IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 21, 2021, accepted June 19, 2021, date of publication June 25, 2021, date of current version July 7, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3092312

Large-Scale Discrete Fourier Transform on TPUs

TIANJIAN LU, (Senior Member, IEEE), YI-FAN CHEN, BLAKE HECHTMAN, TAO WANG, AND JOHN ANDERSON
Google Research, Mountain View, CA 94043, USA
Corresponding author: Tianjian Lu (tianjianlu@google.com)

This work did not involve human subjects or animals in its research.

ABSTRACT In this work, we present two parallel algorithms for the large-scale discrete Fourier transform (DFT) on Tensor Processing Unit (TPU) clusters. The two parallel algorithms are associated with two DFT formulations: one formulation, denoted as KDFT, is based on the Kronecker product; the other is based on the famous Cooley-Tukey algorithm and phase adjustment, denoted as FFT. Both KDFT and FFT formulations take full advantage of TPU's strength in matrix multiplications. The KDFT formulation allows direct use of nonuniform inputs without additional step. In the two parallel algorithms, the same strategy of data decomposition is applied to the input data. Through the data decomposition, the dense matrix multiplications in KDFT and FFT are kept local within TPU cores, which can be performed completely in parallel. The communication among TPU cores is achieved through the one-shuffle scheme in both parallel algorithms, with which sending and receiving data takes place simultaneously between two neighboring cores and along the same direction on the interconnect network. The one-shuffle scheme is designed for the interconnect topology of TPU clusters, minimizing the time required by the communication among TPU cores. Both KDFT and FFT are implemented in TensorFlow. The three-dimensional complex DFT is performed on an example of dimension $8192 \times 8192 \times 8192$ with a full TPU Pod: the run time of KDFT is 12.66 seconds and that of FFT is 8.3 seconds. Scaling analysis is provided to demonstrate the high parallel efficiency of the two DFT implementations on TPUs.

INDEX TERMS Discrete fourier transform, fast fourier transform, parallel computing, tensorflow, tensor processing unit.

I. INTRODUCTION

The discrete Fourier transform (DFT) is critical in many scientific and engineering applications, including time series and waveform analyses, convolution and correlation computations, solutions to partial differential equations, density function theory in first-principle calculations, spectrum analyzer, synthetic aperture radar, computed tomography, magnetic resonance imaging, and derivatives pricing [1]–[4]. However, the computation efficiency of DFT is often the formidable bottleneck in handling large-scale problems due to the large data size and real-time-processing requirement [5], [6]. In general, efforts on enhancing the computation efficiency of DFT fall into two categories: seeking fast algorithms and adapting the fast algorithms to hardware accelerators. One breakthrough of the fast-algorithm category is the Cooley-Tukey algorithm [7], also known as the fast Fourier transform (FFT), which reduces the complexity of N-point DFT from $O(N^2)$ to $O(N \log N)$. The Cooley-Tukey algorithm assuming that the number of data is a power of two is known as the Radix-2 algorithm and followed by Mixed-Radix [3] and Split-Radix [8] algorithms.

In addition to the fast algorithms, the performance of hardware accelerators has been steadily driving the efficiency enhancement of DFT computation: the first implementation of the FFT algorithm was realized on ILLIAC IV parallel computer [9], [10]; over the years, the DFT computation has been adapted to both shared-memory [11], [12] and distributed-memory architectures [13]–[17]. The advancement of hardware accelerators has enabled massive parallelization for DFT computation. One such example is deploying the FFT computation on manycore processors [18]. Another example is implementing the FFT algorithm on clusters of graphics processing units (GPUs) [19]. A GPU cluster contains a number of nodes (machines) and within each node, GPUs are connected through PCIe, a high-speed serial interface. The Cooley-Tukey algorithm and its variants often require a large number of memory accesses per arithmetic operation such that the bandwidth limitation of PCIe becomes the computation bottleneck of the overall performance of

The associate editor coordinating the review of this manuscript and approving it for publication was Daniel Grosu.

FFT on GPU clusters. Prior to the recent development of novel high-speed interconnects such as NVLink [20], [21], many efforts related to the GPU-accelerated DFT computation are spent on minimizing the PCIe transfer time [3], [22]. It is worth mentioning that the route of algorithm-hardware co-design has also been taken with Field Programmable Gate Arrays (FPGAs) to optimize the configurations of a customized hardware accelerator for high-performance computing of DFT [23]–[25].

The recent success of machine learning (ML), or deep learning (DL) in particular, has spurred a new wave of hardware accelerators. In many ML applications, it becomes increasingly challenging to balance the performance-cost-energy of processors with the growth of data. Domain-specific hardware is considered as a promising approach to achieve this [26]. One example of the domain-specific hardware is Google's Tensor Processing Unit (TPU) [27]. As a reference, TPU v3 provides 420 teraflops and 128 GiB high-bandwidth memory (HBM) [28]. In witnessing how DFT computation benefits from the development of hardware accelerators, it is tempting to ask whether TPU can empower the large-scale DFT computation. It is plausible with the following four reasons: (1) TPU is an ML application-specific integrated circuit (ASIC), devised for neural networks (NNs); NNs require massive amounts of multiplications and additions between the data and parameters and TPU can handle these computations in terms of matrix multiplications in a very efficient manner [29]; similarly, DFT can also be formulated as matrix multiplications between the input data and the Vandemonde matrix; (2) TPU chips are connected directly to each other with dedicated, high-speed, and low-latency interconnects, bypassing host CPU or any networking resources; therefore, the large-scale DFT computation can be distributed among multiple TPUs with minimal communication time and hence very high parallel efficiency; (3) the large capacity of the in-package memory of TPU makes it possible to handle large-scale DFT efficiently; and (4) TPU is programmable with software front ends such as TensorFlow [30] and PyTorch [31], both of which make it straightforward to implement the parallel algorithms of DFT on TPUs. In fact, all the aforementioned four reasons have been verified in the high-performance Monte Carlo simulations on TPUs [32], [33].

In this work, we designed and implemented two parallel algorithms for DFT on TPUs: one is based on the Kronecker product, to be specific, dense matrix multiplications between the input data and the Vandermonde matrix, denoted as KDFT in this work; and the other is based on the Cooley-Tukey algorithm and phase adjustment, denoted as FFT in this work. For a N-point DFT, the computation complexity of KDFT is $O(N^2)$, whereas that of FFT is $O(N \log N)$. Both parallel algorithms take full advantage of TPU's strength in matrix multiplications. It is worth mentioning that KDFT takes in nonuniform input data without additional steps. The nonuniform Fourier transform has important applications in signal processing, medical imaging, numerical solutions of partial differential equations, and machine learning [34]–[37]. Both KDFT and FFT use the same strategy of data decomposition over the input data, through which the dense matrix multiplications are kept local within TPU cores and can be performed completely in parallel. Because of the data decomposition, each TPU core contains partial input data such that communication is required to share the data among cores. The communication among TPU cores is achieved through the one-shuffle scheme in both parallel algorithms, with which sending and receiving data takes place simultaneously between two neighboring cores and along the same direction on the interconnect network. The one-shuffle scheme is designed for the interconnect topology of TPU clusters, minimizing the time required by the communication among TPU cores. Scaling analysis is provided to demonstrate the high parallel efficiency of the proposed two algorithms of DFT on TPUs.

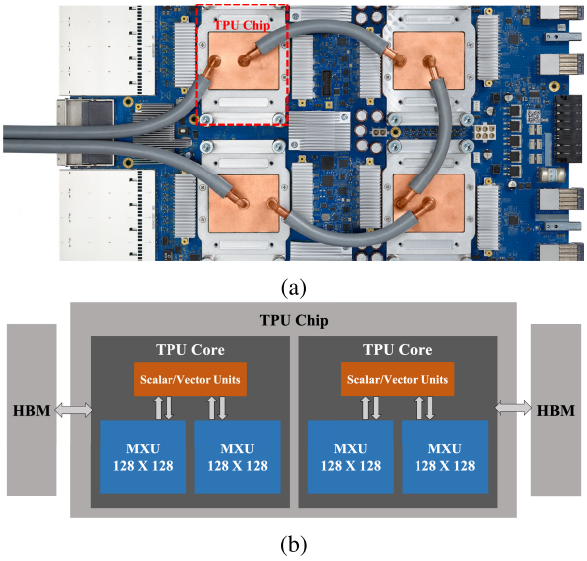


FIGURE 1. (a) TPU v3 has four chips on the same board and (b) each chip contains two cores.

II. TPU SYSTEM ARCHITECTURE

In this section, we provide an overview of the TPU system architecture on both the hardware and software components.

A. HARDWARE ARCHITECTURE

Figure 1 shows one TPU board or unit: there are four TPU chips on the same board; each chip has two cores; and each core contains the scalar, vector, and matrix units (MXU). Structured as a 128 × 128 systolic array, MXU provides the bulk of compute power of a TPU chip and handles 16 K multiply-accumulate (MAC) operations in one clock cycle. The inputs and outputs of MXU are float32 and the MAC operations on MXU are performed with bfloat16 [38]. However, one float32 number can be decomposed into multiple bfloat16 numbers and with appropriate accumulations, high-precision MAC operation can be achieved [39].

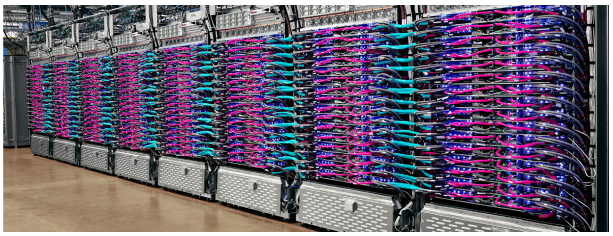


FIGURE 2. TPU v3 Pod in a data center.

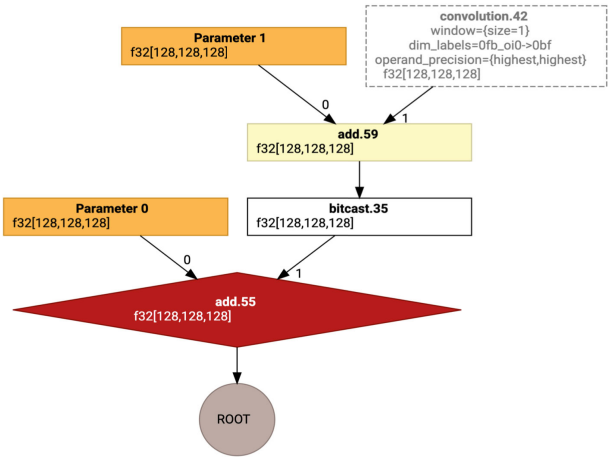


FIGURE 3. A computation graph of TensorFlow operations.

The implementation of both parallel algorithms in this work leverages the strategy of decomposition and accumulation and achieves the precision of float32. As shown in Fig. 1(b), each TPU core has 16 GiB high-bandwidth memory (HBM). The large capacity of in-package memory makes it possible to solve large-scale problems in a highly efficient manner. TPU is designed as a coprocessor on the I/O bus: each board shown in Fig. 1(a) is paired with one host server consisting of CPU, RAM, and hard disk; TPU executes the instructions sent from CPU on the host server through PCIe.

Figure 2 shows a TPU v3 Pod in a data center where a total number of 2048 cores are connected to each other. In a Pod configuration, TPU chips are connected through dedicated high-speed interconnects of very low latency. The interconnect topology is a two-dimensional (2D) toroidal mesh with each chip connected to its four nearest neighbors such that the communication takes place in four directions. These interconnects bypass the CPU networking resources and go from chip to chip directly. In our implementations, we have further optimized the communication strategy to take advantage of the TPU interconnect topology.

B. SOFTWARE ARCHITECTURE

TensorFlow is used to program TPUs in this work. A TensorFlow client converts the TensorFlow operations into a computational graph. A sample computation graph performing addition and convolution operations is shown in Fig. 3. The TensorFlow client sends the graph to a TensorFlow server. The TensorFlow server partitions the computational graph into portions that run on TPU and CPU, respectively. If multiple TPUs are to be employed, the graph is marked for replication. The TensorFlow server then compiles the sub-graph that runs on TPUs into a high level optimizer (HLO) program and invokes the accelerated linear algebra (XLA) compiler. The XLA compiler takes in the HLO program and converts it into a low level optimizer (LLO) program, which is effectively the assembly code for TPUs. Both the generation and compilation of the computational graph occur on the host server. The compiled LLO code is loaded onto TPUs for execution from the host server through PCIe.

The memory usage of a TPU is determined at compile time. Because both the hardware structure of MXU and the memory subsystem on a TPU core prefer certain shapes of a tensor variable involved in an operation, the XLA compiler performs the data layout transformations in order for the hardware to efficiently process the operation. If a tensor variable does not align with the preferred shape, the XLA compiler pads zeros along one dimension to make it a multiple of eight and the other dimension to a multiple of 128. Zero padding under-utilizes the TPU core and leads to sub-optimal performance, which should be taken into account in the implementation of the parallel algorithms on TPUs.

III. DFT FORMULATIONS

In this section, we provide the detailed formulations for both KDFT and FFT.

A. KDFT FORMULATION

The KDFT formulation starts from the general form of DFT, which is defined as

$$X_k \triangleq X(z_k) = \sum_{n=0}^{N-1} x_n z_k^{-n}, \tag{1}$$

where $\triangleq$ denotes "defined to be", x_n represents the input, and $\{z_k\}_{k=0}^{N-1}$ are N distinctly and arbitrarily sampled points on the z-plane. Equation (1) can be rewritten into the matrix form

$$\{X\} = [V]\{x\}, \tag{2}$$

where

$$\{X\} = (X(z_0), X(z_1), \cdots, X(z_{N-1}))^{\mathrm{T}},$$
$$\{x\} = (x_0, x_1, \cdots, x_{N-1})^{\mathrm{T}},$$

and

$$[V] = \begin{pmatrix} 1 & z_0^{-1} & z_0^{-2} & \cdots & z_0^{-(N-1)} \\ 1 & z_1^{-1} & z_1^{-2} & \cdots & z_1^{-(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & z_{N-1}^{-1} & z_{N-1}^{-2} & \cdots & z_{N-1}^{-(N-1)} \end{pmatrix}. \tag{3}$$

Note that $[V]$ is the Vandermonde matrix of dimension $N \times N$. Computing the inverse DFT is equivalent to solving the linear system in Equation (2). In the situation when $\{z_k\}_{k=0}^{N-1}$ are uniformly sampled on the z-plane, the Vandermonde matrix $[V]$ becomes unitary and contains the roots of unity.

The general form of a 2D DFT can be written as

$$X(z_{1k}, z_{2k}) = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x(n_1, n_2) z_{1k}^{-n_1} z_{2k}^{-n_2}, \tag{4}$$

where $[x]$ has the dimension of $N_1 \times N_2$ and $\{(z_{1k}, z_{2k})\}_{k=0}^{N_1N_2-1}$ represents the set of distinctly and arbitrarily sampled points in (z_1, z_2) space. It is worth mentioning that the sampling with (z_{1k}, z_{2k}) has to ensure the existence of the inverse DFT. If the sampling is performed on rectangular grids, Equation (4) can be rewritten into the matrix form as

$$[X] = [V_1]\,[x]\,[V_2]^{\mathrm{T}}, \tag{5}$$

where

$$[X] = \begin{pmatrix} X(z_{10}, z_{20}) & X(z_{10}, z_{21}) & \cdots & X(z_{10}, z_{2,N_2-1}) \\ X(z_{11}, z_{20}) & X(z_{11}, z_{21}) & \cdots & X(z_{11}, z_{2,N_2-1}) \\ \vdots & \vdots & \ddots & \vdots \\ X(z_{1,N_1-1}, z_{20}) & X(z_{1,N_1-1}, z_{21}) & \cdots & X(z_{1,N_1-1}, z_{2,N_2-1}) \end{pmatrix}, \tag{6}$$

$$[x] = \begin{pmatrix} x(0,0) & x(0,1) & \cdots & x(0,N_2-1) \\ x(1,0) & x(1,1) & \cdots & x(1,N_2-1) \\ \vdots & \vdots & \ddots & \vdots \\ x(N_1-1,0) & x(N_1-1,1) & \cdots & x(N_1-1,N_2-1) \end{pmatrix}, \tag{7}$$

$$[V_1] = \begin{pmatrix} 1 & z_{10}^{-1} & z_{10}^{-2} & \cdots & z_{10}^{-(N_1-1)} \\ 1 & z_{11}^{-1} & z_{11}^{-2} & \cdots & z_{11}^{-(N_1-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & z_{1,N_1-1}^{-1} & z_{1,N_1-1}^{-2} & \cdots & z_{1,N_1-1}^{-(N_1-1)} \end{pmatrix}, \tag{8}$$

and

$$[V_2] = \begin{pmatrix} 1 & z_{20}^{-1} & z_{20}^{-2} & \cdots & z_{20}^{-(N_2-1)} \\ 1 & z_{21}^{-1} & z_{21}^{-2} & \cdots & z_{21}^{-(N_2-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & z_{2,N_2-1}^{-1} & z_{2,N_2-1}^{-2} & \cdots & z_{2,N_2-1}^{-(N_2-1)} \end{pmatrix}. \tag{9}$$

Note that both $[V_1]$ and $[V_2]$ are Vandermonde matrices of dimensions $N_1 \times N_1$ and $N_2 \times N_2$, respectively. One can further lump $[x]$ into a vector such that Equation (5) can be written into the same matrix form as Equation (2), in which $[V]$ for the 2D DFT is expressed as

$$[V] = [V_1] \otimes [V_2], \tag{10}$$

where $\otimes$ denotes the Kronecker product [40].

Similarly, the three-dimensional (3D) DFT defined by

$$X(z_{1k}, z_{2k}, z_{3k}) = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \sum_{n_3=0}^{N_3-1} x(n_1, n_2, n_3) z_{1k}^{-n_1} z_{2k}^{-n_2} z_{3k}^{-n_3} \tag{11}$$

can be rewritten into the matrix form as

$$\{X\} = [V_1] \otimes [V_2] \otimes [V_3]\,\{x\}, \tag{12}$$

where

$$[V_j] = \begin{pmatrix} 1 & z_{j0}^{-1} & z_{j0}^{-2} & \cdots & z_{j0}^{-(N_j-1)} \\ 1 & z_{j1}^{-1} & z_{j1}^{-2} & \cdots & z_{j1}^{-(N_j-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & z_{j,N_j-1}^{-1} & z_{j,N_j-1}^{-2} & \cdots & z_{j,N_j-1}^{-(N_j-1)} \end{pmatrix}, \quad j \in \{1, 2, 3\} \tag{13}$$

For the 3D DFT defined in Equation (12), the sampling is performed on rectangular grids in (z_1, z_2, z_3) space and the Vandermonde matrix $[V_3]$ has the dimension of $N_3 \times N_3$. It can be seen that the Kronecker product bridges the gap between the matrix and tensor operations, through which the contraction between a rank-2 tensor and another rank-3 tensor in the 3D DFT can be formulated as matrix multiplications. The KDFT formulation can be easily extended to higher dimensions.

B. FFT FORMULATION

The FFT formulation starts with

$$X_k \triangleq \sum_{n=0}^{N-1} x_n e^{-j2\pi \frac{nk}{N}}, \tag{14}$$

in which x_n represents the input data and the frequency sampling has to be uniform. The global index n in Equation (14) can be expressed as

$$n = Pl + \beta, \tag{15}$$

where $l = 0, 1, \cdots, \frac{N}{P} - 1$ and $\beta = 0, 1, \cdots, P-1$. With Equation (15), Equation (14) can be rewritten as

$$X_k \triangleq \sum_{n=0}^{N-1} x_{(Pl+\beta)} e^{-j2\pi \frac{(Pl+\beta)k}{N}} \tag{16}$$

$$= \sum_{\beta=0}^{P-1} e^{-j2\pi \frac{\beta k}{N}} \left(\sum_{l=0}^{\frac{N}{P}-1} x_{(Pl+\beta)} e^{-j2\pi \frac{lk}{\frac{N}{P}}} \right). \tag{17}$$

In Equation (17),

$$\widetilde{X}_k = \sum_{l=0}^{\frac{N}{P}-1} x_{(Pl+\beta)} e^{-j2\pi \frac{lk}{\frac{N}{P}}} \tag{18}$$

is computed with the famous Cooley-Tukey algorithm locally on individual cores. Prior to the local transform, the gathering of the input among the cores is required, which arises from the global indexing in Equation (15). After the local transform, the phase adjustment needs to applied, which is formulated as matrix multiplications similar to that in Equation (2). Higher dimensional FFT such as 2D and 3D can be achieved by repeating this one-dimensional (1D) scheme along the corresponding dimensions.

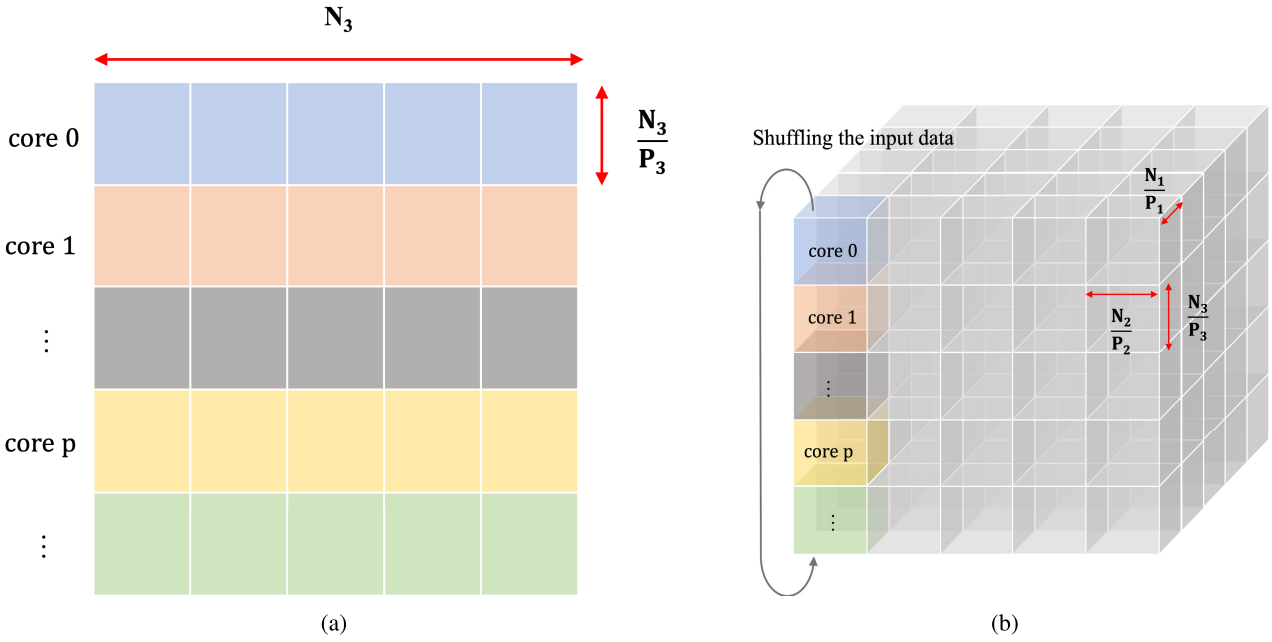


FIGURE 4. Through the data decomposition with the TPU computation shape (P_1, P_2, P_3), each TPU core contains (a) the Vandermonde matrix of dimension $\frac{N_i}{P_i} \times N_i$, $i = 1, 2, 3$ and (b) the block of input data of dimension $\frac{N_1}{P_1} \times \frac{N_2}{P_2} \times \frac{N_3}{P_3}$ for a 3D DFT. The core index is denoted by $p = 0, 1, \cdots, P_i - 1$, $i = 1, 2, 3$. The Fourier transform along the third dimension requires shuffling the blocks of the input data among the cores that are grouped by the third dimension of the computation shape P_3.

IV. IMPLEMENTATION OF THE PARALLEL ALGORITHMS

In this section, we provide details for implementing both KDFT and FFT on TPUs, including the data decomposition and the one-shuffle scheme.

A. DATA DECOMPOSITION

The data decomposition applied to the input data localizes the matrix multiplications on individual cores, which is critical to achieve high parallel efficiency on TPUs. For a 3D DFT, the data decomposition is applied to the input data along all three dimensions. The decomposition is based on a TPU computation shape (P_1, P_2, P_3) where P_1, P_2, and P_3 denote the number of TPU cores along the first, the second, and the third dimension, respectively. Given the TPU computation shape (P_1, P_2, P_3) and the input data of dimension $N_1 \times N_2 \times N_3$, each TPU core contains a data block of dimension $\frac{N_1}{P_1} \times \frac{N_2}{P_2} \times \frac{N_3}{P_3}$ as shown in Fig. 4(a). The data decomposition is also applied to the Vandermonde matrix and is along the frequency domain. As shown in Fig. 4(b), each core has a slice of the Vandermonde matrix with dimension $\frac{N_i}{P_i} \times N_i$, $i = 1, 2, 3$. It is also shown in Fig. 4 that each core is assigned an index p along each dimension and $p_i = 0, 1, \cdots, P_i - 1$, where $i = 1, 2, 3$. With the proposed data decomposition, the dense matrix multiplications of both KDFT and FFT are kept local within individual TPU cores and performed completely in parallel.

B. ONE-SHUFFLE SCHEME

The one-shuffle scheme described in Algorithm 1 is used by both KDFT and FFT. We use KDFT to illustrate the one-shuffle scheme. There are two major operations in KDFT: the tensor contraction between the Vandermonde matrix and the input data; and the communication among TPU cores to send and receive the data. The tensor contraction is based on `einsum` and the communication among TPU cores is with `collective_permute`. After one operation of tensor contraction, the block of the input data initially assigned on a TPU core is shuffled once with its neighboring core. The one-time shuffling takes place along the same direction on the interconnect network. As shown in Fig. 4(b), the DFT along the third dimension requires shuffling the blocks of the input data among the cores that are grouped by the third dimension of the computation shape P_3. In FFT, the one-shuffle scheme is used for applying the phase adjustment, in which the Vandermonde matrix in Fig. 4(a) contains the phase-shift information.

With the one-shuffle scheme, sending and receiving data takes place simultaneously between two neighboring cores and along the same direction on the 2D toroidal network. The one-shuffle scheme minimizes the communication time and leads to high parallel efficiency, which will be demonstrated through the parallel efficiency analysis in the following sections.

C. IMPLEMENTATION OF THE PARALLEL ALGORITHM FOR KDFT

Figure 5 illustrates the one-shuffle scheme in the parallel algorithm based on KDFT with a 3D example. We use c_0, c_1, and c_2 to denote three adjacent TPU cores, the operations on which are highlighted with three different colors

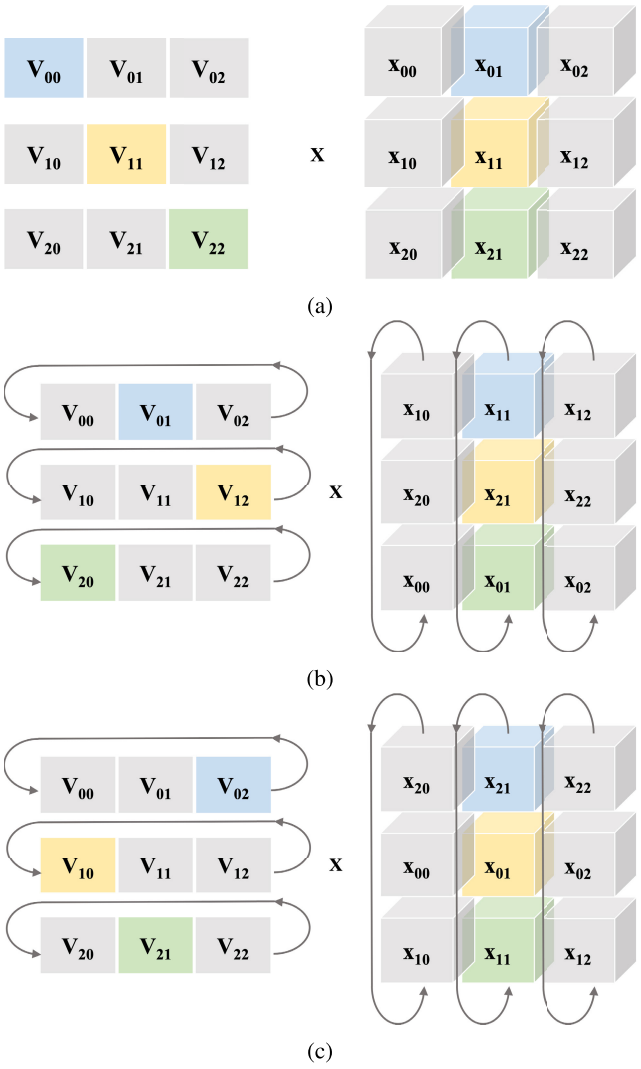


FIGURE 5. The one-shuffle scheme in the parallel algorithm based on KDFT is illustrated with a 3D example. We use c_0, c_1, and c_2 to denote three adjacent cores, the operations on which are highlighted with blue, yellow, and green, respectively. The data decomposition results in the block of the input data x_{01} and the slice of the Vandermonde matrix $[V_{00}, V_{01}, V_{02}]$ on core c_0, x_{11} and $[V_{10}, V_{11}, V_{12}]$ on core c_1, and x_{21} and $[V_{20}, V_{21}, V_{22}]$ on core c_2. The steps involved in the one-shuffle scheme are: (a) computing $V_{00} \cdot x_{01}$ on core c_0, $V_{11} \cdot x_{11}$ on core c_1, and $V_{22} \cdot x_{21}$ on core c_2 with $\cdot$ representing the operation of tensor contraction; (b) collectively permuting the inputs between two neighboring cores such that x_{11} on core c_0, x_{21} on core c_1, and x_{01} on core c_2 and computing $V_{01} \cdot x_{11}$ on core c_0, $V_{12} \cdot x_{21}$ on core c_1, and $V_{20} \cdot x_{01}$ on core c_2; (c) collectively permuting the inputs such that x_{21} on core c_0, x_{01} on core c_1, and x_{11} on core c_2 and computing $V_{02} \cdot x_{21}$ on core c_0, $V_{10} \cdot x_{01}$ on core c_1, and $V_{21} \cdot x_{11}$ on core c_2. The `collective_permute` operation in shuffling the input between neighboring TPU cores is with source-target pairs (c_1, c_0), (c_2, c_1), and (c_0, c_2) in the form of (source, target).

accordingly in Fig. 5. After the data decomposition, core c_0 contains a block of the input data x_{01} and a slice of the Vandermonde matrix $[V_{00}, V_{01}, V_{02}]$, core c_1 contains x_{11} and $[V_{10}, V_{11}, V_{12}]$, and core c_2 contains x_{21} and $[V_{20}, V_{21}, V_{22}]$. Note that the subscripts appearing in the block of the input data x_{p_1,p_2,p_3} are core indices. For simplicity, we ignore the core index on the third dimension, which is the same

Algorithm 1 The One-Shuffle Scheme

```
function one_shuffle(v, x, core_idx, num_cores, src_tgt_pairs)
    iteration_idx ← 0
    slice_idx ← core_idx
    x_out ← einsum(v[slice_idx], x)
    slice_idx ← mod(slice_idx + 1, num_cores)
    while iteration_idx < num_cores − 1 do
        x ← collective_permute(x, src_tgt_pairs)
        x_out ← x_out + einsum(v[slice_idx], x)
        slice_idx ← mod(slice_idx + 1, num_cores)
        iteration_idx ← iteration_idx + 1
    return x_out
```

across cores c_0, c_1, and c_2. With three `einsum` and two `collective_permute` operations, one obtains the partial DFT written as $V_{00} \cdot x_{01} + V_{01} \cdot x_{11} + V_{02} \cdot x_{21}$ on core c_0, where $\cdot$ represents the tensor contraction. The steps taken by the partial DFT computation along one dimension are as follows:

1. apply `einsum` to compute $V_{00} \cdot x_{01}$ on core c_0, $V_{11} \cdot x_{11}$ on core c_1, and $V_{22} \cdot x_{21}$ on core c_2 as shown in Fig. 5(a);
2. apply `collective_permute` to shuffle the input between neighboring TPU cores with source-target pairs (c_1, c_0), (c_2, c_1), and (c_0, c_2) in the form of (source, target) such that core c_0 contains x_{11}, core c_1 contains x_{21}, and core c_2 contains x_{01} as shown in Fig. 5(b);
3. apply `einsum` to compute $V_{01} \cdot x_{11}$ on core c_0, $V_{12} \cdot x_{21}$ on core c_1, and $V_{20} \cdot x_{01}$ on core c_2 and add the results from step 1;
4. apply `collective_permute` with source-target pairs (c_1, c_0), (c_2, c_1), (c_0, c_2), after which core c_0 contains x_{21}, core c_1 contains x_{01}, and core c_2 contains x_{11} as shown in Fig. 5(c);
5. apply `einsum` to compute $V_{02} \cdot x_{21}$ on core c_0, $V_{10} \cdot x_{01}$ on core c_1, and $V_{21} \cdot x_{11}$ on core c_2 and add the results from step 3.

D. IMPLEMENTATION OF THE PARALLEL ALGORITHM FOR FFT

Figure 6 illustrates the four steps of a 1D FFT on TPUs: the data decomposition, the gathering of the input, the local transform, and the phase adjustment. Figure 6(a) shows the input assigned to individual cores after the data decomposition. In order to achieve an in-order transform, to be specific, the ordering of the obtained results in the transform domain remains the same as that in the input, it requires a local re-ordering of the input prior to the transform, which can be achieved through `einsum`. The re-ordering operation is local within individual TPU cores. The gathering of the input as shown in Fig. 6(b) is implemented with `all_to_all`. After the gathering, the Cooley-Tukey-algorithm-based transform is performed locally on individual cores, which is implemented with `tf.signal.fft` as shown in Fig. 6(c).

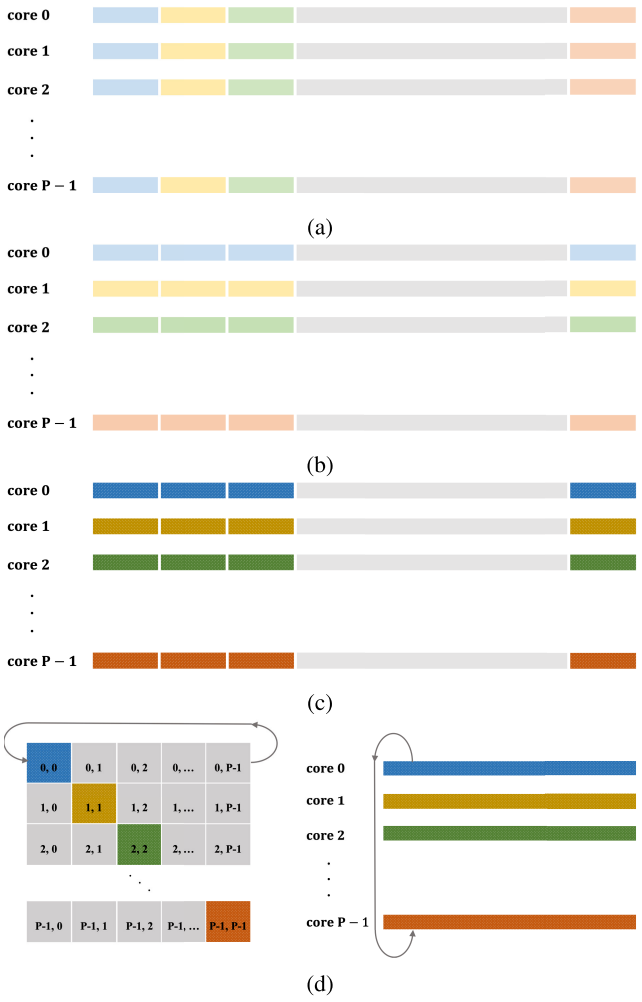


FIGURE 6. Four steps for a 1D FFT on TPUs: (a) the data decomposition, (b) the gathering of the input, (c) the transform performed locally on individual cores, and (d) the phase adjustment through the one-shuffle scheme. The pair of the indices in (d) represents the source and target pairs used by `collective_permute`. A 3D FFT consists of three 1D FFT operations along each of the three dimensions: for each 1D FFT, it follows the same steps; and the only difference is that the blocks of input/transformed data highlighted by color are 3D tensors..

At last, the locally-obtained transform results are summed over all the cores with phase adjustment, which is achieved through the one-shuffle scheme. It can be seen that in the 1D FFT on TPUs, the communication is required in gathering the input and applying the phase adjustment.

Higher dimensional FFT such as 2D and 3D consists of multiple 1D FFT operations along the corresponding dimensions. For example, a 3D FFT consists of three 1D FFT operations along each of the three dimensions: for each 1D FFT, it follows the same steps illustrated in Fig. 6; and the only difference is that the blocks of input/transformed data highlighted by color are 3D tensors.

V. PARALLEL EFFICIENCY ANALYSIS

In this section, both the strong and weak scaling analyses are provided to demonstrate the efficiency of the proposed two DFT parallel algorithms on TPUs. For the strong scaling analysis, the problem size is kept the same as proportionally more TPU cores are employed. For the weak scaling analysis, the number of TPU cores remains the same as the problem size increases. The TPU profiling tool [28], which provides information on the utilization of the hardware and the efficiency of individual operations at the program level is used to analyze the performance of DFT on TPUs. A screenshot of the trace viewer from the TPU profiling tool is shown in Fig. 7. With the profiling tool, one can breakdown the operations at the HLO level, which is quite helpful in identifying the bottleneck of the parallel efficiency and making improvements to the algorithm designs.

A. STRONG SCALING ANALYSIS OF 2D KDFT

Figure 8 shows the computation time of the 2D KDFT with up to 128 TPU cores on an example of dimension 8192×8192. It can be seen from Fig. 8 that a close-to-linear scaling of the computation time with respect to the number of TPU cores is achieved. As a reference, the ideal computation time from the linear scaling is provided in Fig. 8, which is defined by

$$\text{ideal time} = \frac{T_2}{\frac{N_{\text{core}}}{2}}, \tag{19}$$

where T_2 denotes the total computation time with two TPU cores and N_{core} is the total number of TPU cores being used. As mentioned in the parallel implementation, the total computation time consists of two parts: the time of matrix multiplications, or `einsum` to be specific, and the communication time of sending and receiving the block of input data across TPU cores. It can be seen from Fig. 8 that the time of matrix multiplications scales linearly with respect to the total number of TPU cores. This is because the matrix multiplications are kept completely local within individual cores. The computation time of the 2D KDFT scales approximately linearly with respect to the number of TPU cores, with the gap between the actual and the ideal computation time arising from the communication among TPU cores.

B. STRONG SCALING ANALYSIS OF 3D KDFT

The parallel efficiency of the 3D KDFT is demonstrated through an example of dimension $2048 \times 2048 \times 2048$. Similar to the 2D case, the problem size is also fixed as proportionally more TPU cores are employed. The total computation time is depicted in Fig. 9. As a reference, the ideal computation time from linear scaling is provided in Fig. 9, which is defined by

$$\text{ideal time} = \frac{T_{32}}{\frac{N_{\text{core}}}{32}}, \tag{20}$$

where T_{32} denotes the total computation time with 32 TPU cores. It can be seen from Fig. 9 that the computation time scales approximately linearly with respect to the number of TPU cores.

The gap between the actual and the ideal computation time in the 3D case also results from the communication among

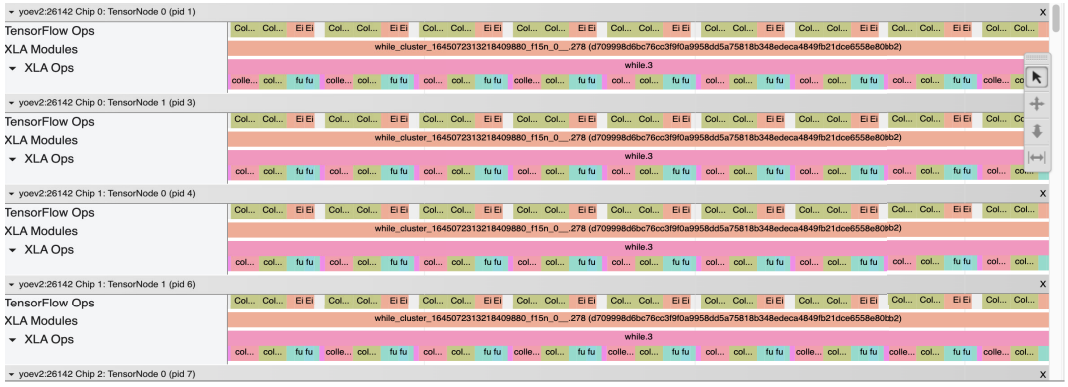


FIGURE 7. A screenshot of the trace viewer from the TPU profiling tool.

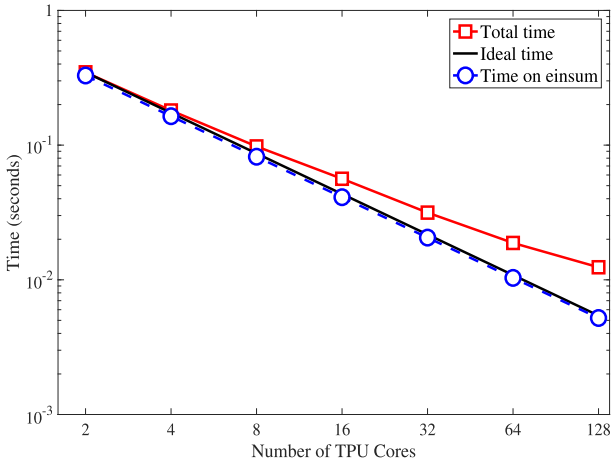


FIGURE 8. The computation time of the 2D KDFT with up to 128 TPU cores on an example of dimension 8192 × 8192.

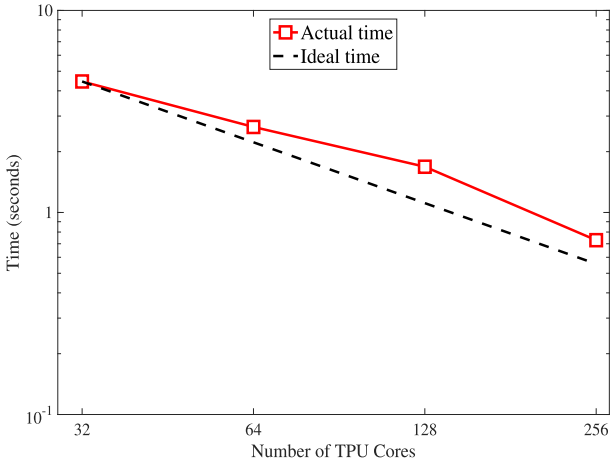


FIGURE 9. The computation time of the 3D KDFT with up to 256 TPU cores on an example of dimension 2048 × 2048 × 2048.

TPU cores. As mentioned in the parallel implementation, the data decomposition is applied to the input data along all the three dimensions with a TPU computation shape. The computation shape in this example has the form of $(4, 4, n_2)$ with four TPU cores along the first dimension, four along the second dimension, and n_2 along the third dimension. It is indeed the number of TPU cores along the third dimension that varies in Fig. 9. For example, the computation shapes $(4, 4, 8)$ and $(4, 4, 16)$ are corresponding to 128 and 256 TPU cores, respectively. As the number of TPU cores along the third dimension doubles itself, the size of the input data contained on each core is reduced by half. As a result, the computation time associated with a single operation of `collective_permute` or `einsum` is also reduced by half, which is shown in Fig. 10. However, as more cores are being used, the total number of `collective_permute` operations increases. For example, it requires a total number of 15 `collective_permute` operations in the Fourier transform along the third dimension in the case of 256 TPU cores or with the TPU computation shape $(4, 4, 16)$, whereas only 7 `collective_permute` operations are required in the case of 128 TPU cores or with the TPU computation shape $(4, 4, 8)$. It can be seen that even though the time associated with one single `collective_permute` operation decreases when more TPU cores are used, the total communication time for the DFT along the third dimension does not change. This explains the gap between the total and the ideal computation time in Fig. 9.

C. STRONG SCALING ANALYSIS OF 3D FFT

The parallel efficiency of the 3D FFT is demonstrated through an example of dimension 2048 × 2048 × 2048. The problem size is fixed as proportionally more TPU cores are employed. The total computation time is depicted in Fig. 11. As a reference, the ideal computation time from linear scaling is provided in Fig. 11, which is defined by

$$\text{ideal time} = \frac{T_{16}}{\frac{N_{\text{core}}}{16}}, \tag{21}$$

where T_{16} denotes the total computation time with 16 TPU cores. As shown in Fig. 11 that close-to-linear scaling is observed in the 3D FFT computation on TPUs. The example

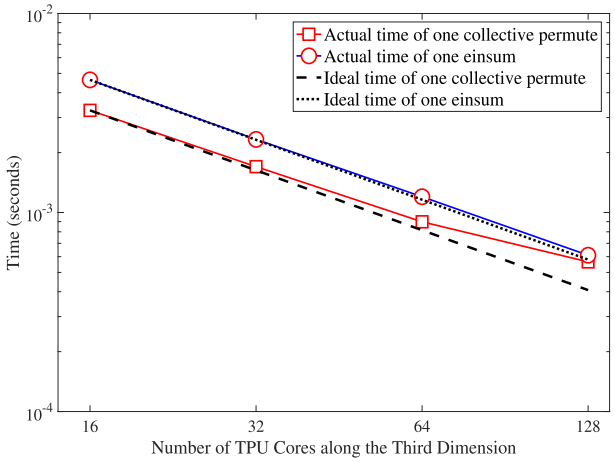


FIGURE 10. The computation time of one single operation of `collective_permute` and `einsum`, respectively, in the 3D KDFT along the third dimension with respect to the number of TPU cores.

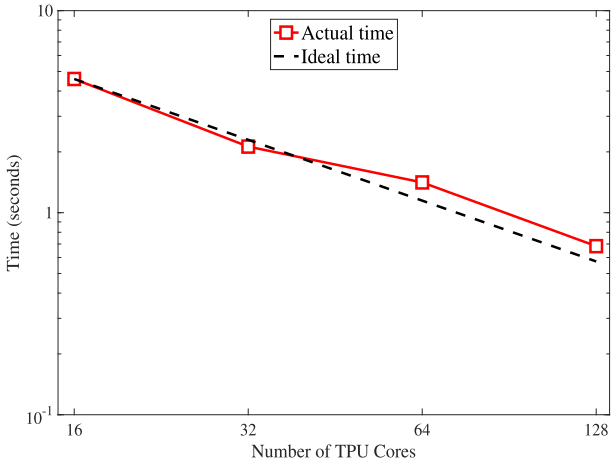


FIGURE 11. The computation time of the 3D FFT with up to 128 TPU cores on an example of dimension 2048 × 2048 × 2048.

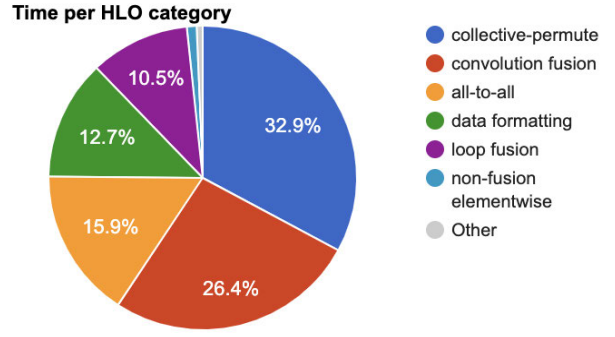


FIGURE 12. The breakdown of the computation time with the TPU profiling tool for the 3D FFT with 256 TPU cores on an example of dimension 2048 × 2048 × 2048.

of size 2048 × 2048 × 2048 is considered small for more than 128 cores. The breakdown of the total computation time from the TPU profiling tool is provided in Fig. 12. It can be seen that the communication time consisting of both `all_to_all` and `collective_permute` starts dominating the total computation time.

D. 3D KDFT AND FFT ON A FULL TPU POD

In addition to the strong scaling analysis, the computation time of a few 3D DFT and FFT examples on a full TPU Pod with 2048 cores is provided in Table 1. The runtimes reported in this work are for complex transforms. As a reference, the runtime of a real FFT for the problem size 8192 × 8192 × 8192 on 2048 nodes of Fujitsu PRIMERGY CX1640 M1 cluster is 5.36 seconds (converted from 10 TFlops) [41].

TABLE 1. Computation time of the 3D KDFT and FFT on a full TPU Pod with 2048 TPU cores.

No.	Problem size	Time (seconds)	
		KDFT	FFT
1	8192 × 8192 × 8192	12.66	8.30
2	4096 × 4096 × 4096	1.07	1.01
3	2048 × 2048 × 2048	0.120	0.118
4	1024 × 1024 × 1024	0.0220	0.0148

In Table 1, as the problem size increases from 2048 × 2048 × 2048 to 4096 × 4096 × 4096, the computation time of DFT increases 9.7 times. Similarly, the computation time of DFT increases 11.8 times when the problem size increases from 4096×4096×4096 to 8192×8192×8192. As a comparison, the computation time of FFT scales up approximately eight times as the problem size increases by eight times. For the two problems of sizes 2048 × 2048 × 2048 and 4096 × 4096 × 4096, the total computation time of KDFT is about the same as that of FFT. Given the computation complexity difference between KFDFT and FFT, it demonstrates TPU's strength in matrix multiplications.

VI. CONCLUSION AND DISCUSSION

In this work, we proposed and implemented two parallel algorithms of DFT on TPUs, to be specific, KDFT and FFT. The formulation of KDFT is based on the Kronecker product. The formulation of FFT is based on the Cooley-Tukey algorithm and the phase adjustment. Both formulations take full advantage of TPU's strength in matrix multiplications. The implementation is in TensorFlow. Through implementing the two parallel algorithms, TPU—the domain-specific hardware accelerator for machine learning applications—is employed in the parallel computing of large-scale DFT. The data decomposition adopted in both parallel algorithms enables the localization of the dense matrix multiplications on individual TPU cores, which can be performed completely in parallel. As for the communication among TPU

cores, the one-shuffle scheme is designed based on the TPU interconnect topology, with which sending and receiving data takes place simultaneously between two neighboring cores and along the same direction on the interconnect network. The one-shuffle scheme requires minimal communication time among TPU cores and achieves very high parallel efficiency. Scaling analysis is provided to understand the parallel efficiency of the proposed DFT algorithms on TPUs.

With the demonstrated computation efficiency, the large-scale DFT on TPUs opens an array of opportunities for scientific computing. One possible future work is to integrate the DFT on TPUs with medical image reconstruction, where nonuniform Fourier transform is extensively used. Another future work is to extend the two proposed algorithms into a framework and address large-scale DFT of higher dimensions. Finally, the precision of matrix multiplications in this work can be improved from float32 to float64.

ACKNOWLEDGMENT

The authors would like to thank David Majnemer, Reid Tatge, Dehao Chen, Yusef Shafi, Damien Pierce, James Lottes, Matthias Ihme, and Rene Salmon for valuable discussions and helpful comments, which have greatly improved the paper.

REFERENCES

[1] R. N. Bracewell and R. N. Bracewell, *The Fourier Transform and Its Applications*, vol. 31999. New York, NY, USA: McGraw-Hill, 1986.

[2] A. Grama, V. Kumar, A. Gupta, and G. Karypis, *Introduction to Parallel Computing*. London, U.K.: Pearson, 2003.

[3] D. Takahashi, *Fast Fourier Transform Algorithms for Parallel Computers*. Singapore: Springer, 2019.

[4] R. Cont, *Frontiers in Quantitative Finance: Volatility and Credit Risk Modeling*, vol. 519. Hoboken, NJ, USA: Wiley, 2009.

[5] G. B. Giannakis, F. Bach, R. Cendrillon, M. Mahoney, and J. Neville, "Signal processing for big data [from the guest editors]," *IEEE Signal Process. Mag.*, vol. 31, no. 5, pp. 15–16, Sep. 2014.

[6] E. Olshannikova, A. Ometov, Y. Koucheryavy, and T. Olsson, "Visualizing big data with augmented and virtual reality: Challenges and research agenda," *J. Big Data*, vol. 2, no. 1, p. 22, Dec. 2015.

[7] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.*, vol. 19, no. 90, pp. 297–301, 1965.

[8] P. Duhamel and H. Hollmann, "Split radix FFT algorithm," *Electron. Lett.*, vol. 20, no. 1, pp. 14–16, 1984.

[9] G. Ackins, *Fast Fourier Transform Via ILLIAC IV*, document 4 146, Oct. 1968.

[10] J. E. Stevens, Jr., *A Fast Fourier Transform Subroutine for ILLIAC IV*, document 17, 1971.

[11] P. N. Swarztrauber, "Multiprocessor FFTs," *Parallel Comput.*, vol. 5, nos. 1–2, pp. 197–210, Jul. 1987.

[12] D. H. Bailey, "FFTs in external or hierarchical memory," *J. Supercomput.*, vol. 4, no. 1, pp. 23–35, Mar. 1990.

[13] A. Gupta and V. Kumar, "The scalability of FFT on parallel computers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 8, pp. 922–932, 1993.

[14] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proc. IEEE*, vol. 93, no. 2, pp. 216–231, Feb. 2005.

[15] D. Pekurovsky, "P3DFFT: A framework for parallel computations of Fourier transforms in three dimensions," *SIAM J. Sci. Comput.*, vol. 34, no. 4, pp. C192–C209, Jan. 2012.

[16] D. Takahashi, "An implementation of parallel 3-D FFT with 2-D decomposition on a massively parallel cluster of multi-core processors," in *Proc. Int. Conf. Parallel Process. Appl. Math.* Springer, 2009, pp. 606–614.

[17] D. T. Popovici, M. D. Schatz, F. Franchetti, and T. M. Low, "A flexible framework for parallel multi-dimensional DFTs," 2019, *arXiv:1904.10119*. [Online]. Available: http://arxiv.org/abs/1904.10119

[18] J. Kim. (2018) *Leveraging Optimized FFT on Intel Xeon Platforms*. https://www.alcf.anl.gov/support-center/training-assets/leveraging-optimized-fft-intel-xeon-platforms

[19] K. R. Roe, K. Hester, and R. Pascual. (2019). *Multi-GPU FFT Performance on Different Hardware Configurations*. [Online]. Available: https://developer.nvidia.com/gtc/2019/video/S9158

[20] D. Foley and J. Danskin, "Ultra-performance Pascal GPU and NVLink interconnect," *IEEE Micro*, vol. 37, no. 2, pp. 7–17, Mar. 2017.

[21] A. Li, S. Leon Song, J. Chen, J. Li, X. Liu, N. Tallent, and K. Barker, "Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect," 2019, *arXiv:1903.04611*. [Online]. Available: http://arxiv.org/abs/1903.04611

[22] Y. Chen, X. Cui, and H. Mei, "Large-scale FFT on GPU clusters," in *Proc. 24th ACM Int. Conf. Supercomput.*, 2010, pp. 315–324.

[23] S. K. Nag and H. K. Verma, "Method for parallel-efficient configuring an FPGA for large FFTs and other vector rotation computations," U.S. Patent 6 021 423, Feb. 1, 2000.

[24] M. Garrido, M. Acevedo, A. Ehliar, and O. Gustafsson, "Challenging the limits of FFT performance on FPGAs," in *Proc. Int. Symp. Integr. Circuits (ISIC)*, Dec. 2014, pp. 172–175.

[25] C.-L. Yu, K. Irick, C. Chakrabarti, and V. Narayanan, "Multidimensional DFT IP generator for FPGA platforms," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 58, no. 4, pp. 755–764, Apr. 2011.

[26] I. Stoica, D. Song, R. Ada Popa, D. Patterson, M. W. Mahoney, R. Katz, A. D. Joseph, M. Jordan, J. M. Hellerstein, J. E. Gonzalez, K. Goldberg, A. Ghodsi, D. Culler, and P. Abbeel, "A berkeley view of systems challenges for AI," 2017, *arXiv:1712.05855*. [Online]. Available: http://arxiv.org/abs/1712.05855

[27] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, and R. Boyle, "In-datacenter performance analysis of a tensor processing unit," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2017, pp. 1–12.

[28] *Cloud TPUs*. [Online]. Available: https://cloud.google.com/tpu/

[29] N. Jouppi. (2017). *Quantifying the Performance of the TPU, Our First Machine Learning Chip*. [Online]. Available: https://cloud.google.com/blog/products/gcp/quantifying-the-performance-of-the-tpu-our-first-machine-learning-chip

[30] Y. Wu *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016, *arXiv:1609.08144*. [Online]. Available: http://arxiv.org/abs/1609.08144

[31] N. Ketkar, "Introduction to PyTorch," in *Deep Learning With Python*. Springer, 2017, pp. 195–208.

[32] K. Yang, Y.-F. Chen, G. Roumpos, C. Colby, and J. Anderson, "High performance Monte Carlo simulation of Ising model on TPU clusters," in *Proc. Int. Conf. for High Perform. Comput., Netw., Storage Anal.*, 2019, pp. 1–15, doi: 10.1145/3295500.3356149.

[33] F. Belletti, D. King, K. Yang, R. Nelet, Y. Shafi, Y.-F. Chen, and J. Anderson, "Tensor processing units for financial Monte Carlo," 2019, *arXiv:1906.02818*. [Online]. Available: http://arxiv.org/abs/1906.02818

[34] S. Bagchi and S. K. Mitra, "The nonuniform discrete Fourier transform and its applications in filter design. I. 1-D," *IEEE Trans. Circuits Syst. II. Analog Digit. Signal Process.*, vol. 43, no. 6, pp. 422–433, Jun. 1996.

[35] S. Bagchi and S. K. Mitra, "The nonuniform discrete Fourier transform and its applications in filter design. II. 2-D," *IEEE Trans. Circuits Syst. II. Analog Digit. Signal Process.*, vol. 43, no. 6, pp. 434–444, Jun. 1996.

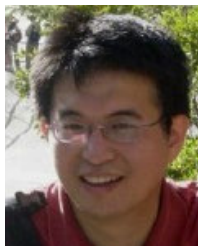
[36] J.-Y. Lee and L. Greengard, "The type 3 nonuniform FFT and its applications," *J. Comput. Phys.*, vol. 206, no. 1, pp. 1–5, Jun. 2005.

[37] A. Rahimi and B. Recht, "Random features for large-scale kernel machines," in *Proc. Adv. Neural Inf. Process. Syst.*, 2008, pp. 1177–1184.

[38] *Using Bfloat16 With Tensor Flow Models*. [Online]. Available: https://cloud.google.com/tpu/docs/bfloat16

[39] G. Henry, P. Tak Peter Tang, and A. Heinecke, "Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations," 2019, *arXiv:1904.06376*. [Online]. Available: http://arxiv.org/abs/1904.06376
[40] P. A. Regalia and M. K. Sanjit, "Kronecker products, unitary matrices and signal processing applications," *SIAM Rev.*, vol. 31, no. 4, pp. 586–613, Dec. 1989.
[41] D. Takahashi, "Implementation of parallel 3-D real FFT with 2-D decomposition on Intel Xeon Phi clusters," in *Proc. Int. Conf. Parallel Process. Appl. Math.* Cham, Switzerland: Springer, 2019, pp. 151–161.

TIANJIAN LU (Senior Member, IEEE) received the B.E. degree in electrical engineering from the National University of Singapore, Singapore, in 2010, and the M.S. and Ph.D. degrees in electrical engineering from the University of Illinois at Urbana–Champaign, in 2012 and 2016, respectively. Since 2016, he has been working with Google. His research interests include multiphysics simulation and machine learning. He was a recipient of the Best Student Paper Award (The First Place Winner) at the 31th International Review of Progress in ACES, Williamsburg, VA, USA, in 2015 and the Best Student Paper Award at the IEEE Electrical Design of Advanced Packaging and Systems (EDAPS), Honolulu, HI USA, in 2016. He was also a recipient of the P. D. Coleman Outstanding Research Award by the Department of Electrical and Computer Engineering, University of Illinois at Urbana–Champaign, in 2016.



YI-FAN CHEN was trained as an Applied Physicist with the Ph.D. degree and has worked in various fields, including photolithography, video processing, high performance computing. He has been with Google, since 2012.

BLAKE HECHTMAN is a Software Engineer with Google.

TAO WANG has been working on TPU performance, since 2018. He is a Software Engineer with Google.

JOHN ANDERSON is a Research Scientist with Google.