Google’s TPU supercomputers train deep neural networks 50x faster than general-purpose supercomputers running a high-performance computing benchmark.

BY NORMAN P. JOUPPI, DOE HYUN YOON, GEORGE KURIAN, SHENG LI, NISHANT PATIL, JAMES LAUDON, CLIFF YOUNG, AND DAVID PATTERSON

A Domain-Specific Supercomputer for Training Deep Neural Networks

The recent success of deep neural networks (DNNs) has inspired a resurgence in domain specific architectures (DSAs) to run them, partially as a result of the deceleration of microprocessor performance improvement due to the slowing of Moore’s Law. DNNs have two phases: training, which constructs accurate models, and inference, which serves those models. Google’s Tensor Processing Unit (TPU) offered 50x improvement in performance per watt over conventional architectures for inference. We naturally asked whether a successor could do the same for training. This article explores how Google built the first production DSA for the much harder training problem, first deployed in 2017. Computer architects try to create designs that maximize performance on a set of benchmarks while minimizing costs, such as fabrication or operating cost. In the case of DSAs like Google’s TPUs, many of the principles and experiences from decades of building general-purpose CPUs change or do not apply. For example, here are features of the inference TPU (TPUv1) and the training TPU (TPUv2) share but are uncommon in CPUs:

- 1–2 large cores versus 32–64 small cores in server CPUs.
- The computational heavy lifting is handled by two-dimensional (2D)
DNN (Deep Neural Network) wisdom is that bigger machines lead to bigger breakthroughs.

and if we were building an inference accelerator, we could stop there. For training, this is less than a third of the story. SGD next measures the difference or error between the model’s result and the known good result from the training set using a loss function. Then back-propagation runs the model in reverse, layer-by-layer, to produce a set of error/loss values for each layer’s output. These losses measure the deviation from the desired output. Last, weight update combines the input of each layer with the loss value to calculate a set of deltas—changes to weights—which, when added to the weights, would have resulted in nearly zero loss. Updates can have small magnitude. Shrinking further, updates are scaled down by the learning rate to keep SGD numerically stable. Moreover, a suite of algorithmic refinements—including momentum,\(^\text{30}\) batch normalization,\(^\text{31}\) and optimizers such as Adaptive Gradient (AdaGrad)\(^\text{14}\)—require their own state and alter the SGD algorithm to reduce the number of steps to achieve desired accuracy.

Each SGD step makes a tiny adjustment to the weights that improves the model with respect to a single \((\text{input, result})\) pair. Each pass through the entire dataset is an \textit{epoch}; DNNs typically take tens to hundreds of epochs to train. SGD gradually transforms the random initial weights into a trained model, sometimes capable of superhuman accuracy.

Given this background, we can compare inference and training. Both share some computational elements including matrix multiplications, convolutions, and activation functions, so inference and training DSAs might have similar functional units. Key architectural aspects where the requirements differ include:

\begin{itemize}
  \item \textbf{Harder parallelization:} Each inference is independent, so a simple cluster of servers with DSA chips can scale up inference. A training run iterates over millions of examples, coordinating across parallel resources because it must produce a single consistent set of weights for the model. The number of examples processed in parallel, and the time to evaluate that multiple-example \textit{minibatch}—often shortened to \textit{batch}—directly affect total end-to-end training time. A \textit{step} is the computation to process one minibatch.
\end{itemize}
More computation: Back-propagation requires derivatives for every computation in a model. It includes activation functions (some of which are transcendental), and multiplication by transposed weight matrices.

More memory: Weight update accesses intermediate values from forward and back propagation, vastly upgrading storage requirements; temporary storage can be 10x weight storage. For inference, a small activation working set can usually be kept on chip.

More programmability: Training algorithms and models are continually changing, so a machine restricted to current best-practice algorithms during design could rapidly become obsolete.

Wider data: Quantized arithmetic—a 8-bit integer instead of 32-bit floating point (FP)—can work for inference like in TPUv1 but reduced-precision training is an active research area. The challenge is sufficiently capturing the SGD sum of many small weight updates to preserve the accuracy of using 32-bit FP arithmetic to train models.

After explaining the TPUv2 architecture, we describe the domain specific language (TensorFlow) and compiler (XLA) for TPUv2 and compare the architecture and technology choices for the TPUv2 versus a GPU, the most popular computer for DNN training. Later, we compare performance per chip and full supercomputers of TPs and GPUs using production applications and the MLPerf benchmarks.

Designing a Domain-Specific Supercomputer

In 2014, when the TPUv2 project began, the landscape for high-performance machine learning computation was very different from today. Training took place on clusters of CPUs. State-of-the-art parallel training used asynchronous SGD, in part to tolerate tail latencies in shared clusters. Parallel training also divided CPUs into a bipartite graph of workers (running the SGD loop) and parameter servers (hosting weights and adding updates to them).

The DNN training computation appetite appeared unlimited. (Indeed, the computation requirements for the largest training runs grew 10x annually from 2012 to 2018.) Thus, in 2014 we chose to build a DSA supercomputer in stead of clustering CPU hosts with DSA chips. The first reason is that training time is huge. Table 1 shows that one TPUv2 chip would take two to 16 months to train a single Google production application, so a typical application might want to use hundreds of chips. Second, DNN wisdom is that bigger datasets plus bigger machines lead to bigger breakthroughs. Moreover, results like AutoML use 50x more computation to find DNN models that achieve higher accuracy scores than the best models of human DNN experts.

Designing a DSA supercomputer interconnect. The critical architecture feature of a modern supercomputer is how its chips communicate: what is the speed of a link; what is the interconnect topology; does it have centralized versus distributed switches; and so on. This choice is much easier for a DSA supercomputer, as the communication patterns are limited and known. For training, most traffic is an all-reduce over weight updates from all nodes of the machine.

If we distribute switch functionality into each chip rather than as a standalone unit, the all-reduction can be built in a dimension-balanced, bandwidth-optimal way for a 2D torus topology (see Figure 1). An on-device switch provides virtual-circuit, deadlock-free routing. To enable a 2D torus, the chip has four custom Inter-Core Interconnect (ICI) links, each running at 496Gbits/s per direction in TPUv2. ICI enables direct connections between chips to form a supercomputer using only 13% of each chip (see Figure 3). Direct links simplify rack-level deployment, but in a multi-rack system the racks must be adjacent.

One measure of an interconnect is its bisection bandwidth—the bandwidth

Table 1. Days to train production programs on one TPUv2 chip.

<table>
<thead>
<tr>
<th>MLP0</th>
<th>MLP1</th>
<th>CNN0</th>
<th>CNN1</th>
<th>RNN0</th>
<th>RNN1</th>
</tr>
</thead>
<tbody>
<tr>
<td>475</td>
<td>117</td>
<td>63</td>
<td>115</td>
<td>77</td>
<td>147</td>
</tr>
</tbody>
</table>

Figure 1. A 2D-torus topology. TPUv2 uses a 16x16 2D torus.
available between two halves of a network of the worst-case split. The TPUv2 supercomputer uses a 16x16 2D torus (256 chips), which is 32 links x 496 Gbits/s = 15.9 Terabits/s of bisection bandwidth. As a comparison, a separate Infiniband switch (used in CPU clusters) that connected 64 hosts (each with, say, four DSA chips) has 64 ports using “only” 100 Gbit/s links and a bisection bandwidth of at most 6.4 Terabits/s. Our TPUv2 supercomputer provides 2.5x the bisection bandwidth over conventional cluster switches while skipping the cost of the Infiniband network cards, Infiniband switch, and the communication delays of going through the CPU hosts of clusters.

Fortuitously, building a fast interconnect inspired algorithmic advances. With dedicated hardware, and sharding the examples of a minibatch over nodes of the machine, there is little tail latency, and synchronous parallel training becomes possible. Internal studies suggested that synchronous training could beat asynchronous SGD with equivalent resources. Asynchronous training introduces heterogeneity plus parameter servers that eventually limit parallelization, as the weights get sharded and the bandwidth from parameter servers to workers becomes a bottleneck. Synchronous training eliminated the parameter servers allowing peer-to-peer among workers, using the all-reduce to ensure workers begin and end each parallel step with consistent copies of weights.

Synchronous training has two phases in the critical path—a compute phase and a communication phase that reconciles the weights across learners. The slowest learners and slowest messages through the network limit performance of such a synchronous system. Since the communication phase is in the critical path, a fast interconnect that quickly reconciles weights across learners with well-controlled tail latencies is critical for fast training. The ICI network is key to the excellent TPU supercomputer scaling results; later we show 96%–99% of perfect linear scaleup.

**Designing a DSA supercomputer node.** The TPUv2 node of the supercomputer followed the main ideas of TPUv1: A large two-dimensional matrix multiply unit (MXU) using a systolic array to reduce area and energy plus large, software-controlled on-chip memories instead of caches. The large MXUs of the TPUs rely on large batch sizes, which amortize memory accesses for weights—performance often increases when memory traffic reduces.

Shallue et al. examined the effect of increasing batch size on training time, and found three regions for all models (as seen in Table 2): 1. **Perfect scaling region:** Each doubling of batch size halves the number of training steps. 2. **Diminishing returns region:** Increasing batch size still reduces the number of steps, but more slowly. 3. **Maximum data parallelism region:** Increasing batch size provides no benefits whatsoever.

Such scaling while preserving accuracy required tuning the learning rate, batch size, and other hyperparameters.

Fortunately for TPUs, these recent results show that batch sizes of 256–8,192 scale perfectly without losing accuracy, which makes large MXUs an attractive option for high performance.

Unlike TPUv1, TPUv2 uses two cores per chip. Global wires on a chip don’t scale with shrinking feature size, so their relative delay increases. Given that training can use many processors, two smaller TensorCores per chip prevented the excessive latencies of a single large full-chip core. We stopped at two because it is easier to efficiently generate programs for two brawny cores per chip than numerous wimpy cores.

Figure 2 shows the six major blocks of a TensorCore and Figure 3 shows their placement in the TPUv2 chip: 1. **Inter-Core Interconnect (ICI).** Explained earlier.

2. **High Bandwidth Interconnect (HBM).** TPUv1 was memory bound for most of its applications. We solved its memory bottleneck by using High Bandwidth Memory (HBM) DRAM in TPUv2. It offers 20 times the bandwidth of TPUv1 by using an interposer substrate that connects the TPUv2 chip via thirty-two 128-bit buses to four short stacks of DRAM chips. Conventional servers support many more DRAM chips, but at a much lower bandwidth of at most eight 64-bit busses.

3. **The Core Sequecer fetches VLIW (Very Long Instruction Word) instructions from the core’s on-chip, software-managed Instruction Memory (Imem), executes scalar operations using a 4K 32-bit scalar data memory (Smem) and 32 32-bit scalar registers (Sregs), and forwards vector instructions to the VPU. The 32-bit VLIW instruction can launch eight operations: two scalar, two vector ALU, vector load and store, and a pair of slots that queue data to and from the matrix

<table>
<thead>
<tr>
<th>Model</th>
<th>Perfect</th>
<th>Diminishing</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformer on LM1B</td>
<td>≤256</td>
<td>256–4096</td>
<td>≥4096</td>
</tr>
<tr>
<td>Simple CNN on Fashion MNIST</td>
<td>≤512</td>
<td>512–2048</td>
<td>≥2048</td>
</tr>
<tr>
<td>ResNet-50 on Imagenet</td>
<td>≤8192</td>
<td>8192–65536</td>
<td>≥65536</td>
</tr>
</tbody>
</table>

Table 2. Batch sizes for the three regions of Shallue. LM1B, Fashion MNIST, and Imagenet are standard DNN datasets.
multiply and transpose units. The XLA compiler schedules loading Imem via independent overlays of code, as unlike conventional CPUs, there is no instruction cache.

4. The Vector Processing Unit (VPU) performs vector operations using a large on-chip vector memory (Vmem) with 32K 128 × 32-bit elements (16MiB), and 32 2D vector registers (Vregs) that each contain 128 × 8 32-bit elements (4 KiB). The VPU streams data to and from the MXU through decoupling FIFOs. The VPU collects and distributes data to Vmem via data-level parallelism (2D matrix and vector functional units) and instruction-level parallelism (8 operations per instruction).

Your beautiful DSA can fail if best-practice algorithms change, rendering it prematurely obsolete. We handled such a crisis in 2015 during our design in supporting batch normalization. Briefly, batch normalization subtracts out the mean and divides by the standard deviation of a batch, making the values look like samples from the normal distribution. In practice, it both improves prediction accuracy and reduces time-to-train up to 14×! Batch normalization emerged early in 2015, and the results made it a must-do for us. We divided it into vector additions and multiplications over the batch, plus one inverse-square-root calculation. However, the vector operation count was high. We thus added a second SIMD dimension to our vector unit, making its registers and ALUs 128x8 (rather than just 1D 128-wide) and adding an inverse square root operation to the transcendental unit.

5. The MXU produces 32-bit FP products from 16-bit FP inputs that accumulate in 32 bits. All other computations are in 32-bit FP except for results going directly to an MXU input, which are converted to 16-bit FP.

The MXUs are large, but we reduced their size from 256x256 in TPUv1 to 128x128 and have multiple MXUs per chip. The bandwidth required to feed and obtain results from an MXU is proportional to its perimeter, while the computation it provides is proportional to its area. Larger arrays provide more compute per byte of interface bandwidth, but larger arrays can be inefficient. Simulations show that convolutional model utilization of

Figure 3. TPUv2 chip floor plan.

It has two TensorCores: Node fabric data and NF controller move on-chip data.
four 128x128 MXUs is 37%–48%, which is 1.6x of a single 256x256 MXU (22%–30%) yet take about the same die area. The reason is that some convolutions are naturally smaller than 256x256, so sections of the MXU would be idle. Sixteen 64x64 MXUs would have a little higher utilization (38%–52%) but would need more area. The reason is the MXU area is determined either by the logic for the multipliers or by the wires on its perimeter for the inputs, outputs, and control. In our technology, for 128x128 and larger the MXU’s area is limited by the multipliers but area for 64x64 and smaller MXUs is limited by the I/O and control wires.

6. The Transpose Reduction Permute Unit does 128x128 matrix transposes, reductions, and permutations of the VPU lanes.

Alternative DSA supercomputer architecture. Peak performance is ≥8x higher when using 16-bit FP instead of 32-bit FP for matrix multiply (see Table 3), so it’s vital to use 16-bit to get highest performance. While we could have built an MXU using standard IEEE fp16 and fp32 floating point formats (see Figure 5), we first checked the accuracy of 16-bit operations for DNNs. We found that:

- Matrix multiplication outputs and internal sums must remain in fp32.
- The 5-bit exponent of fp16 matrix multiplication inputs leads to failure.

**Table 3. Key processor features.**

<table>
<thead>
<tr>
<th>Feature</th>
<th>TPUv1</th>
<th>TPUv2</th>
<th>TPUv3</th>
<th>Volta</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peak TeraFLOPS/Chip</td>
<td>92 (8b int)</td>
<td>46 (16b)</td>
<td>123 (16b)</td>
<td>125 (16b)</td>
</tr>
<tr>
<td>Network links x Gbits/s/Chip</td>
<td>--</td>
<td>4 x 496</td>
<td>4 x 656</td>
<td>6 x 200</td>
</tr>
<tr>
<td>Max chips/supercomputer</td>
<td>--</td>
<td>256</td>
<td>1024</td>
<td>Varies</td>
</tr>
<tr>
<td>Peak PetaFLOPS/supercomputer</td>
<td>--</td>
<td>11.8</td>
<td>126</td>
<td>Varies</td>
</tr>
<tr>
<td>Bisection Terabits/supercomputer</td>
<td>--</td>
<td>15.9</td>
<td>42.0</td>
<td>Varies</td>
</tr>
<tr>
<td>Clock Rate (MHz)</td>
<td>700</td>
<td>700</td>
<td>940</td>
<td>1530</td>
</tr>
<tr>
<td>TDP (Watts)/Chip</td>
<td>75</td>
<td>280</td>
<td>450</td>
<td>450</td>
</tr>
<tr>
<td>TDP (Kwatts)/supercomputer</td>
<td>--</td>
<td>124</td>
<td>594</td>
<td>Varies</td>
</tr>
<tr>
<td>Die Size (mm²)</td>
<td>&lt;331</td>
<td>&lt;611</td>
<td>&lt;648</td>
<td>815</td>
</tr>
<tr>
<td>Chip Technology</td>
<td>28nm</td>
<td>&gt;12nm</td>
<td>&gt;12nm</td>
<td>12nm</td>
</tr>
<tr>
<td>Memory size (on-/off-chip)</td>
<td>28MiB/8GiB</td>
<td>32MiB/16GiB</td>
<td>32MiB/32GiB</td>
<td>36MiB/32GiB</td>
</tr>
<tr>
<td>Memory GB/s/Chip</td>
<td>34</td>
<td>700</td>
<td>900</td>
<td>900</td>
</tr>
<tr>
<td>MXUs/Core, MXU Size</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Cores/Chip</td>
<td>256x256</td>
<td>128x128</td>
<td>128x128</td>
<td>4x4</td>
</tr>
<tr>
<td>Chips/CPU Host</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>80</td>
</tr>
</tbody>
</table>

**Figure 4. A TPUv2 supercomputer has up to 256 chips and is 18-ft. long (top).**

A TPUv3 supercomputer consisting of up to 1,024 chips (below) is about 7-ft. tall and 36-ft. long. A TPUv2 board (center) holds four air-cooled chips and a TPUv3 board (right) also has four chips but uses liquid cooling.
of computations that go outside its narrow range, which the 8-bit exponent of fp32 avoids.

- Reducing the matrix multiplication input mantissa size from fp32’s 23 bits to 7 bits did not hurt accuracy.

The resulting brain floating format (bf16) in Figure 5 keeps the same 8-bit exponent as fp32. Given the same exponent size, there is no danger in losing the small update values due to FP underflow of a smaller exponent, so all programs in this article used bf16 on TPUs without much difficulty. Beyond our experience that it works for training production applications, a recent Intel study corroborated its benefits. However, fp16 requires adjustments to training software (loss scaling) to deliver convergence and efficiency. It preserves the effect from small gradients by scaling losses to fit the smaller exponents of fp16.

As the size of an FP multiplier scales with the square of the mantissa width, the bf16 multiplier is half the size and energy of a fp16 multiplier: $B/112 \approx 0.5$ (accounting for the implicit leading mantissa bit). Bf16 delivers a rare combination: reducing hardware and energy while simplifying software by making loss scaling unnecessary. Thus, ARM and Intel have revealed future chips with bf16.

### Designing a DSA

#### Supercomputer Compiler

The next step was getting software for our hardware. To program CPUs and GPUs for machine learning, a framework such as TensorFlow (TF) specifies the model and data operations machine-independently. TF is a domain-specific library built on Python. NVIDIA GPU-dependent work is supported by a combination of the CUDA language, the CuBLAS and CuDNN libraries, and the TensorRT system. TPUv2/v3s also use TF, with the new system XLA (for accelerated linear algebra) handling the TPU-dependent mapping. XLA also targets CPUs and GPUs. Like many systems that map from domain-specific languages to code, XLA integrates a high-level library and a compiler. A TF front end generates code in an intermediate representation for XLA.

It would seem it should be more difficult to get great performance in a programming system based on Python like TF. However, ML frameworks offer both a higher level of expressiveness and the potential for much better optimization information than lower-level languages like C++. TF programs are graphs of operations, where multi-dimensional array operations are first-class citizens:

- They operate on multi-dimensional arrays explicitly, rather than implicitly via nested loops as in C++.
- They use explicit, analyzable, and bounded data access patterns versus arbitrary access patterns like C++.
- They have known memory aliasing behavior, unlike C++.

These three factors allow the XLA compiler to safely and correctly transform programs in ways that traditional compilers rarely attain.

XLA does whole-program analysis and optimization. With 2D vector registers and compute units in TPUv2/v3, the layout of data in both compute units and memory is critical to performance, perhaps more than for a vector or SIMD processor. Building efficient code for vector machines, with 1D memory and compute units, is well understood. For the MXU, two 2D inputs interact to produce a 2D output. Each operand has a memory layout, which gets transformed into a layout in 2D registers, which in turn must be fed at the exact moment to meet systolic array timing in the MXU. (A systolic array reduces register accesses by choreographing data flowing from different directions to regularly arrive at cross points that combine them.) Depending on layout choices, the 2D registers dimensions of 128 and 8 might not be filled, lowering ALU and memory utilization. Moreover, lacking caches, XLA manages all memory transfers, including code overlays and DMA pushes to remote nodes over ICI.

XLA exploits the huge parallelism that an input TF dataflow graph represents. Beyond the parallelism of operations (“ops”) in a graph, each op can comprise millions of multiplications and additions on data tensors of millions of elements. XLA maps this abundant parallelism across hundreds of chips in a supercomputer, a few cores per chip, multiple units per core, and thousands of multipliers and adders inside each functional unit. The domain-specific TF language and XLA representation allow precise reasoning about memory use at every point in the program. There are no “aliasing” issues where the compiler must determine whether two pointers might address the same memory—every piece of memory cor-
respects to a known program variable or temporary. The XLA compiler is free to slice, tile, and lay out memory and operations to best use the on-chip memory bandwidth and to reduce the memory footprint on chip or off chip.

TPUs use a VLIW architecture to express instruction-level parallelism to the many compute units of a TensorCore. XLA uses standard VLIW compilation techniques including loop unrolling, instruction scheduling, and software pipelining to keep all compute units busy and to simultaneously move data through the memory hierarchy to feed them.

Given a memory layout of data, operator fusion can reduce memory use and boost performance. Fusion is a traditional compiler optimization—but applied now to 2D data—that combines ops to reduce memory traffic compared to executing operators sequentially. For example, fusing a matrix multiplication with a following activation function skips writing and reading the intermediate products from memory. Table 4 shows the speedup from the fusion optimization on 2D data from 1.8 to 6.3.

The TF intermediate form for XLA has thousands of ops. The number of ops increases when programmers cannot combine existing ops if composition is inefficient. Alas, expanding the number of ops is an engineering challenge, since software libraries need to be developed for CPUs, GPUs, and TPUs. The hope was that the XLA compiler could synthesize these thousands of ops from a smaller set of primitive ops.

The XLA team needed only 96 ops as the compiler’s target to reduce work for the library/compiler by enhancing composability. For example, XLA has a single op for convolution (kConvolution) letting the compiler handle all the memory layout variations. The TF intermediate form has nine; for example, Conv2D, Conv2DBackpropFilter, DepthwiseConv2dNative, and DepthwiseConv2dNativeBackpropFilter. For the CNN1 program, the XLA compiler fused 63 different operations with at least one kConvolution.

Since ML platforms and DSAs offered a new set of compiler challenges, it was unclear how fast they would improve. Table 5 shows the median gain over only six months for MLPerf from version 0.5 to 0.6 was 1.3x for GPUs and 2.1x for TPUs! (Perhaps the younger XLA compiler has more opportunity to improve than the more mature CUDA stack.) One reason for the large gain is the focus on benchmarks, but production applications also advanced. Increasing bf16 use, optimizing model architecture, and XLA generating better code sped up CNN0 by 1.8x in 15 months and improving partitioning/placement for embeddings and XLA optimizations accelerated MLP0 by 1.65x.

### Contrasting GPU and TPU Architectures

As details of TPU and GPU architectures are now public, let us compare TPU and GPU choices before we compare performance.

Multi-chip parallelization is built into TPUs through ICI and supported through all-reduce operations plumbed through XLA to TF. Similar-sized multi-chip GPU systems use a tiered networking approach, with NVIDIA’s NVLink inside a chassis and host-controlled InfiniBand networks and switches to tie multiple chassis together.

TPUs offer bf16 FP arithmetic designed for DNNs inside 128x128 systolic arrays that halves the die area and energy versus IEEE fp16 FP multipliers. Volta GPUs have also embraced reduced-precision systolic arrays, with a finer granularity—4x4 or 16x16 depending on hardware or software descriptions—while using fp16 rather than bf16, so they may require software to perform loss scaling plus extra die area and energy.

TPUs are dual-core, in-order machines, where the XLA compiler overlaps computation, memory, and network activities. GPUs are latency-tolerant many-core machines, where each core has many threads and thus very large (20MiB) register files. Threading hardware plus CUDA coding conventions support overlapped operations.

TPUs use software controlled 32MiB scratchpad memories that the compiler schedules, while Volta hardware manages a 6MiB cache and software manages a .5MiB scratchpad memory. The XLA compiler directs sequential DRAM accesses typical of DNNs via direct memory access (DMA) controllers on TPUs while GPUs use multithreading plus coalescing hardware for them.

Thottethodi and Vijaykumar concluded that when compared to TPUs: “[GPUs] incur high overhead in performance, area, and energy due to heavy multithreading which is unnecessary for DNNs which have prefetchable, sequential memory accesses. The systolic organization [of TPUs] ... capture[s] DNNs’ data reuse while being simple by avoiding multithreading.”

In addition to the contrasting architectural choices, TPU and GPU chips use different technologies, die areas, clock rates, and power. Table 6 gives three related cost measures of these systems: approximate die size adjusted for technology; power for a 16-chip
system; and cloud price per chip. The GPU adjusted die size is more than twice that of the TPUs, which suggests the capital costs of the chips is at least double, since there would be at least twice as many TPU dies per wafer. GPU power is 1.3x–1.6x higher, which suggests higher operating expenses, as the total cost of ownership is correlated with power. Finally, the hourly rental prices on Google Cloud Engine are 1.6x–2.9x higher for the GPU. These three different measures consistently suggest TPUv2 and TPUv3 are roughly half to three fourths as expensive as the Volta GPU.

Performance Evaluation

In computer architecture, we “grade on a curve” versus “grade on an absolute scale,” so we need to measure performance relative to the competition. Before showing performance of TPU supercomputers, we must establish the virtues of a single chip, for a 1024x speedup from 1,024 wimpy chips is uninteresting.

We first compare training performance for a standard set of ML benchmarks and Google production applications for TPUv2/v3 chip and the Volta GPU chip; TPUv3 and Volta are about the same speed. We then check if four MXUs per chip in TPUv3 really helped, or if other bottlenecks in the TPUv3 chip made the extra MXUs superfluous; they helped! We conclude the chip comparison looking at inference for TPUv2/v3 versus TPUv1; TPUv2/v3 are much faster.

Having established the merits of the TPU chips, we then evaluate the TPUv2/v3 supercomputer. The first step is to see how well it scales; we see 96%–99% of perfect linear speedup at 1024 chips. We then compare the fraction of peak performance and performance per Watt of TPU and traditional supercomputers; TPs have 5x-10x better performance per Watt.

Chip performance: TPUv2/v3 versus the Volta GPU

Figure 6 shows the performance of TPUv3 and the Volta GPU over TPUv2 for two sets of programs. The first set is five programs that Google and NVIDIA both submitted to MLPerf 0.6 in May 2019, and both use 16-bit multiplication with NVIDIA software performing loss scaling. The geometric mean speedup of these programs over TPUv2 is 1.8 for TPUv3 and 1.9 for Volta.

We also wanted to measure performance of production workloads. We chose six production applications similar to what we used for TPUv1 as representative of Google’s workload:

- In MultiLayer Perceptrons (MLP) each new layer of a model is a set of nonlinear functions of a weighted sum of all outputs (fully connected) from a prior one. This classic DNN usually has text as input. MLP0 is unpublished but MLP1 is RankBrain, which ranks search results for a Web page.
- In Convolutional Neural Networks (CNN), each ensuing layer is a set of nonlinear functions of weighted sums of spatially nearby subsets of outputs from the prior layer. CNNs usually have images as inputs. CNN0 is AlphaZero, a reinforcement learning algorithm with extensive use of CNNs, which mastered the games chess, Go, and shogi. CNN1 is a Google-internal model for image recognition.
- In Recurrent Neural Networks (RNN), each subsequent model layer is a collection of nonlinear functions of weighted sums of outputs and the previous state. Sequence prediction problems, such as language translation, use RNNs. RNN0 is RNMT+ and RNN1 is Improved LAS.

We recently compared the representative datacenter workloads by model type for inference on TPUv1 versus TPUv2/v3 for training. Table 7 illustrates the fast-changing nature of DNNs. We originally used the name LSTM (Long Short-Term Memory) for TPUv1 applications, a type of RNN. Although sampled three years apart—July 2016 versus April 2019—we were still surprised that CNNs were a much larger part of datacenter training, and that a new model Transformer published the year that TPUv2 was de-
employed—was as popular as RNNs. (Transformer is part of MLPerf 0.5.)

Transformer is intended for the same tasks as RNNs, such as translation, but is considerably faster since it lends itself to parallelization while RNNs have sequential dependencies. The layers of Transformer are a mix of MLPs and attention layers. Attention is the key new mechanism used in Transformer; it lets neural networks look up data associatively, in a memory-like structure whose indices themselves are learned. The components of attention resemble those of other layers, including matrix multiplications and dot products, which map well to TPU hardware. One difference is that attention matrices grow with sequence length, adding dynamic shape and memory requirements that complicate some optimizations done by XLA. The success of this recent model (see Figure 6) highlights TPU programmability.

The geometric mean speedup of the six production applications was 1.8 for TPUv3 but only 0.4 for Volta, primarily because they use 8x slower fp32 on GPUs instead of fp16 (Table 3). These are large production applications that are continuously improved, and not simple benchmarks, so it’s a lot of work to get them to run at all, and more to run well. As noted earlier, application programmers focus on TPUs, since they are in everyday use, so there is little urge to include loss scaling needed for fp16. (TF kernels for embeddings have not been developed for GPUs, so we exclude MLPs from the GPU geometric mean as they could not run.)

**Is TPUv3 memory bound or compute bound?** While the peak compute improvement of TPUv3 over TPUv2 is 2.7x, the improvements in memory bandwidth, ICI bandwidth, and clock rate are only ≈1.35x. We wondered whether the extra MXUs in TPUv3 would be underutilized due to bottlenecks elsewhere. Figure 6 shows that one production application runs a bit higher than the memory improvement at 1.4x, but the other five and all the MLPerf 0.6 benchmarks run much faster at 1.6x to 2.3x. The large application batch sizes and sufficient on-chip storage enabled these good results. As the MXUs are not a large part of the chip (Figure 3), doubling the MXUs in TPUv3 clearly proved beneficial.

![Supercomputer scaling: TPUv3 and Volta.](image)

**Table 8. Days to train MLPerf 0.5 benchmarks on one TPUv2 chip. See Table 1 for time to train production applications.**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SSD</th>
<th>Mask R-CNN</th>
<th>GNMT</th>
<th>Transformer</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet50</td>
<td>0.8</td>
<td>1.9</td>
<td>0.2</td>
<td>0.3</td>
</tr>
</tbody>
</table>

Inference on a training chip: TPUv2/v3 vs. TPUv1. What about inference speed? Running it on a training chip—which works since it is like the forward pass—could help applications that require frequent training on fresh data. TPUv2/v3 do not support 8-bit integer data types, so inference uses bf16. One upside of using the same arithmetic for training and inference is that ML experts don’t need to do extra work—called quantization—to ensure the same accuracy of the DNN model.

One danger is the larger batch sizes needed to run efficiently on TPUv2/v3 could hurt inference latency. Fortunately, we have DNN models that can meet their latency targets with batch sizes of greater than 1,000. With billions of daily users, inferences per second across the whole data center fleet can be very high.

The LSTM0 benchmark, for instance, ran at 48 inferences per second with a response time of 122ms on TPUv1. TPUv2 runs it 5.6x as fast with a 2.8x lower response time (44ms) at the same batch size. The lower latency in turn allows for larger batches compared to TPUv1 to be served in production yet still meet latency targets. With larger batches, the throughput rose to 11x with a latency improvement of 2x (58ms) vs TPUv1. TPUv3 reduces latency 1.3x (45ms) versus TPUv2 at the same batch size.

**DSA supercomputer scaling performance.** Alas, only ResNet-50 from MLPerf 0.6 can scale beyond 1,000 TPUs and GPUs. Figure 7 shows three ResNet-50 results. Ying et al. published a ResNet-50 results on TPUv3 that delivered 77% of perfect linear scaleup at 1,024 chips, but the TPUv3 version for MLPerf 0.6 only runs at 52%. The difference is in MLPerf’s ground rules. MLPerf requires including evaluation in the training time. (Evaluation runs a holdout dataset after a model training finishes to determine its accuracy.) Like Ying et al., most researchers exclude it when reporting performance. More unusually, MLPerf requires running evaluation at the end of every four epochs to deter benchmark cheating. ML developers would never evaluate that frequently. For MLPerf 0.6, NVIDIA ran ResNet-50 on a cluster of 96 DGX-2H each with 16 Voltas connected via Infiniband switches at 41% of linear scale-up for 1,536 chips.
Table 9. Traditional versus TPU supercomputer Top500 and Green500 rank (June 2019) for Linpack and AlphaZero.

<table>
<thead>
<tr>
<th>Name</th>
<th>Cores</th>
<th>Benchmark</th>
<th>Data</th>
<th>Peta Flop/s</th>
<th>% of Peak</th>
<th>Mega-watts</th>
<th>GFlop/Watt</th>
<th>Top 500</th>
<th>Green 500</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tianhe</td>
<td>4885k</td>
<td>Linpack</td>
<td>32/64 bit</td>
<td>61.4</td>
<td>61%</td>
<td>18.48</td>
<td>3.3</td>
<td>4</td>
<td>57</td>
</tr>
<tr>
<td>SaturnV</td>
<td>22k</td>
<td>Linpack</td>
<td>32/64 bit</td>
<td>1.1</td>
<td>59%</td>
<td>0.97</td>
<td>5.1</td>
<td>469</td>
<td>1</td>
</tr>
<tr>
<td>ABCI</td>
<td>392k</td>
<td>Linpack</td>
<td>32/64 bit</td>
<td>19.9</td>
<td>61%</td>
<td>1.65</td>
<td>14.4</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>TPUv2</td>
<td>0.5k</td>
<td>AlphaZero</td>
<td>16/32 bit</td>
<td>9.9</td>
<td>84%</td>
<td>0.12</td>
<td>79.9</td>
<td>22</td>
<td>2</td>
</tr>
<tr>
<td>TPUv3</td>
<td>2k</td>
<td>AlphaZero</td>
<td>16/32 bit</td>
<td>86.0</td>
<td>70%</td>
<td>0.59</td>
<td>146.3</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

See article for caveats about comparing Linpack on 64-bit floating point to ML training on 16-bit floating point.

Table 10. Time to train supercomputers from NVIDIA, Fujitsu, and Google on the ResNet-50 benchmark from MLPerf 0.6.

<table>
<thead>
<tr>
<th></th>
<th>NVIDIA cluster</th>
<th>ABCI Supercomputer</th>
<th>TPUv3 Supercomputer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformer</td>
<td>80 seconds</td>
<td>70 seconds</td>
<td>77 seconds</td>
</tr>
<tr>
<td>MLP</td>
<td>1536 Voltas + 192 CPUs</td>
<td>2048 Voltas + 1024 CPUs</td>
<td>1024 TPUv3s + 128 CPUs</td>
</tr>
</tbody>
</table>
gains. Anton systems\textsuperscript{11} showed two order-of-magnitude speedups over traditional supercomputers on molecular dynamics workloads. They also resulted from hardware/software/algorithms codesign, with custom chips, interconnect, and arithmetic.

**Conclusion**

Benchmarks suggest the TPUv3 chip performs similarly to the contemporary Volta GPU chip, but parallel scaling for production applications is stronger for the TPUv3 supercomputer:

- Three scale to 1,024 chips at 99% linear speedup;
- One scales to 1,024 chips at 96% linear speedup; and
- Two scale to 1,024 chips but are limited by embeddings.

Remarkably, a TPUv3 supercomputer runs a production application using real-world data at 70% of peak performance, higher than general-purpose supercomputers run the Linpack benchmark using weak scaling of manufactured data. Moreover, TPU supercomputers with 256–1,024 chips running a production application have 5x–10x performance/Watt of the #1 traditional supercomputer on the Green500 list running Linpack and 24x–44x of the #4 supercomputer on the Top500 list. Reasons for this success include the built-in ICI network, large systolic arrays, and bfi\textsubscript{16} arithmetic, which we expect will become a standard data type for DNN DSAs.

TPUv2/v3 have smaller dies in an old semiconductor process and lower Watt of the #1 traditional supercomputer, Remarkably, a TPUv3 supercomputer on the Green500 list running Linpack is cost-effective and can deliver high arithmetic.

**References**


Copyright held by authors/owners.

Norman P. Jouppi is a Distinguished Hardware Engineer at Google, Mountain View, CA, USA.

Doe Hyun Yoon is a staff software engineer at Google, Mountain View, CA, USA.

George Kurian is a senior software engineer at Google, Mountain View, CA, USA.

Sheng Li is a senior staff software engineer and tech lead on ML Accelerator Optimization at Scale at Google, Mountain View, CA, USA.

Nishant Patil is a senior staff software engineer at Google, Mountain View, CA, USA.

James Laudon is an engineering director at Google, Mountain View, CA, USA.

Cliff Young is a software engineer at Google, Mountain View, CA, USA.

David Patterson is a Distinguished Engineer at Google, Mountain View, CA, USA, a professor of Graduate School at the University of California, Berkeley, CA, USA, and Director of the NSF-funded International Open Source Laboratory at Berkeley, CA, and Shenzhen, China.