PNAS RESEARCH ARTICLE | COMPUTER SCIENCES

Large-scale distributed linear algebra with tensor processing units

Adam G. M. Lewis[a,b,1], Jackson Beall[a,b], Martin Ganahl[a,b], Markus Hauru[b], Shrestha Basu Mallick[b], and Guifre Vidal[b,c]

Edited by David Weitz, Harvard University, Cambridge, MA; received December 16, 2021; accepted June 29, 2022

We have repurposed Google tensor processing units (TPUs), application-specific chips developed for machine learning, into large-scale dense linear algebra supercomputers. The TPUs' fast intercore interconnects (ICIs), physically two-dimensional network topology, and high-bandwidth memory (HBM) permit distributed matrix multiplication algorithms to rapidly become computationally bound. In this regime, the matrix-multiply units (MXUs) dominate the runtime, yielding impressive scaling, performance, and raw size: Operating in float32 precision, a full 2,048-core pod of third-generation TPUs can multiply two matrices with linear size $N = 2^{20} = 1{,}048{,}576$ in about 2 min. Via curated algorithms emphasizing large, single-core matrix multiplications, other tasks in dense linear algebra can similarly scale. As examples, we present 1) QR decomposition; 2) resolution of linear systems; and 3) the computation of matrix functions by polynomial iteration, demonstrated by the matrix polar factorization.

TPUs | scientific computation | linear algebra | distributed computing | ASICs

Significance

We demonstrate that Google TPUs, novel computer chips optimized for artificial intelligence, can also efficiently perform scientific computation at massive scale. The concrete examples of matrix multiplication, QR factorization, linear solution, and application of matrix functions are benchmarked upon matrices of linear size up to $N = 1{,}048{,}576$, achieving throughput around 90% of the single-core value at the largest values of N considered. We believe TPUs and TPU-like architectures will soon play a role in scientific computation analogous to the GPU but at larger scale, and mean for this study to catalyze the process.

Neural network inference and training requires low-precision multiplication of large matrices. To service this need, Google has reincarnated the systolic array as the tensor processing unit (TPU). As is typical of application-specific integrated circuits (ASICs), compared to CPUs at fixed wattage, TPUs sacrifice flexibility for speed: They, essentially, only multiply matrices, but are very good at doing so. We have measured distributed, single-precision (floating point 32 or fp32) matrix multiplication performance of around 21 peta floating-point operations per second (FLOPS)—competitive with an academic cluster allocation, while much more accessible and carbon friendly—on a third-generation TPU "pod" of 2,048 cores. But a means to harness such performance for scientific simulation is presently lacking. We are aware of two approaches to such a harness, which we view as complementary. The first (1–4) accelerates some CPU-based computation with some kind of TPU-based machine learning algorithm, for example, by using a neural network to precondition generalized minimal residual method (GMRES). The second, to which this paper contributes, curates traditional scientific algorithms to run efficiently on TPUs directly. Previous work by others in this vein has concerned discrete (5) and fast (6) distributed Fourier transforms, Monte Carlo simulation (7), and image processing (8). Our group's sister papers address quantum circuit simulation (9), many-body quantum physics (11, 12), electronic structure computation via density functional theory (DFT) (12), and coupled cluster (CC) methods (13), and tensor network algorithms such as the density matrix renormalization group (DMRG) (14).

This paper concerns the more foundational tasks of distributed dense linear algebra. While a single TPU core can already store and operate on large matrices (e.g., of size 16,384, 32,768 in single precision), the main advantage of TPUs is their ability to scale to full pods, which can handle much larger matrices (e.g., 1,048,576, 1,048,576, or 2,048× larger size).* Accordingly, our focus is on understanding how to perform distributed, multicore versions of linear algebra operations whose single-core version is already provided by the JAX library. Specifically, in this paper, we demonstrate four distributed dense linear algebra tasks at scale (see Fig. 1 for benchmarks):

1) distributed matrix multiplication, using the scalable universal matrix multiplication algorithm (SUMMA) (15) algorithm to translate from the TPUs' efficient single-core matrix multiplication to comparably-efficient, distributed matrix multiplication without data replication;
2) distributed QR decomposition, using an adapted communication-avoiding QR (CAQR) algorithm (16) emphasizing matrix multiplication;

*We consider TPU network topologies with a 2:1 aspect ratio, so that local blocks of square matrices have a corresponding 1:2 aspect ratio.

Author affiliations: [a]Simulation & Optimization Team, Sandbox AQ, Palo Alto, CA 94301; [b]Sandbox Alphabet X, The Moonshot Factory, Mountain View, CA 94043; and [c]Google Quantum AI, Google LLC, Santa Barbara, CA 93111

Author contributions: A.G.M.L., M.G., M.H., S.B.M., and G.V. designed research; A.G.M.L., J.B., M.G., M.H., S.B.M., and G.V. performed research; A.G.M.L. and G.V. analyzed data; A.G.M.L. wrote the paper; and A.G.M.L., J.B., M.G., and M.H. wrote code.

A.G.M.L., J.B., M.G., M.H., S.B.M., and G.V. performed this research while employees of Alphabet Inc, the proprietor of the TPU architecture.

[1]To whom correspondence may be addressed. Email: adam.lws@gmail.com.

Published August 8, 2022.

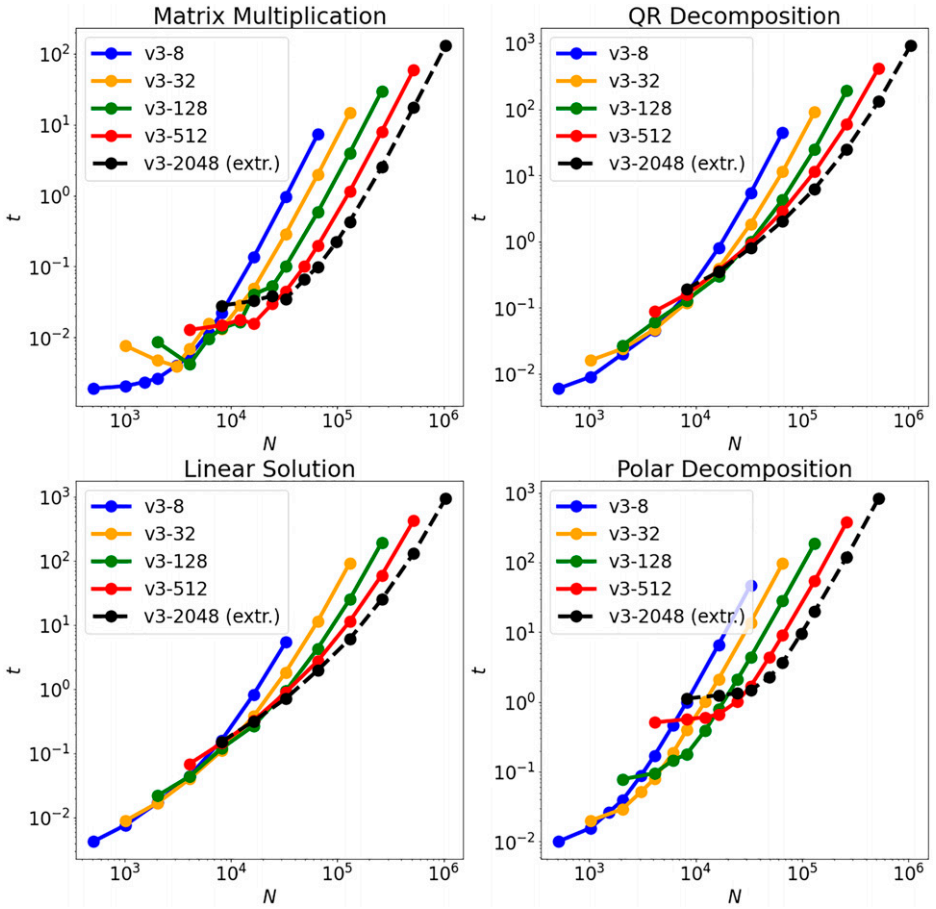

Fig. 1. Wall clock time t in seconds vs. linear size N of square input for (*Top Left*) matrix multiplication, (*Top Right*) QR decomposition, (*Bottom Left*) linear solution, and (*Bottom Right*) polar decomposition. The notation, e.g., v3-8, refers to a third-generation (v3) TPU network of eight cores (v3-8). The v3-2048 results are a linear extrapolation (extr.) from v3-512, necessitated by temporary resource constraints. As concrete examples, on a full TPUv3 pod (2,048 cores), working with dense (N, N) matrices of linear size $N = 2^{20} = 1{,}048{,}576$, a matrix multiplication takes about 2 min, whereas both a QR decomposition and solving a linear system take about 20 min; similarly, the polar decomposition takes about 20 min for linear size $N = 2^{19} = 524{,}288$. These and all experiments were conducted on TPU v3s accessed via Google Cloud. We used Python version 3.7, and JAX version 0.2.13 with Jaxlib version 0.1.67. None of our timings profile host–device communications.

3) solution of linear systems, implemented as a distributed QR decomposition followed by a distributed triangular solve; and
4) distributed computation of matrix functions—as a specific example, we show the polar decomposition, expressing a given matrix as the product of one unitary and one positive-semidefinite factor.

In sister papers, we use some of these tasks to accelerate and scale up a number of applications. For instance, a variant of SUMMA is used for CC computations (13); the distributed QR decomposition is used for DMRG (14); and distributed matrix functions similar to a polar decomposition, as well as the inverse square root, are used for the purification step of DFT (12).

Two remarks are in order. The results of this paper refer exclusively to single precision. However, TPUs can also perform linear algebra in (emulated) double precision, as needed, for example, in some quantum chemistry applications (12). For matrix multiplications, this incurs a roughly $11\times$ increase in computational cost. Moreover, we also note that, while the benchmark results presented here were obtained with third-generation TPUs (denoted TPUv3), fourth generation TPUs (denoted TPUv4) are already available. A TPUv4 pod (8,192 cores) can handle matrices with linear size $2\times$ larger than a TPUv3 pod.

The remainder of the paper is structured as follows. Section 1 briefly explicates the TPU architecture. Section 2 explains our approach to the four tasks listed above and presents benchmarks. Section 3 gives a closing discussion.

1. TPUs

Each TPUv3 chip has two cores, each equipped with two "matrix multiply units" (MXUs)—systolic arrays capable of multiplying two $(128, 128)$ matrices in 128 cycles. The chips are connected to one another via relatively fast interconnects, in a two-dimensional (2D) toroidal network ranging from 4 chips (8 cores) to 1,024 chips (2,048 cores) total, with each group of 4 chips (8 cores) controlled by a separate host CPU. See ref. 17 for many more details on the TPU architecture.

TPUs natively perform bf16-precision matrix multiplication with fp32 accumulation. That is, the TPU stores and sums data as fp32, but each individual matrix multiplication of a floating-point number is done in a specialized low-precision format called "brain float 16" or bf16, comparable to fp16 but with slightly more range and slightly less precision. The TPU can still operate in fp32 precision, however, via an internal mixed-precision algorithm which incurs a roughly $6\times$ penalty in compute time.

One generally programs for the TPU using XLA (Accelerated Linear Algebra) (18), an optimized graph compiler proprietary to Google. XLA translates from roughly C-like commands called High Level Operations (HLOs) to roughly assembly-like equivalents called Low Level Operations (LLOs). The HLOs themselves

may be written directly, but are usually instead "traced" from any of several higher-level languages. We used JAX (19), a NumPy-like interface to XLA.

The TPU architecture and its access via XLA introduces several constraints:

- Since XLA requires prior knowledge of memory boundaries, there is limited support for dynamical array shapes. All shapes must be computable from "static" data available at compile time, with changes to static data incurring an expensive re-compilation. This can complicate algorithms involving, for example, a shrinking block size.
- TPUs are optimized to perform large matrix multiplications. Thus, a relatively straightforward path to their efficient use is to find algorithms which also involve large matrix multiplications.
- TPUs store data in physically 2D memory, with each "row" able to store an 8 by 128 matrix panel. Matrices whose dimensions are not divisible by 8 or 128, respectively, are, in effect, zero-padded up to the next-largest sizes which are.

The subsequent discussion showcases a selection of distributed dense linear algebra algorithms that function well despite these constraints. We have chosen these specific algorithms because of their widespread use in scientific computing and/or their pivotal role in several applications in refs. 10–14 that were mentioned in the Introduction.

2. Distributed Linear Algebra Benchmarks

We target three "core" tasks representing essential computations of, for example, LAPACK:

- matrix multiplication: computation of $\mathbf{C}$ in $\mathbf{AB} = \mathbf{C}$ given matrices $\mathbf{A}$ and $\mathbf{B}$;
- QR factorization: computation of $\mathbf{Q}$ with orthonormal columns and upper-triangular $\mathbf{R}$ in $\mathbf{A} = \mathbf{QR}$ given $\mathbf{A}$; and
- linear solution: computation of $\mathbf{x}$ in $\mathbf{Ax} = \mathbf{b}$ given $\mathbf{A}$ and $\mathbf{b}$.

We also illustrate another task, for which TPUs turn out to be especially suitable:

- matrix functions: computation of $f(\mathbf{A})$, given a matrix $\mathbf{A}$ and a function $f(x)$, where $f(\mathbf{A})$ is a matrix obtained from $\mathbf{A}$ by transforming by f (depending on context) either its singular values or its eigenvalues.

We will illustrate matrix functions explicitly with the polar factorization, which can be construed as the case where f is the signum function acting on singular values, and thus mapping positive singular values to $+1$, and with matrix inversion.

A. Distributed Matrix Multiplication. The first step is to build large-scale matrix multiplication from fast local matrix multiplication and fast interchip communications over a 2D toroidal topology. This can be achieved using any of a variety of distributed matrix multiplication algorithms. We use SUMMA (15), whose memory footprint is tunable, and which straightforwardly handles transposed matrix multiplication.

SUMMA requires matrices be distributed across processors as 2D blocks. A group of p TPU cores is first divided into a (p_r, p_c) processor grid. An (M, N) matrix is then divided into $(m = M/p_r, n = N/p_c)$ blocks, and each block assigned to exactly one processor. The assignment must be "adapted" to the matrix, meaning:

- traversing through p_r or m with n and p_c fixed, also traverses through M with N fixed (row adapted), and
- traversing through p_c or n with m and p_r fixed, also traverses along N with M fixed (column adapted).

Though SUMMA does not require it, for simplicity, we furthermore adopt the checkerboard distribution illustrated in Fig. 2:

- m and p_r are contiguous in M, and
- n and p_c are contiguous in N.

We zero-pad as required when p_r does not evenly divide M, or p_c does not evenly divide N. Heuristically, the checkerboard distribution assigns matrix blocks to processors by overlaying the TPU grid ("checkerboard") atop the mathematical matrix.

Distributed linear algebra packages more commonly adopt a block cyclic distribution, in which adjacent matrix blocks are assigned cyclically to adjacent processors, rather than contiguously in local memory as in the checkerboard distribution. This allows slices of the distributed matrix to be taken without affecting load balance. Sections 2B and 2C will demonstrate algorithms which, indeed, suffer from the poor load balance of the checkerboard distribution. However, we will also show (see Fig. 4) that each TPU core must be fed a matrix of about 25% of the maximum available linear size to begin saturating the serial throughput of the MXUs. In practice, this need for very large block sizes makes the block cyclic distribution impractical.

Now let us discuss the SUMMA algorithm. We will rehearse the untransposed case, and thus seek

$$C_{ij} = \sum_k^K A_{ik} B_{kj} \qquad [1]$$

for an (M, N) matrix $\mathbf{C}$, and (M, K) matrix $\mathbf{A}$, and a (K, N) matrix $\mathbf{B}$. It is convenient to also write the above equation as

$$\mathbf{C} = \mathbf{AB}. \qquad [2]$$

SUMMA works by dividing the K values of index k into N_b "panels" of k_b entries each. We will use Greek letters to enumerate

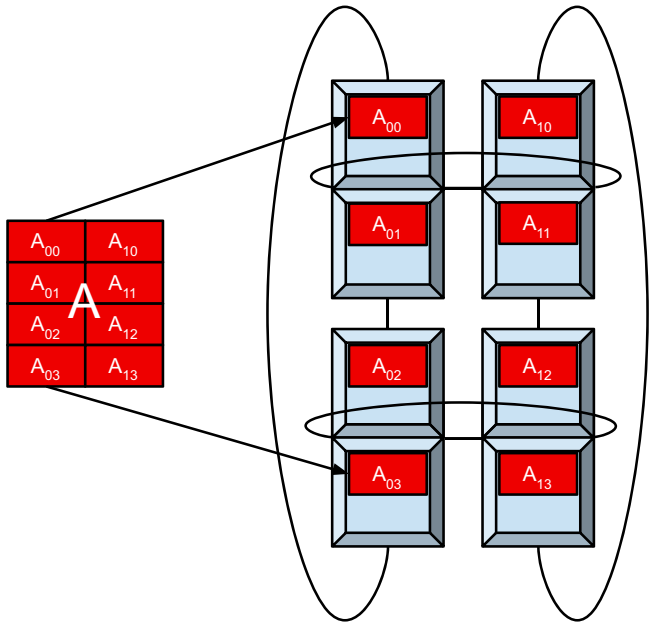

Fig. 2. The matrix **A** is "checkerboard" distributed onto a (4, 2) "v3-8" TPU grid, by partition into a corresponding (4, 2) grid of contiguous matrix blocks (red rectangles). The TPU cores are depicted as light blue squares, each separate chip is depicted as a pair of two adjacent such squares, and the 2D toroidal network connectivity between chips is depicted as black lines.

such panels, for example, κ. Let us define corresponding matrix panels $\mathbf{A}^{(\kappa)}$ and $\mathbf{B}^{(\kappa)}$ by

$$\mathbf{A}^{(\kappa)} \equiv A_{0:M-1\ k':k''}, \quad \mathbf{B}^{(\kappa)} \equiv B_{k':k''\ 0:N-1}, \qquad [3]$$

where $k' = \kappa k_b$ and $k'' = (\kappa+1)k_b - 1$. Expressed in this notation, Eq. **1** becomes

$$\mathbf{C} = \sum_{\kappa}^{N_b} \mathbf{A}^{(\kappa)}\mathbf{B}^{(\kappa)}. \qquad [4]$$

Notice that each term in the summand of Eq. **4** is a matrix product, $\mathbf{C}^{(\kappa)} = \mathbf{A}^{(\kappa)}\mathbf{B}^{(\kappa)}$. SUMMA works by parallelizing each individual such matrix product.

Given that $\mathbf{A}$ and $\mathbf{B}$ are already checkerboard distributed, the block column panel $\mathbf{A}^{(\kappa)}$ must therefore be broadcast to all other processor columns within processor rows, and the block row panel $\mathbf{B}^{(\kappa)}$ must be broadcast to all other processor rows within processor columns. Performing these broadcasts simultaneously exploits all four channels of each TPU chip in a pipelined fashion, with a maximum broadcasted distance of $\max(p_r, p_c)//2$ (whether and how to pipeline, in practice, is decided automatically by the XLA compiler). The resulting matrix $\mathbf{C}$ inherits the same checkerboard distribution as the inputs, as illustrated in Fig. 3.

By choosing k_b to be small relative to m and n but large enough to yield good single-core throughput (larger than about 512, in practice), this algorithm makes near-optimal use of TPU resources, while consuming negligible memory apart from that needed to store $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$. This is evinced in Fig. 1, *Top Left*, which shows the wall clock time required to multiply square fp32 matrices of size N distributed across various TPU v3 configurations.

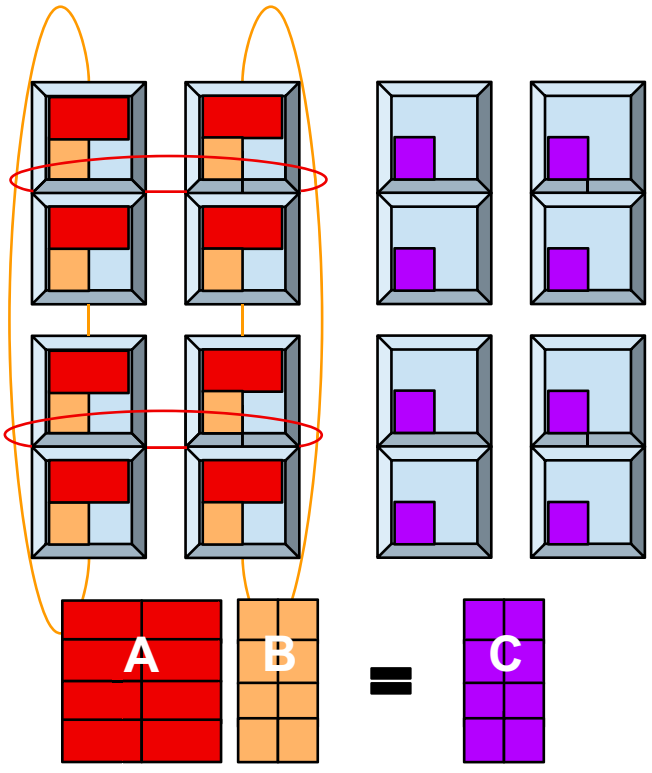

Fig. 3. Matrix distributions before and after SUMMA matrix multiplication, $\mathbf{AB} = \mathbf{C}$. (*Left*) The matrix factors **A** (red) and **B** (orange) are checkerboard distributed onto a v3-8 TPU grid. During multiplication, **A** will be communicated across processor columns (along the red-colored interconnects), and **B** will be communicated across processor rows (along the orange-colored interconnects). (*Right*) Distribution of the result matrix **C**.

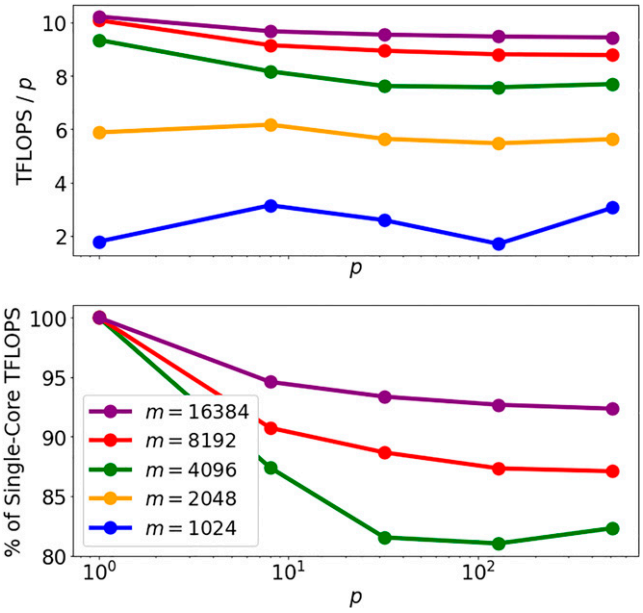

Fig. 4. Weak scaling data for distributed TPU matrix multiplication. Each curve holds local matrix sizes, fixed by the number of rows m per core, constant. The x axis shows the number of TPU cores p. (*Top*) The TFLOPS Eq. **5** per core. (*Bottom*) What percentage of the corresponding $p = 1$ value is attained by each point on the top three curves.

The maximum value of N is determined, by the necessity, to fit $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ in memory, demonstrating SUMMA's negligible need for additional memory. For instance, on a full TPU pod (2,048 cores), we can fit two (N, N) matrices of linear size $N = 2^{20} = 1{,}048{,}576$, which can then be multiplied in about 2 min. For large enough N, the straight lines on the log–log plot indicate runtime is dominated by $O(N^3)$ operations.

The excellent scaling with increasing number of TPU cores p can be seen by, in turn, consulting the two graphs in Fig. 4; p here is the x axis, while each line holds the number of matrix rows per core $m = N/\sqrt{2p}$ fixed. Notice that the undistributed $p = 1$ case, which does not invoke SUMMA, is also included.

Fig. 4 invokes the throughput speed of the operations in tera FLOPS (TFLOPS),

$$\text{TFLOPS} \equiv \frac{2N^3}{t \cdot 10^{12}}, \qquad [5]$$

where t is the measured wall clock time in seconds. Very heuristically, Eq. **5** measures the number of multiplications and additions implicitly performed by the TPUs per second. Fig. 4, *Top* plots the TFLOPS per core (TFLOPS/p) against p. We see that the $p = 1$ performance only begins to saturate (to a bit more than 10 TFLOPS) around $m = 4{,}096$, which is an appreciable fraction of the memory available per core. As alluded to earlier, this motivates our choice of a checkerboard rather than a block-cyclic distribution, since the latter would necessitate smaller local blocks and thus significantly degrade performance in all but the largest cases.

Optimal scaling would be indicated by flat horizontal lines. For large m, we quite nearly reach this optimum, as depicted quantitatively in Fig. 4, *Bottom*, which shows the percentage of the corresponding $p = 1$ value attained by each point of the top three curves. For $m = 16{,}384$, this is quite nearly 95%. The nonmonotonicity of the bottom two curves is presumably a consequence of the operation not being fully computationally bound here.

In this study, we are primarily concerned with the large N regime. Scaling is less favorable for small N, both within and between TPU configurations. Two problems occur when N is

small: The block outer products in Eq. **4** become too small to obtain good serial throughput from the TPU cores, and the constant overhead cost to initiate a communication becomes important relative to the cost of communication itself. Smaller N performance could be improved, if needed, by exploiting the extra available memory. By copying $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ between some or all processors rather than distributing among them, the individual summands in Eq. **4** can be evaluated in parallel; this strategy is sometimes known as a "2.5D algorithm." Similar considerations could be applied to the QR and matrix function algorithms.

As visible from the bottom two ($m = 1{,}024$, blue; and $m = 2{,}048$, yellow) curves in Fig. 4, *Top*, performance also becomes less predictable and more implementation dependent in the small N regime, since the MXU is not fully utilized in this case. For example, with $m = 1{,}024$, $p = 1$ actually performs much worse than $p = 8$, since the local matrices involved are twice as large in the latter case, due to the now-rectangular computational grid. Subsequent values of p begin to perform worse, presumably due to additional communication costs as more hosts are deployed. The suddenly improved performance at the largest value of p is somewhat surprising; we note, however, that the effect disappears at large N.

B. QR Factorization. The QR factorization rewrites an (M, N) matrix $\mathbf{A}$ with $M \geq N$ as the product of a "Q factor" with orthonormal columns and an upper-triangular "R factor," $\mathbf{A} = \mathbf{QR}$. Two closely related factorizations can be distinguished: the "full" factorization, with $\mathbf{Q}$ (M, M) and $\mathbf{R}$ (M, N); and the "reduced" factorization, with $\mathbf{Q}$ (M, N) and $\mathbf{R}$ (N, N). Both cases serve as a primitive in many applications, since, for example, the reduced $\mathbf{Q}$ factor orthonormally spans the column space of $\mathbf{A}$.

JAX via XLA provides an efficient and stable single-core QR factorization algorithm based on blocked Householder transformations as described in ref. 20. We focus here on distributing the computation over TPU grids, using a suitably adjusted version of the CAQR algorithm of ref. 16. In brief, our approach is as follows:

1) A panel of b columns of $\mathbf{A}$ is selected, labeled $\mathbf{A}_l$ in Fig. 5.
2) Column factorization: The full QR decomposition of that panel is implicitly computed, $\mathbf{A}_l = \mathbf{Q}_f\mathbf{R}_f$. $\mathbf{A}_l$ is replaced with $\mathbf{R}_f$.
3) Panel update: The remaining columns $\mathbf{A_r}$ are replaced by $\mathbf{Q}_f^H\mathbf{A}_r$.

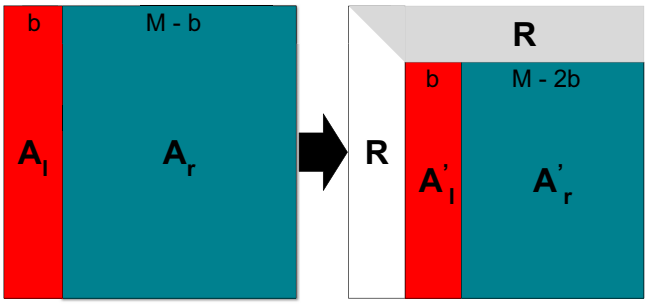

Fig. 5. Depiction of the CAQR algorithm (16) used to factor matrices distributed across 2D processor grids. Each iteration uses TSQR to factor the column panel $\mathbf{A_l}$, first into its reduced Q factor, and then into an implicit WY representation of its full Q factor. The latter is then applied to the panel $\mathbf{A_r}$, replacing a new strip of $\mathbf{A}$ with data from its $\mathbf{R}$ factor. The process is then repeated with new $\mathbf{A_l}$ and $\mathbf{A_r}$ (labeled $\mathbf{A'_l}$ and $\mathbf{A'_r}$, right of the arrow in the figure) until $\mathbf{R}$ has fully replaced $\mathbf{A}$.

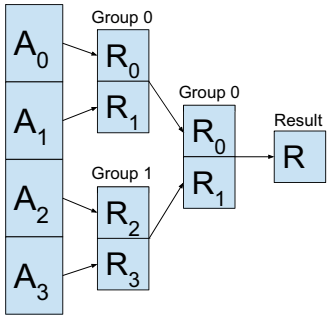

Fig. 6. Depiction of the TSQR algorithm (16) used to factor "tall skinny" matrices distributed across columns of processors. Each processor first computes a local QR decomposition of its matrix panel $\mathbf{A}_j$. The resulting $\mathbf{R}$ are gathered between processor pairs and then stacked. The process is iterated until each processor contains the same $\mathbf{R}$ factor, which is that of the full $\mathbf{A}$. Only the computation of $\mathbf{R}$ is illustrated; the local $\mathbf{Q}$ factors can be accumulated by having each processor multiply each "reduced" $\mathbf{Q}$ factor obtained by its successor during each step.

The above basic procedure is known as "right-looking block QR." Typically, the column factorization step would be handed by computing "Householder" representations of individual columns of $\mathbf{Q}_f$ one by one, but this involves too many scalar operations on TPUs.

Instead, we use the so-called "TSQR" algorithm (16), which computes the reduced QR factorization of a tall and skinny matrix $\mathbf{A}_l$. Tall and skinny means that the matrix can be divided into row panels of size m_r such that $m_r \geq N$. Since our $\mathbf{A}_l$ is a slice of b columns from a checkerboard-distributed $\mathbf{A}$, for our purposes, this means $M//p_r \geq b$, where p_r is the number of processor rows.

The TSQR algorithm performs the factorization of $\mathbf{A}_l$ via the binary reduction depicted in Fig. 6, with pseudocode given as *Algorithm 1*. Each processor in a column computes a local QR decomposition of $\mathbf{A}_l$, yielding a local $\mathbf{R}$ factor. The processors are arranged into groups of two, and the local $\mathbf{R}$ factors are gathered within these groups. Pairs of groups are successively combined, and the process is repeated until only a single $\mathbf{R}$ factor remains, which is that of $\mathbf{A}_l$.

This procedure yields the reduced factors $\mathbf{Q}_r$ and $\mathbf{R}_r$ of $\mathbf{A}_l$. The full $\mathbf{R}_f$ factor is straightforwardly obtained by appending rows of zeros to $\mathbf{R}_r$. We get the full $\mathbf{Q}_f$ factor implicitly as its so-called $\mathbf{WY}$ representation (20), $\mathbf{Q}_f = \mathbf{I} - \mathbf{WY}^H$, where $\mathbf{W}$ and $\mathbf{Y}$ are both (M, b).

To compute $\mathbf{W}$ and $\mathbf{Y}$, we use a slight modification of the "Yamamoto" procedure outlined in ref. 21. The Yamamoto procedure has us form

$$\mathbf{Q}_f = \mathbf{I} - \mathbf{WTW}^H \quad \textbf{[6a]}$$

$$\mathbf{W} = \mathbf{Q}_r - \mathbf{I} \quad \textbf{[6b]}$$

$$\mathbf{T}^{-1} = \mathbf{I} - \mathbf{Q}_1, \quad \textbf{[6c]}$$

where $\mathbf{T}^{-1}$ is (b, b) and $\mathbf{Q}_1$ is the first b rows of $\mathbf{Q}_r$. $\mathbf{T}^{-1}$ rather than $\mathbf{T}$ is stored, and multiplications by $\mathbf{T}$ are handled via linear solution. This representation is simple to compute, and it saves memory compared to the $\mathbf{W}$ $\mathbf{Y}$ form, since $\mathbf{T}$ is smaller than $\mathbf{Y}$. Nevertheless, we prefer to form $\mathbf{Y}$ explicitly via $\mathbf{Y}^H = \mathbf{TW}^H$, so that only one, trivially parallel, linear solve need be performed—compared to one per each multiplication by $\mathbf{Q}_f$.

Note that Eq. **6** breaks down if $\mathbf{T}^{-1}$ is ill conditioned, which can occur, for example, if $\mathbf{Q}_1$ is, itself, very near to the identity. The said difficulty can be alleviated by a slight generalization described in ref. 21, replacing each $\mathbf{I}$ in Eq. **6** with a diagonal matrix of signs chosen to improve $\mathbf{T}^{-1}$'s conditioning. However, neither

ref. 21 nor the references it cites specifies how, precisely, to choose these. Generalizing from heuristics like "flip the sign wherever $\mathbf{T}^{-1}$ would otherwise have a row or column of zeros" proves not entirely trivial. Having yet to encounter a practical case of breakdown, we have not implemented the full generalization.

With $\mathbf{W}$ and $\mathbf{Y}$ in hand, we can now straightforwardly perform the panel update step, yielding the full CAQR algorithm. It is depicted in Fig. 5 and given as pseudocode in *Algorithm 2*.

Performance is depicted in Fig. 1, *Top Right*. For instance, on a full TPU pod (2,048 cores), we obtain the QR decomposition of an (N, N) matrix of linear size $N = 2^{20} = 1{,}048{,}576$ in about 20 min. Excellent scaling is seen with large N, showing that the task is dominated by the matrix multiplication update steps. However, our choice of a checkerboard rather than a block-cyclic distribution pattern for the matrix $\mathbf{A}$ can result in poor load balancing, since, in effect, we treat an equally sized matrix at each iteration.

More quantitatively, let us consider the case that $\mathbf{Q}$ is not computed, so that the dominant expense of the algorithm is the update of $\mathbf{A}$ (line 7 in *Algorithm 2*). Let $\tilde{M}$ and $\tilde{N}$ be the respective dimensions of the matrix block being updated. The total cost c of the updates is then

$$c = \sum_{i=0}^{i=\frac{N}{b}-1} 4b\tilde{M}\tilde{N}. \qquad [7]$$

Suppose one could correctly reduce the size of the updated block during the computation, as would be made possible by a block-cyclic data distribution. At iteration i, we have $\tilde{M} = M - (i+1)b$ and $\tilde{N} = N - (i+1)b$, and thus a block-cyclic expense c_{bc} of

$$c_{\mathrm{bc}} = 2MN^2 - \frac{2}{3}N^3. \qquad [8]$$

However, using our checkerboard distribution, processors must work as if the block does not reduce in size. We then have $\tilde{M} = M$, $\tilde{N} = N$, and thus a checkerboard-distributed expense c_{chk} of

$$c_{\mathrm{chk}} = 4MN^2. \qquad [9]$$

The ratio of these factors, which is three in the $N = M$ case (decreasing to 2.4 if $\mathbf{Q}$ is also computed), is the unrealized optimization offered by a block-cyclic distribution. The optimization could be realized fairly straightforwardly—although with some risk of losing single-core throughput speed due to the smaller local matrix sizes—but we leave doing so to future studies. Note also that the penalty is at least partially compensated for by the improved single-core throughput in the checkerboard-distributed case, achieved by the larger individual blocks fed to the MXUs.

Algorithm 1 TSQR

Require: (M, N) matrix $\mathbf{A}$ distributed among p_r processor rows such that $N > M//p_r$.
1: group size $\leftarrow 1$.
2: $\mathbf{R}, \mathbf{Q} \leftarrow qr(\mathbf{A})$.
3: **while** group size $< p_r$ **do**
4: Double the group size.
5: Broadcast both unique $\mathbf{R}$ factors within each group.
6: Vertically stack the newly broadcast $\mathbf{R}$.
7: $\mathbf{Q}, \mathbf{R} \leftarrow qr(\mathbf{R})$.
8: $\mathbf{Q} \leftarrow \mathbf{Q}\mathbf{Q}_l$ (optional)
9: **return** $\mathbf{Q}$, $\mathbf{R}$

Algorithm 2 Right-looking CAQR

Require: Checkerboard-distributed (M, N) matrix $\mathbf{A}$.
1: Divide $\mathbf{A}$ into N_b column panels of size b, $b = M//N_b$.
2: $\mathbf{Q} \leftarrow \mathbf{I}$.
3: **for** $j \in [0, N_b)$**, do**
4: $\mathbf{A}_l \leftarrow (M - jN_b, N_b)$ panel of $\mathbf{A}$ from $\mathbf{A}_{jN_b, jN_b}$.
5: $\mathbf{A}_r \leftarrow$ all entries in $\mathbf{A}$ right of $\mathbf{A}_l$.
6: $\mathbf{Q}_r, \mathbf{R}_r \leftarrow$ TSQR$(\mathbf{A}_l)$ (reduced Q factor)
7: Replace $\mathbf{A}_l$ in $\mathbf{A}$ with $[\mathbf{R}_r, \mathbf{0}]^T$.
8: Compute $\mathbf{W}, \mathbf{Y}$ such that (s.t.) $\mathbf{Q}_f = \mathbf{I} - \mathbf{W}\mathbf{Y}^H$ (see text)
9: Replace $\mathbf{A}_r$ in $\mathbf{A}$ with $\mathbf{Q}_f^H \mathbf{A}_r$.
10: $\mathbf{Q} \leftarrow \mathbf{Q}_{:, M-jN_b:}\mathbf{Q}_f$ (optional).
11: **return** $\mathbf{Q}$, $\mathbf{R} \leftarrow \mathbf{A}$.

C. Linear Solution. By "linear solution," we mean the determination of $\mathbf{x}$ in

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \qquad [10]$$

where $\mathbf{A}$ and $\mathbf{b}$ are given, with $\mathbf{A}$ an (N, N) matrix, and $\mathbf{x}$ and $\mathbf{b}$ both (N, k). We consider the case of $\mathbf{A}$ given as a dense, full-rank matrix, in which case Eq. **10** is typically solved in $O(N^3)$ operations via an initial LU decomposition.

Unfortunately, an efficient distributed-TPU LU factorization is not yet available. Instead, we use the QR factorization (as described above), which is more stable and only marginally less efficient. Writing $\mathbf{A} = \mathbf{Q}\mathbf{R}$, we have

$$\mathbf{R}\mathbf{x} = \mathbf{b}', \qquad [11]$$

where $\mathbf{b}' \equiv \mathbf{Q}^H\mathbf{b}$. That is, we have mapped the general linear system in Eq. **10** to the upper triangular one in Eq. **11**. In a scalar implementation, such upper triangular systems are trivially soluble by repeated substitution. The row containing a single nonzero element, for example, corresponds to the scalar equation $y_N = x_N$, which is substituted into the row containing two nonzero elements, and so on.

This scalar algorithm is, however, quite TPU unfriendly. Instead, we first note that a reasonably performant single-TPU upper triangular solver, which uses the TPU vector processor and blocking to achieve acceptable performance, ships with JAX. We can leverage this into a naive, but acceptably performant, distributed triangular solver as depicted in Fig. 7. The coefficient matrix $\mathbf{R}$ is first divided into square blocks such that each is local

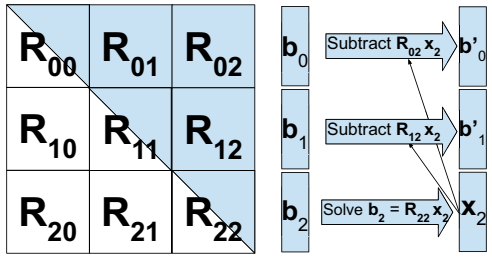

Fig. 7. Depiction of our somewhat naive approach to solving upper triangular systems. We divide the upper triangular matrix **R** into square blocks such that each is local to a core, and those on the main block diagonal are themselves locally upper triangular. The solution is then found by moving upward along the main block diagonal from the bottom right—the figure depicts the first such step. At each step, the relevant panel, in this case, $\mathbf{x}_2$, of the solution **x** is first found by solving the corresponding local triangular system, in this case, $\mathbf{R}_{22}\mathbf{x}_2 = \mathbf{b}_2$. The result is broadcast upward along its column panel, and used to update the coefficients **b** as depicted. The procedure then iterates to the upper left.

to a given processor (a processor may, however, contain more than one block). The submatrices on the block main diagonal are then themselves upper triangular.

Algorithm 3 Distributed triangular solver.

Require: Checkerboard-distributed (N, N) upper-triangular coefficient matrix $\mathbf{R}$, $(N,)$ right-hand side vector $\mathbf{b}$ copied between columns.

Divide $\mathbf{R}$ into N_b square blocks of linear size $N//N_b$, indexed with Greek letters.
Divide $\mathbf{b}$ into N_b row panels also of size $N//N_b$, indexed with Greek letters.
for $\kappa \in (N_b, 0]$, **do**
  Solve $\mathbf{R}_{\kappa\kappa}\,\mathbf{x}_\kappa = \mathbf{b}_\kappa$.
  Broadcast $\mathbf{x}_\kappa$ upward along its processor column.
  $\mathbf{b}_\kappa \leftarrow \mathbf{x}_\kappa$
  for $\gamma \in [0, \kappa)$, **do** ▷ Do this in parallel.
    $\mathbf{b}_\gamma \leftarrow \mathbf{b}_\gamma - \mathbf{A}_{\gamma\kappa}\mathbf{x}_\kappa$.
  Copy the updated $\mathbf{b}$ to the other processor columns.
return $\mathbf{b}$

From here, we perform a direct blocked analogy of the scalar elimination procedure described above, with each column panel of $\mathbf{R}$ treated in serial. First, the triangular system at the bottom of the panel is solved (e.g., $\mathbf{R}_{22}\mathbf{x}_2 = \mathbf{b}_2$ in Fig. 7). The resulting $\mathbf{x}_i$ is the panel of the full solution overlapping its corresponding $\mathbf{b}_i$, and, if desired, $\mathbf{b}_i$ may be overwritten by it in place. The corresponding substitution is achieved by broadcasting $\mathbf{x}_i$ to the blocks above it, and subtracting $\mathbf{R}\mathbf{x}_i$ from each $\mathbf{b}$ panel above. Pseudocode is given as *Algorithm 3*.

After the initial QR factorization, this algorithm is poorly load balanced. The cores storing zeroes of $\mathbf{R}$ are left completely idle; only the update step runs in parallel; and, during it, only the cores above the current main block diagonal do work. Much better load balancing could be achieved by adopting a block-cyclic data distribution, so that the processor grid was not so tightly coupled to the matrix block locations. The algorithm is, however, sufficiently efficient to represent a small expense compared to the QR step, as can be seen in Fig. 1, *Bottom Left*. As an example, for an (N, N) matrix $\mathbf{A}$ of linear size $N = 2^{20} = 1{,}048{,}576$, we can solve a linear system on a full TPU pod (2,048 cores), again, in about 20 min.

D. Matrix Functions. Above, we considered application of TPU slices toward bread-and-butter tasks in scientific computing. Since TPUs are natively optimized for matrix multiplication, it is most natural to consider also tasks based on matrix multiplication, such as matrix functions (implemented approximately as matrix polynomials, thus requiring matrix multiplications and additions). Next, we briefly review two types of matrix functions that transform, respectively, the singular values and the eigenvalues of a matrix.

Recall first that every matrix has a singular value decomposition (SVD),

$$\mathbf{A} = \mathbf{U}_s\,\Sigma\,\mathbf{V}_s^H, \qquad [12]$$

with Σ a diagonal matrix of real singular values and $\mathbf{U}_s$ and $\mathbf{V}_s^H$, the left and right singular vectors, both unitary. Given any polynomial $f(x) = a_1 x + a_3 x^3 + a_5 x^5 + \cdots$ made only of odd powers of x, we can define the matrix function $f(\mathbf{A})$ acting on the singular values of $\mathbf{A}$,

$$f(\mathbf{A}) \equiv \mathbf{U}_s\, f(\Sigma)\, \mathbf{V}_s^H \quad \text{(singular values)}, \qquad [13]$$

where $f(\Sigma)$ is a diagonal matrix where each diagonal entry contains the result of applying f to the corresponding singular value in Σ. Notice that $\mathbf{A}$ and $\mathbf{A}(\mathbf{A}^\dagger\mathbf{A})^n$ share the same structure of singular vectors for any integer $n = 0, 1, 2, \cdots$; that is,

$$\mathbf{A}(\mathbf{A}^\dagger\mathbf{A})^n = \mathbf{U}_s\,\Sigma^{2n+1}\,\mathbf{V}_s^H. \qquad [14]$$

It then follows that we can compute $f(\mathbf{A})$ by means of the matrix polynomial expansion

$$a_1\mathbf{A} + a_3\mathbf{A}\mathbf{A}^H\mathbf{A} + a_5\mathbf{A}\left(\mathbf{A}^H\mathbf{A}\right)^2 + \ldots \qquad [15]$$
$$= \mathbf{U}_s\left(a_1\Sigma + a_3\Sigma^3 + a_5\Sigma^5 + \cdots\right)\mathbf{V}_s^H \qquad [16]$$
$$= \mathbf{U}_s\, f(\Sigma)\,\mathbf{V}_s^H = f(\mathbf{A}). \qquad [17]$$

Recall now that every diagonalizable square matrix also has an eigenvalue decomposition,

$$\mathbf{A} = \mathbf{P}_e\,\Omega\,\mathbf{P}_e^{-1}, \qquad [18]$$

with Ω as a diagonal matrix of (possibly complex) eigenvalues and $\mathbf{P}_e$ as an invertible matrix whose columns encode the right eigenvectors of $\mathbf{A}$. Given an arbitrary polynomial $g(x) = a_0 + a_1 x + a_2 x^2 + \cdots$, we can define the matrix function $g(\mathbf{A})$ for a diagonalizable square matrix $\mathbf{A}$ by acting on its eigenvalues,

$$g(\mathbf{A}) \equiv \mathbf{P}_e\, g(\Omega)\,\mathbf{P}_e^{-1} \quad \text{(eigenvalues)}, \qquad [19]$$

with $g(\Omega)$ as a diagonal matrix where each diagonal entry contains the result of applying g to the corresponding eigenvalue in Ω. We emphasize that this definition of matrix function, based on transforming the eigenvalues while preserving the structure of eigenvectors, is not equivalent to that in Eq. **13**, which transformed the singular values while preserving the singular vectors. We observe that the matrices $\mathbf{A}^n$ for $n = 0, 1, 2, \cdots$ share the same structure of eigenvectors; that is,

$$\mathbf{A}^n = \mathbf{P}_e\,\Omega^n\,\mathbf{P}_e^{-1}. \qquad [20]$$

It then follows that we can compute $g(\mathbf{A})$ by means of the matrix polynomial expansion

$$a_0\mathbf{I} + a_1\mathbf{A} + a_2\mathbf{A}^2 + \cdots \qquad [21]$$
$$= \mathbf{P}_e\left(a_0\mathbf{I} + a_1\Omega + a_2\Omega^2 + \cdots\right)\mathbf{P}_e^{-1} \qquad [22]$$
$$= \mathbf{P}_e\, g(\Omega)\mathbf{P}_e^{-1} = g(\mathbf{A}). \qquad [23]$$

Various matrix functions of interest, such as matrix sign function and matrix inverse (see below), but also matrix principal square root, matrix inverse principal square root, matrix exponential, matrix logarithm, etc., can be accurately approximated by polynomials (or polynomial iterations) of one of the two forms above, and thus efficiently computed and scaled on TPUs. Here we illustrate this with the so-called polar decomposition, which is obtained through applying the sign function to the singular values, where the sign function is approximated by means of a polynomial iteration made of small polynomials of the type in Eq. **15**.

The polar decomposition of an arbitrary (M, N) matrix $\mathbf{A}$ with $M \geq N$ is defined by

$$\mathbf{A} = \mathbf{U}\mathbf{H}, \qquad [24]$$

where the (M, N) matrix $\mathbf{U}$ has N orthonormal columns, and the (N, N) matrix $\mathbf{H}$ is positive semidefinite. This is a matrix version of the polar decomposition $z = e^{i\phi}|z|$ of a complex number z

into its complex phase $e^{i\phi}$ and its nonnegative norm $|z|$. In terms of the SVD Eq. **12**, we have

$$\mathbf{U} = \mathbf{U}_s \mathbf{V}_s^H, \qquad [25]$$

that is, that the polar factor $\mathbf{U}$ can be obtained by setting all the singular values of $\mathbf{A}$ to one while leaving the singular vectors untouched.

It is easy to confirm that repeated application of the scalar polynomial iteration

$$x_{i+1} = \frac{1}{2} x_i (3 - x_i^2) \qquad [26]$$

sends any initial $x_0 \in (0, \sqrt{3}) \to +1$, while sending $x_0 = 0$ to zero. In other words, this polynomial iteration converges to the sign function when applied on the interval $[0, \sqrt{3})$. The corresponding matrix polynomial, the Newton–Schulz iteration,

$$\mathbf{X}_{i+1} = \frac{1}{2} \mathbf{X}_i (3\mathbf{I} - \mathbf{X}_i^H \mathbf{X}_i), \qquad [27]$$

starting with $\mathbf{X}_0 \equiv \mathbf{A}$, thus has the same effect upon the singular values of $\mathbf{A}$, and therefore has the unitary polar factor of $\mathbf{U}$ in Eq. **25** as its fixed point. As confirmed in ref. 22, this iteration is numerically stable for any $\mathbf{A}$ with $||\mathbf{A}||_2 < \sqrt{3}$, where $||\mathbf{A}||_2$ denotes the spectral two-norm of $\mathbf{A}$, or its largest singular value. We can ensure this property for general input by an initial rescaling. We use $\mathbf{A} \to \mathbf{X}_0 \equiv \mathbf{A}(\sqrt{3} - \delta)/||\mathbf{A}||_F$, where $||\mathbf{A}||_F$ denotes the Frobenius norm (which can be computed easily as $\sqrt{\mathrm{tr}(\mathbf{A}\mathbf{A}^H)}$ and fulfils $||\mathbf{A}||_F \geq ||\mathbf{A}||_2$), and δ is an arbitrary, small positive number.

Once the smallest singular value s_0 of $\mathbf{A}$ grows to 0.1 or so, through the Newton–Schulz iterations Eq. **27**, it then subsequently enjoys quadratic convergence to one. When working in single precision, this means that it only requires about 10 further iterations before it reaches one within that precision. Convergence before this point can, unfortunately, be rather slow, so that 35 to 50 iterations might be required if s_0 is initially very small.

To improve upon this, we choose a desired minimum singular value s_-, and apply a preconditioning polynomial,

$$a = \frac{3}{2}\sqrt{3} - s_- \qquad [28a]$$

$$x_{i+1} = a x_i \left(1 - \frac{4}{27}(a x_i)^2\right) \qquad [28b]$$

$$\mathbf{X}_{i+1} = a \mathbf{X}_i \left(\mathbf{I} - \frac{4}{27} a^2 \mathbf{X}_i^H \mathbf{X}_i\right). \qquad [28c]$$

While Eq. **28b** does not monotonically drive values toward one (Fig. 8), it does monotonically drive any beneath s_- upward, more quickly than Eq. **26**, while keeping those larger comfortably above the threshold $[s_-, 1]$. Consequently, Eq. **28c** rapidly improves the conditioning of $\mathbf{X}$ without affecting its singular vectors. The number of applications needed to obtain a spectrum in $[s_-, 1]$ can be tracked by repeatedly feeding an estimated initial minimum singular value s_0 through Eq. **28b** until a value greater than s_- is obtained. We typically use $s_0 = \epsilon$, the machine precision—about 10^{-7} in single precision—which requires about 10 to 15 iterations for $s_- = 0.1$. Notice that, in order to use Eqs. **28a**–**28c**, we want to rescale the initial matrix $\mathbf{A}$ to have singular values in the interval $[0, 1)$, which we achieve through $\mathbf{A} \to \mathbf{X}_0 \equiv \mathbf{A}(1 - \delta)/||\mathbf{A}||_F$ for some small $\delta > 0$.

Algorithm 4 summarizes our approach. In total, it then takes about 25 iterations to obtain the polar factor of an arbitrary matrix, which could potentially be reduced to about 10 for well-conditioned input with $s_0 = 0.1$. Thus, this operation is equivalent to about 50 matrix multiplications. This is demonstrated in Fig. 1, *Bottom Right*, essentially a rescaling of Fig. 1, *Top Left* by a factor of about 50. Memory footprint, scaling, and use of hardware resources follow the same reasoning as for SUMMA, since the algorithm consists simply of repeated calls to SUMMA. As an example, on a full TPU pod (2,048 cores), we can compute the polar decomposition of an (N, N) matrix of linear size $N = 2^{19} = 524{,}288$ in about 20 min.

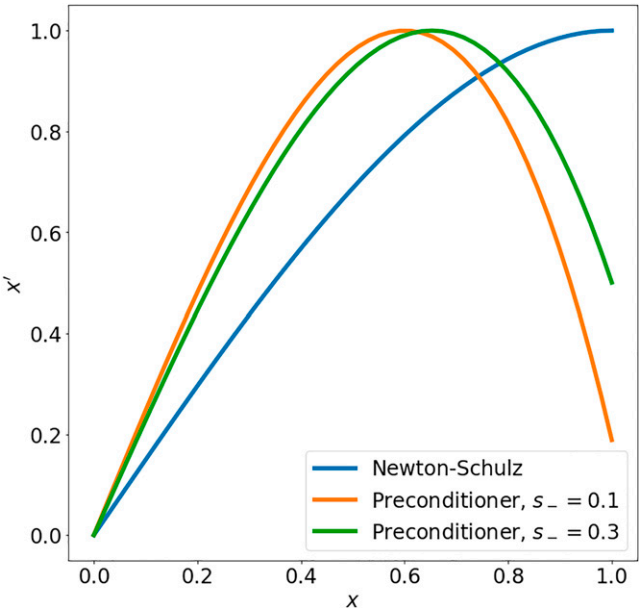

Fig. 8. The Newton–Schulz polynomial Eq. **26**, alongside the preconditioning polynomial Eq. **28b** for different choices of s_-. No input is mapped by Eq. **28b** beneath s_-, but, compared to Eq. **26**, the slope of the latter is much larger near zero.

As alluded to above, various iterations besides that leading to the polar decomposition can also be efficiently implemented. For example, for the electronic structure DFT computations presented in ref. 12, the matrix inverse square root of an overlap matrix for single-electron basis functions needs to be computed, as well as a so-called purification of a Hermitian matrix that is similar to the polar decomposition described above. The Newton–Schulz procedure may also be used to compute matrix inverses, as detailed in *Algorithm 5*.

Algorithm 4 Preconditioned Newton-Schulz Polar Factorization

Require: (M, N) matrix $\mathbf{A}$ with $M \geq N$, threshold $s_- \sim 0.1$, error tolerance ϵ, estimated smallest singular value s_0.
Ensure: Isometric $\mathbf{U}$ giving positive semidefinite $\mathbf{H} \equiv \mathbf{U}^H \mathbf{A}$.

$\mathbf{U} \leftarrow \mathbf{A}/||\mathbf{A}||_F$.
$s \leftarrow s_0$ ▷ Or ϵ if unsupplied.
$a \leftarrow \frac{3}{2}\sqrt{3} - s_-$
while $s < s_-$ **, do** ▷ Bounds singular values by $[s_-, 1]$.
  $s \leftarrow as(1 - \frac{4}{27} a^2 s^2)$
  $\mathbf{U} \leftarrow a\mathbf{U}(\mathbf{I} - \frac{4}{27} a^2 \mathbf{U}^H \mathbf{U})$
while $\delta > \max(M, N)\epsilon$ **, do**
  $\mathbf{U}' \leftarrow \frac{1}{2}\mathbf{U}(3\mathbf{I} - \mathbf{U}^H \mathbf{U})$
  $\delta \leftarrow ||\mathbf{U}' - \mathbf{U}||_F$
  $\mathbf{U} \leftarrow \mathbf{U}'$
return $\mathbf{U}$

Algorithm 5 Newton-Schulz Matrix Inversion

Require: (N, M) full-rank matrix $\mathbf{A}$.
Ensure: $\mathbf{X}$ s.t. one of $\mathbf{XA} = \mathbf{I}$ or $\mathbf{AX} = \mathbf{I}$.

```
a ← ||A A^H||_F.
X ← (1/a) A^H.
ε ← u                                  ▷ u is unit roundoff.
δ ← 2ε                                 ▷ Initial error greater than tolerance.
while δ > ε , do
    if XA = I desired, then
        X' ← X(2I − AX)
    else if AX = I desired, then
        X' ← (2I − AX)X
    δ ← ||X' − X||_F
    ε ← 2ε                             ▷ Note growing error tolerance.
    X ← X'
return return X
```

We can, in fact, approximate any sufficiently smooth function with a polynomial expansion. Given a scalar function g, if we know that the eigenvalues of a square matrix $\mathbf{A}$ are within some interval $[a, b]$, and we have a polynomial p_d that approximates g to a desired accuracy within that interval, we can evaluate $p_d(\mathbf{A})$ as in Eq. **21** to approximate $g(\mathbf{A})$ in the sense of Eq. **19**. Naive polynomial expansions are often oscillatory, but expansions in terms of Chebyshev polynomials minimize such oscillations, making them ideal for this use. The accuracy of the approximation is controlled by the degree d of the polynomial p_d, and evaluating $p_d(\mathbf{A})$ requires d matrix products, using the so-called Clenshaw summation method (23). How large a d is needed for a given accuracy depends on both the smoothness of g and the spectrum of $\mathbf{A}$, but the advantage of this method is that it can be easily applied to any piecewise smooth g.

3. Conclusion

In this paper, we have demonstrated the potential of TPUs to serve as accelerators for large-scale scientific computation, by using distributed, matrix multiply–based algorithms for the QR decomposition, solving a linear system, and matrix functions such as the polar decomposition. By distributing the matrices over a full pod of third-generation TPUs (2,048 cores), large matrices with linear size up to $N = 2^{20} = 1{,}048{,}576$ can be addressed, with computational times ranging from 2 min (for matrix multiplication) to 20 min (e.g., for QR decomposition). Moreover, a full pod of fourth-generation TPUs (8,192 cores) is expected to address matrices that double the above linear size in comparable times (work in progress). As shown in subsequent papers (9–14), the technology demonstrated here is already significant for a wide range of applications in the context of large-scale simulations and computations of quantum systems, including quantum computation, quantum many-body physics, quantum chemistry, and materials science.

Given the TPUs' role as accelerator, it is natural to wonder how the present results might compare with those achievable by, say, graphics processing units (GPUs). The question is actually a bit slippery. During single-precision matrix multiplication in the large-N limit, a single TPU-v3 chip (consisting of two cores) achieves comparable performance to the 19.5 TFLOPS theoretically obtained by an NVIDIA A100 GPU. For sufficiently large problems, distributed GPU performance should also be comparable, but a comparison here is complicated by the highly variable nature of distributed GPU configurations. In particular, GPU configurations directly connecting sufficiently many cards to directly store a dense $N \approx 10^7$ matrix essentially do not exist except in a very few, for example, academic contexts, inaccessible to the public.

Machine learning ASICs, on the other hand, are broadly accessible as a cloud service. For instance, anyone with a Google Cloud Platform account can have access to a TPU pod. This, alongside the potential for direct integration with TPU-native machine learning workloads, constitutes a qualitative advantage that cannot be addressed by a direct performance comparison. As a result, a number of large-scale scientific computing tasks such as the ones demonstrated in this paper and in refs. 10–14 are now within the reach of any reach group, contributing to democratizing supercomputing throughout the scientific community and beyond.

It is interesting to briefly detail approaches which have not been so successful. First, following the ideas in ref. 24, we, at one point, attempted to use *Algorithm 5* to compute a low-precision inverse with which to precondition a GMRES-based linear solver. While this does work, in the end, the QR approach is simply too much more efficient for this to be useful. Second, following a "spectral divide and conquer" approach described in ref. 25, either the polar factorization or the above purification routine may be successively applied to compute progressively smaller submatrices containing only half of an input matrix's eigenvalue spectrum, theoretically leading to an efficient Hermitian eigensolver based only on matrix multiplication. We have implemented such an eigensolver, but have found it to be lacking in both stability and efficiency in practice, due, partly, to XLA's need to recompile upon encountering matrices of new size.

3.1. Data Availability. Some study data are available. (At present, we are not legally able to share the computer code implementing the described functionality. The benchmark data are essentially completely specified in the manuscript, but could be included in tabulated form if desired.)

ACKNOWLEDGMENTS. This work would not have been possible without the at-times-heroic support from the Google teams associated with JAX and with Cloud TPUs, including but not limited to Skye Wanderman-Milne, Rasmus Larsen, Peter Hawkins, Adam Paszke, Stephan Hoyer, Sameer Agarwal, Matthew Johnson, Zak Stone, and James Bradbury. We thank Chase Riley Roberts, Jae Yoo, Megan Durney, Stefan Leichenauer and the entire Sandbox@Alphabet team for early work, discussions, encouragement, and infrastructure support. This research was supported with Cloud TPUs from Google's TPU Research Cloud. Sandbox is a team within the Alphabet family of companies, which includes Google, Verily, Waymo, X, and others. G.V. is a Canadian Institute For Advanced Research fellow in the Quantum Information Science Program and a Distinguished Visiting Research Chair at Perimeter Institute. Research at Perimeter Institute is supported by the Government of Canada through the Department of Innovation, Science and Economic Development and by the Province of Ontario through the Ministry of Research, Innovation and Science.

1. Y. Bar Sinai, S. Hoyer, J. Hickey, M. Brenner, Learning data-driven discretizations for partial differential equations. *Proc. Natl. Acad. Sci. U.S.A.* **116**, 15344–15349 (2019).
2. A. Bashir *et al.*, Machine learning guided aptamer refinement and discovery. *Nat. Commun.* **12**, 2366 (2021).
3. D. Kochkov *et al.*, Machine learning-accelerated computational fluid dynamics. *Proc. Natl. Acad. Sci. U.S A.* **118**, e2101784118 (2021).
4. L. Li *et al.*, Kohn-Sham equations as regularizer: Building prior knowledge into machine-learned physics. *Phys. Rev. Lett.* **126**, 036401 (2021).
5. T. Lu, Y. F. Chen, B. Hechtman, T. Wang, J. Anderson, *Large-scale discrete Fourier transform on TPUs.* arXiv [Preprint] (2020). https://doi.org/10.48550/arXiv.2002.03260. Accessed 26 July 2022.
6. C. Ma, T. Marin, T. Lu, Y.-F. Chen, Y. Zhuo, "Nonuniform fast Fourier transform on TPUs" in *2021 IEEE 18th International Symposium on Biomedial Imaging (ISBI)*, (Curran Associates, 2021), pp. 783–787.

7. K. Yang, Y. F. Chen, G. Roumpos, C. Colby, J. Anderson, "High performance Monte Carlo simulation of Ising model on TPU clusters" in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '19* (Association for Computing Machinery, New York, NY, 2019).
8. F. Huot, Y.-F. Chen, R. Clapp, C. Boneti, J. Anderson, *High -resolution imaging on TPUs.* arXiv [Preprint] (2019). https://doi.org/10.48550/arxiv.1912.08063. Accessed 26 July 2022.
9. E Gustafson, *et al.*, Large scale multi-node simulations of $\mathbb{Z}_2$ gauge theory quantum circuits using Google Cloud platform. arXiv [Prepint] (2021). https://doi.org/10.48550/arxiv.2110.07482. Accessed 26 July 2022.
10. M Hauru, *et al.*, *Simulation of quantum physics with Tensor Processing Units: Brute-force computation of ground states and time evolution* (2021).
11. A Morningstar, et al., Simulation of quantum many-body dynamics with Tensor Processing Units: Floquet prethermalization. *PRX Quantum* **3**, 020331 (2022).
12. Ryan Pederson et al., Tensor Processing Units for Quantum Chemistry (Sandbox@Alphabet, *in preparation*.) https://doi.org/10.48550/arXiv.2202.01255. Accessed 26 July 2022.
13. John Kozlowski et al., Acceleration and scaling of Couple Cluster methods with Tensor Processing Units (Sandbox@Alphabet, *in preparation*.) (year?).
14. Martin Ganahl et al., Density Matrix Renormalization Group using Tensor Processing Units (Sandbox@Alphabet, *in preparation*.) https://arxiv.org/pdf/2204.05693.pdf. Accessed 26 July 2022.
15. R. A. Van De Geijn, J. Watts, SUMMA: Scalable universal matrix multiplication algorithm. *Concurr. Pract. Exp* **9**, 255–274 (1997).
16. J. Demmel, L. Grigori, M. Hoemmen, J. Langou, Communication-optimal parallel and sequential QR and LU factorizations. *SIAM J. Sci. Comput.* **34**, A206–A239 (2012).
17. N. Jouppi *et al.*, A domain-specific supercomputer for training deep neural networks. *Commun. ACM* **63**, 67–78 (2020).
18. Anonymous, *XLA : Optimizing compiler for machine learners.* https://tensorflow.org/xla. Accessed 26 July 2022.
19. J Bradbury, *et al.*, JAX: composable transformations of Python+NumPy programs (2018). https://jax.readthedocs.io/en/latest/index.html. Accessed 26 July 2022.
20. G. H. Golub, C. F. V. Loan, *Matrix Computations.* (The John Hopkins University Press, Baltimore, MD, ed. 4, 2013).
21. G. Ballard *et al.*, "Reconstructing Householder vectors from tall-skinny qr" in *2014 IEEE 28th International Parallel and Distributed Processing Symposium* (Institute of Electrical and Electronics Engineers, 2014), pp. 1159–1170.
22. Y. Nakatsukasa, N. J. Higham, Backward stability of iterations for computing the polar decomposition. *SIAM J. Matrix Anal. Appl.* **33**, 460–479 (2012).
23. A. Gil, J. Segura, N. Temme, *Numerical Methods for Special Functions* (Society for Industrial and Applied Mathematics, 2007).
24. A. Haidar, S. Tomov, J. Dongarra, N. J. Higham, "Harnessing GPU Tensor Cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers" in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis* (Association for Computing Machinery, New York, NY, 2018), pp. 603–613.
25. Y. Nakatsukasa, N. Higham, Stable and efficient spectral divide and conquer algorithms for the symmetric eigenvalue decomposition and the SVD. *SIAM J. Sci. Comput.* **35**, A1325–A1349 (2013).