memory channels (+10% die) but UB shrinks to 14 MiB (−10% die). Net area: neutral.

**TPU' = clock + GDDR5:**
**GM 2.9× WM 3.9×** (before host overhead)
GM 1.9× WM 3.2× (after host overhead)

**TPU' Performance/Watt vs Haswell:**
Total: 31–86× | Incremental: 69–196× (up from 17–34× / 41–83× for base TPU)

*Source: Jouppi et al. 2017 — Section 7, Figures 10 & 11, Tables 7 & 8*

# The Design Process for Google's Training Chips: TPUv2 and TPUv3

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

# Paper 2: TPUv2 & v3 - Why Training, and the Design Philosophy

TPUv1 solved inference. It immediately exposed the next bottleneck: **training.** TPUv2 deployed 2017. The pivot took a small team under tight time constraints.

## Why Training Is Fundamentally Harder Than Inference

### ① 10× More Computation

Inference: ~$10^9$ FLOPs. Training: $10^{19}$–$10^{22}$ FLOPs. More than 10 orders of magnitude larger. Also more types: backprop requires matrix transpositions, loss functions, gradient accumulation — none needed at inference.

### ② More Memory

Inference discards activations after each layer. Training must keep them for backpropagation — you need the full forward pass stored simultaneously to compute gradients. Far exceeds TPUv1's 24 MiB UB.

### ③ Wider Operands — bfloat16, Not INT8

INT8 has insufficient dynamic range for gradient accumulation. Small gradients underflow to zero. bfloat16 keeps float32's 8-bit exponent (preserving range) with 7-bit mantissa — designed specifically for training.

### ④ More Programmability

Inference runs fixed, known models. Training is experimentation: new architectures, optimizers, loss functions every week. Hardware must work correctly for workloads it has never seen, first time, without tuning.

### ⑤ Harder Parallelization

Inference chips are independent — no coordination needed. Training across thousands of chips must produce one consistent parameter set via synchronized gradient all-reduce. Off-chip communication is easily the bottleneck.

## The Two-Bucket Priority Framework

**Limited team. Limited time. Must choose.**

### ✅ BUCKET 1 — Must Do Great

1. **Build quickly**
2. Achieve high chip performance
3. **Scale efficiently to many chips**
4. **Work for new workloads out of the box**
5. Be cost effective

### ⚙ BUCKET 2 — Good Enough

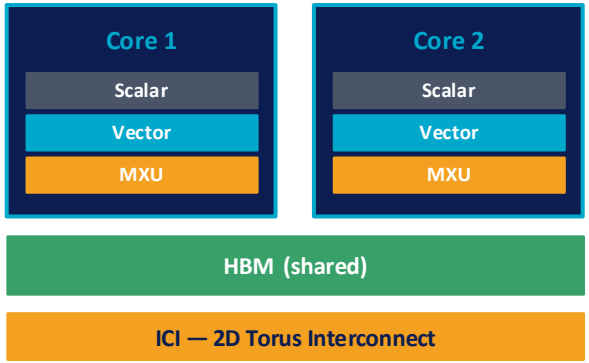
Energy proportionality · Binary ISA compatibility
Full programmability for every possible workload
Sparse support · Broadcast instructions · …

*"Building a good system is as much about what you decide **not** to do as what you decide to do." — Norrie et al. 2021*

*Source: Norrie et al. 2021 — IEEE Micro*

# TPUv2 Architecture: Why Two Cores?

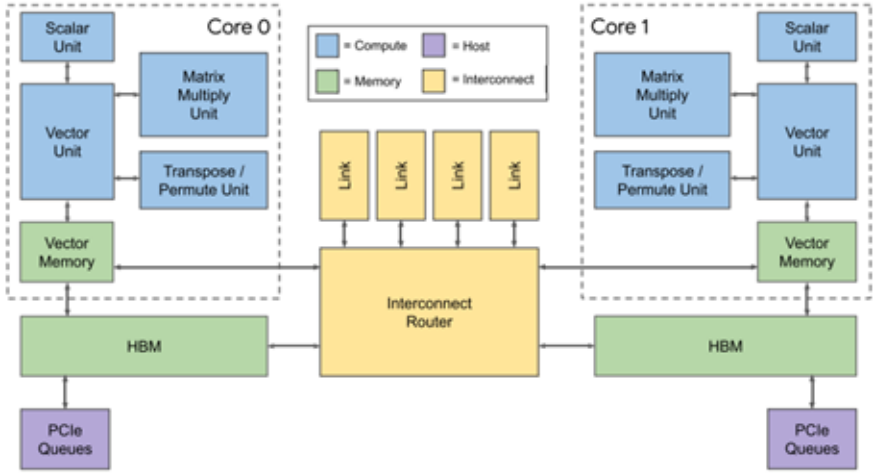
## Why Two Cores? (Figure 2)




**Why not one large core?**
Wire latency. As a core grows physically, signals take longer to cross it. One large core would have unacceptably long pipeline latencies.

**Why not many small cores?**
Programmability (goal ④). Many tiny cores must be explicitly coordinated. Hard to get new workloads running correctly without tuning.

**Why exactly two?**
Two hits the right balance: a second core fits in a reasonable die size (goal ⑤), programmer sees one unified instruction stream per core — easy to reason about (goal ④).




*Source: Norrie et al. 2021 — IEEE Micro, Figure 1 & 2*

# TPUv2 Architecture: Five Piecewise Edits from TPUv1

The paper explicitly frames TPUv2 as **five targeted transformations of TPUv1** — not a clean-sheet redesign. Constraint ① (build quickly) drove this decision.

## Figure 1 — Transforming TPUv1 into a Training Chip

**(a)-->(b) Merge Split SRAM → Unified Vector Memory**

TPUv1 split SRAM across fixed-function units (UB, Accumulators). Training needs flexible on-chip memory for activations, gradients, temporaries — all at once. Merging into one compiler-managed Vector Memory provides that flexibility. Serves goal ④.

**(c)-->(d) Fixed Activation Pipeline → Programmable Vector Unit**

TPUv1's hardwired pipeline: ReLU, sigmoid, pooling only. Training adds batch norm, transpose, arbitrary loss functions. A 128-lane × 8-sublane vector unit handles any element-wise op the compiler generates. Serves goal ④.

**(d)-->(e) MXU Becomes Co-processor of Vector Unit**

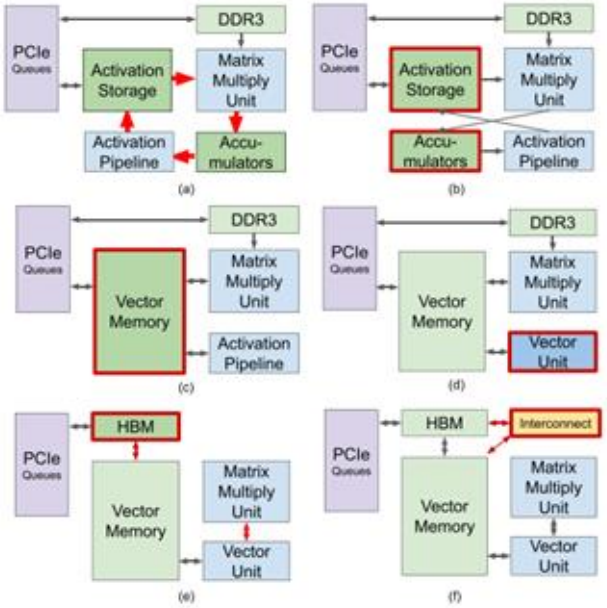
MXU attaches via push/pop FIFOs — vector unit orchestrates, MXU executes. Now uses bfloat16 instead of INT8 for gradient precision. DDR3 moves behind Vector Memory, forming a proper memory hierarchy for mutable parameters.

**(e) DDR3 → In-Package HBM (~20× Bandwidth)**

DDR3 was the #1 bottleneck in TPUv1 (34 GB/s). HBM delivers ~700 GB/s, shifting the roofline ridge point from 1350 to ~250 ops/byte. Critical for training which must read AND write parameters repeatedly. Serves goal ②.

**(f) Add ICI — InterChip Interconnect (2D Torus)**

Training ($10^{19}$–$10^{22}$ FLOPs) cannot fit on one chip. Custom ICI: 4 off-chip links × 60 GB/s = 240 GB/s total. Forms a 2D torus. All-reduce for gradient sync maps naturally to torus topology. Remote HBM DMAs look identical to local HBM. Serves goal ③.




*Source: Norrie et al. 2021 — IEEE Micro, Figure 1 & 2*

# TPUv2 Core - VLIW, Scalar, Vector & Matrix Execution

**VLIW:** compiler packs parallel ops into one 322-bit bundle; hardware executes all in one cycle. No OOO logic. **Tradeoff:** binaries break between chip generations — ISA compatibility deliberately put in Bucket 2.


*← 322 bits — all slots execute in a single clock cycle →*

**Execution Hierarchy (data flows: Scalar → Vector → MXU → Result FIFO → Vector)**

## ① SCALAR UNIT

**Instruction Memory**
DMA-loaded SRAM — not a cache. Compiler pre-loads VLIW bundles via software-managed overlays. Simple (goal ①), not flashy.

**Dual-issue ALU**
32-deep register file of 32-bit words. Two scalar ops/cycle. Standard hold-condition interlocks.

**DMA Port → HBM**
Async DMAs between HBM and Scalar Memory SRAM. Sync flags fire on completion — program can stall until data arrives.

**Forwards Downstream**
Sends decoded bundle + up to 3 scalar register values to the vector unit each cycle.

→

## ② VECTOR UNIT

**128 Vector Lanes × 8 Sublanes each**

**Per-lane: Dual-issue ALU**
32-deep register file. Executes vector instruction slots from VLIW bundle.

**Sublanes — Batch Norm**
8 sublanes/lane → 8×128 vector ops/cycle. Extra vector ratio needed for batch normalization (training op, absent in TPUv1).

**Vector Memory**
Each lane's local SRAM slice. Compiler-managed loads/stores. DMA connects to HBM.

**Result FIFO**
Captures MXU results. Pop slots drain it. Decouples vector timing from MXU latency — compiler doesn't need the exact MXU finish time.

→

## ③ MATRIX UNIT (MXU)

**128×128 systolic array**
32,768 ops/cycle per MXU. Two MXUs per chip = 65,536 total — same as TPUv1 but split across 2 cores.

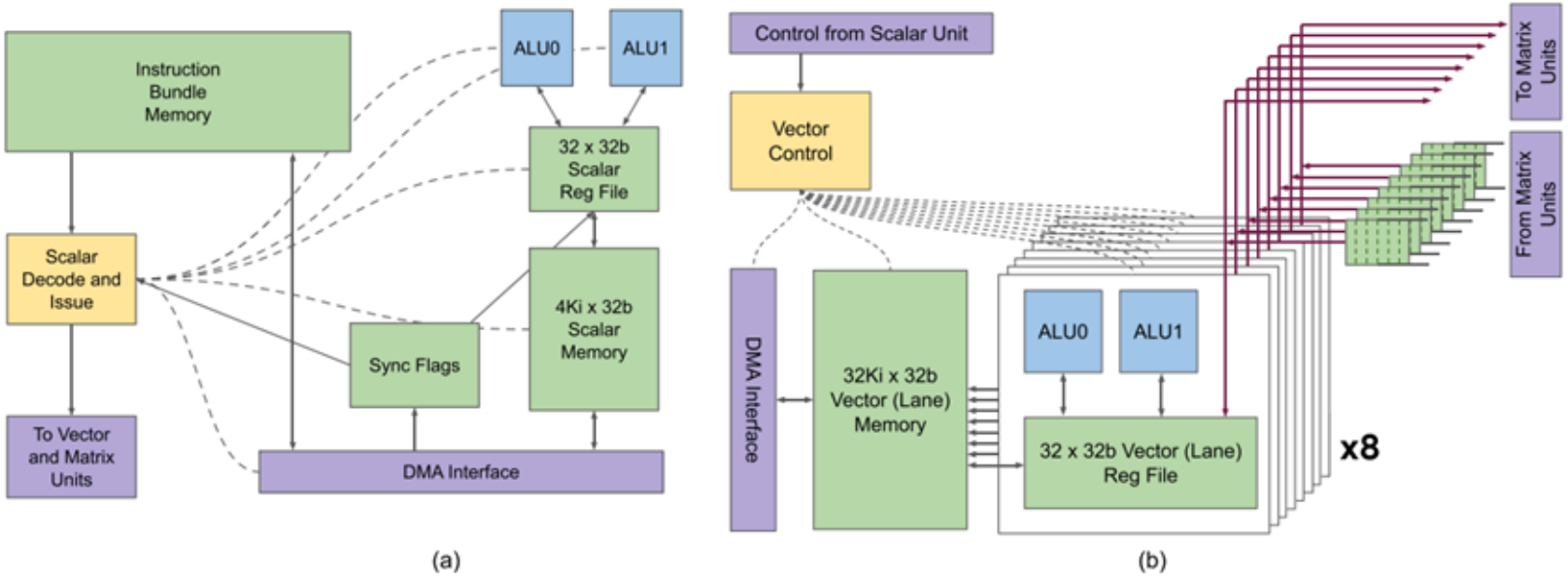
**LHS streams, RHS stationary**
Left-hand matrix streams; right-hand is preloaded and stationary. RHS can be transposed on load — required for backpropagation.

**bfloat16 × fp32 accumulate**
Multiply in bfloat16 for density. Accumulate in full 32-bit float for precision. At 7nm: bf16 add 0.11 pJ vs fp16 0.16 pJ. bf16 multiply 0.21 pJ vs fp16 0.31 pJ.

**No loss scaling — key advantage**
199/200 ML experts at Vector Institute use fp32 on GPUs to avoid fp16 loss scaling. bfloat16 removes this complexity. TPUv2 first — ARM, Intel, NVIDIA followed.

*Source: Norrie et al. 2021 — Figure 3*

# (a) TPUv2 core scalar unit and (b) single vector lane



# TPUv2 Memory System & ICI - Feeding the Datapath at Scale

## Memory System — Two Levels, Compiler-Controlled

**ON-CHIP SRAM - Vector Memory + Scalar Memory**
Architecturally visible - compiler explicitly addresses every byte. Predictable timing, no cache misses. Feeds the core datapath directly.

↕ *async DMA | sync flag fires on completion → program stalls until data arrives*

**IN-PACKAGE HBM - 700 GB/s per chip (20× TPUv1's DDR3)**
**Split: each core owns half the chip's HBM.** Why split? Goal ①: simplifies design and verification vs shared arbitrated bus. **Strided DMAs** allow slicing sub-dimensions of tensors without manual reshape.

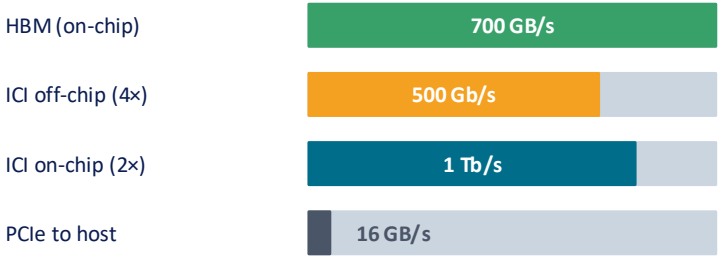
**PCIe Host - 16 GB/s ('the thin straw')**
Input data only. Never route weights/activations here.

**Clean Partitioning of Concerns:**
**Core (blue):** tightly scheduled, predictable. **Memory (green):** async DMA prefetch from HBM. **Sync flags:** handoff between regimes.

## InterChip Interconnect (ICI) — Pod Supercomputer

Single-chip training = months. ICI connects chips into a supercomputer = hours. Goal ③.

**Bandwidth:**




**2D Torus Topology**
4 off-chip links connect each chip to 4 neighbors with wrap-around. Matches ring-allreduce pattern used for gradient sync exactly. Some links within tray; rest through cables across racks.

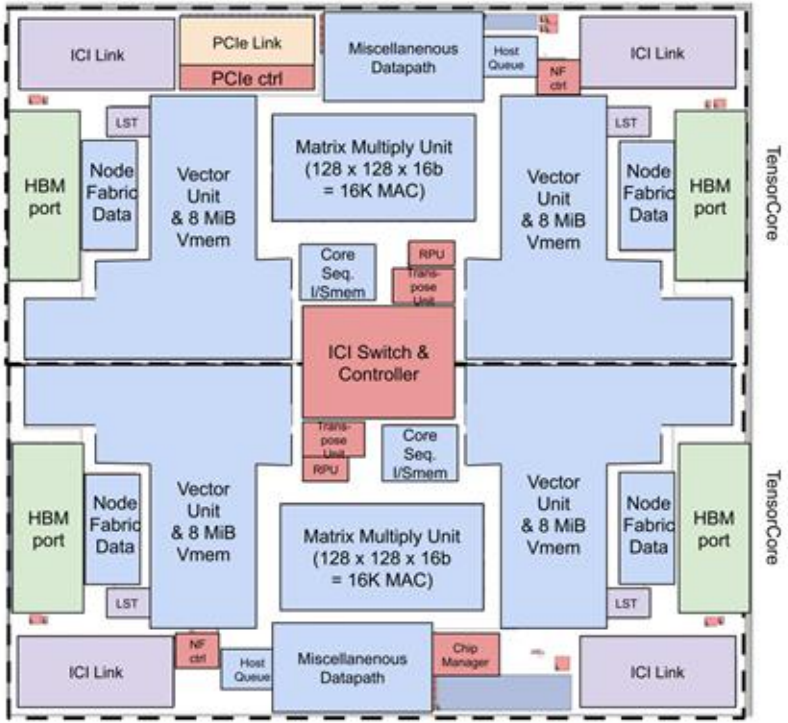
**Push-Only DMA Abstraction**
DMAs to remote chips look identical to local HBM DMAs except push-only. Simplifies hardware (goal ①). Compiler treats full pod memory as one address space (goal ④).

**Synchronous > Async Training**
Async training (parameter server) was previously standard. Google's studies showed sync converges better. High-bandwidth ICI makes synchronous all-reduce feasible at scale.

**System Balance (goal ③)**
PCIe (16 GB/s) feeds input data in. ICI (500 Gb/s) syncs gradients between chips. HBM (700 GB/s) runs computation. Minimizing host round-trips is critical to end-to-end throughput.

*Source: Norrie et al. 2021 — Figure 2 & 4*

# TPUv2 Floorplan (Figure 4)




**Floor Plan (Figure 4) - Density Over Beauty**

Two cores top/bottom, interconnect router in the center ('donut hole'), two MXUs at top-center and bottom-center. White areas filled with wiring. Paper says: 'not stylish, but good enough' focused effort on more important goals. MXU is not the largest die area despite delivering most FLOPS bfloat16 systolic density is extremely high (goal ⑤).

# TPUv3 - Performance Verdict

## Six Targeted Enhancements

**2× — Double MXUs → 2.7× peak FLOPS**

4 MXUs total. High leverage: bfloat16 systolic density is so high that doubling compute costs little die area. Goal ②.

**+30% — Clock: 700 → 940 MHz**

Pipeline tuning at same 16nm node. Boosts all compute-bound workloads ~30%. Goal ②.

**+30% — HBM Bandwidth + 2× Capacity**

Higher bus speed (+30% BW). Doubled capacity lets RNN0/1 use larger batches → higher OI → moves right on roofline. Enables models that didn't fit before. Goals ② ④.

**+30% — ICI: 500 → 650 Gb/s per link**

Faster gradient all-reduce. Less synchronization wait. Goal ③.

**4× — Scale: 256 → 1024 chip systems**

"It seems quaint now, but we thought 256 was huge." ML's appetite grew faster than predicted. Goal ③.

## Performance Verdict

### TPUv3 vs TPUv2 — Why 2.7× Peak Became 1.8× Real

| 2.7× | 1.3× | 1.8× |
|---|---|---|
| Peak compute gain (2× MXU + clock) | Memory / ICI / clock gain | Actual GM perf gain (both benches) |

*Memory-bound workloads (90% of load) couldn't use the doubled MXUs — memory only grew 1.3×. Roofline model predicted this.*

### TPUv3 vs V100 — MLPerf

Same chip perf despite TPUv3 being smaller (<700mm², 16nm) vs V100 (812mm², 12nm). In production: 5× faster because Googlers use fp32 on V100 to avoid fp16 loss scaling.

### CNN0 Above V100 fp32 Roofline

Not a physics violation. V100 uses Winograd Transform (fewer ops for 3×3 convolution) but roofline uses naive op count. Honest measurement quirk, not a miracle.

### MLP0/1 Cannot Run on V100

Embedding tables too large for V100 HBM. TPUv3's doubled capacity enables models that simply don't fit on contemporary GPU hardware practical advantage beyond raw speed.

### 96–99% Linear at 1024 Chips

Near-perfect linear speedup. General-purpose supercomputers rarely achieve this. 50× better GigaFLOPS/Watt vs HPC Linpack domain-specificity dividend.

## Conclusion - Report Card on the Five Goals

**① Build Quickly**

Co-design + DMA + compiler SRAM. Tradeoffs accepted: split HBM, messy layout, FIFOs. Schedule preserved.

**② High Perf.**

Systolic density + 700 GB/s HBM + XLA. 50× GigaFLOPS/Watt vs HPC supercomputer.

**③ Scale Up**

1024-chip pod. 96–99% linear speedup — rare even in HPC.

**④ New Workloads**

Linear algebra core + XLA + large HBM. New architectures work without special tuning.

**⑤ Cost Effective**

Dense MXUs, simple design, no gratuitous features. Good perf/watt/dollar.

*Source: Norrie et al. 2021 — Figure 5, Performance Synopsis & Conclusion*

# TPUv2 and TPUv3 Roofline Models (Figure 5)