Received: 5 August 2022 | Revised: 22 December 2022 | Accepted: 27 December 2022
DOI: 10.4218/etrij.2022-0297

ORIGINAL ARTICLE

ETRI Journal WILEY

Large-scale 3D fast Fourier transform computation on a GPU

Jaehong Lee | **Duksu Kim**

Computer Science and Engineering, KOREATECH, Cheonan, Republic of Korea

Correspondence
Duksu Kim, Computer Science and Engineering, KOREATECH, Cheonan, Republic of Korea.
Email: bluekds@koreatech.ac.kr

Funding information
This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) through the Ministry of Education under Grant 2021R1I1A3048263 (High-Performance CGH Algorithms for Ultra-High Resolution Hologram Generation, 100%) and the Education and Research Promotion Program of Korea University of Technology and Education (KOREATECH), in 2023.

Abstract

We propose a novel graphics processing unit (GPU) algorithm that can handle a large-scale 3D fast Fourier transform (i.e., 3D-FFT) problem whose data size is larger than the GPU's memory. A 1D FFT-based 3D-FFT computational approach is used to solve the limited device memory issue. Moreover, to reduce the communication overhead between the CPU and GPU, we propose a 3D data-transposition method that converts the target 1D vector into a contiguous memory layout and improves data transfer efficiency. The transposed data are communicated between the host and device memories efficiently through the pinned buffer and multiple streams. We apply our method to various large-scale benchmarks and compare its performance with the state-of-the-art multicore CPU FFT library (i.e., fastest Fourier transform in the West [FFTW]) and a prior GPU-based 3D-FFT algorithm. Our method achieves a higher performance (up to 2.89 times) than FFTW; it yields more performance gaps as the data size increases. The performance of the prior GPU algorithm decreases considerably in massive-scale problems, whereas our method's performance is stable.

KEYWORDS
3D-FFT, fast Fourier transform, GPU, high-performance computing, parallel algorithm

1 | INTRODUCTION

The Fourier transform is a mathematical tool that represents waves that vary in time and space in their frequency domains. It has been extensively adopted to analyze the patterns of composite waves [1]. The fast Fourier transform (FFT) is a method used to accelerate the estimation of the discrete Fourier transform (DFT) (e.g., Cooley–Tukey algorithm), thus reducing the computational cost from $O(N^2)$ to $O(N\log N)$, where N is the size of the relevant vector [2]. Most Fourier transform libraries including fastest Fourier transform in the West (FFTW) [3, 4] and OneAPI [5] are based on FFT algorithms. One primary application of the FFT is convolution computation, which is a fundamental operation in the computer vision and machine learning domains. Based on the convolution theorem, the convolution operation can be represented as a set of FFT operations and element-wise matrix multiplications, thereby reducing the computational cost from $O(N^2)$ to $O(N\log N)$. For example, we can significantly decrease the computational time of the angular spectrum method (ASM) by calculating the wave propagation patterns of spatial fields using the convolution theorem [6].

This is an Open Access article distributed under the term of Korea Open Government License (KOGL) Type 4: Source Indication + Commercial Use Prohibition + Change Prohibition (http://www.kogl.or.kr/info/licenseTypeEn.do).
1225-6463/$ © 2023 ETRI

As the FFT process consists of a set of vectorial, element-wise multiplications, and additions, it is a suitable operation for ASM computing units [7]. A graphics processing unit (GPU) is a well-known computing device with a single-instruction multiple-data (SIMD) architecture, and recent GPUs have thousands of processing cores. Therefore, GPUs have been actively employed in many math libraries to accelerate the FFT process in software programs, such as MATLAB [8], CUDA fast Fourier transform [9], and OneAPI [5].

However, most FFT libraries need to load the entire dataset into the GPU memory before performing computations, and the GPU memory size limits the FFT problem size that they can handle. Recent commodity GPUs have limited memory space (in the range of 2 GB–24 GB compared with the system memory), with sizes ranging from tens of GB to several terabytes. Therefore, it is difficult to utilize the prior GPU-based FFT library for a large-scale FFT problem that requires GPU's high-computing capability. To overcome the limited GPU memory size issue, hybrid algorithms utilizing both a central processing unit (CPU) and GPU for FFT computation have been proposed [10]. Hybrid algorithms employ a divide-and-conquer strategy that splits the data into smaller parts. Then, it sends the data (in smaller parts) to the GPU for processing. Additionally, it places the original input data into pinned (or page-locked) memory, which is not paged by the operating system [11], to minimize the data transfer overhead between the CPU and GPU. However, we can only set a part of the system memory as pinned memory. Therefore, the data size that can be handled is still limited.

We propose a novel GPU-based three-dimensional FFT (3D-FFT) algorithm that can handle large-scale three-dimensional data exceeding the pinned-memory size (Section 4). Our method allocates the pinned memory as fit to the GPU memory size and uses the pinned-memory region as a buffer for communication between the CPU and GPU (Section 4.1). To improve data communication efficiency, we propose a 3D transposition method to convert nonsequential data access to sequential access using multiple streams (Section 4.2). Finally, we optimize our 3D-FFT algorithm by overlapping GPU computations and data communication (Section 4.3).

To verify the benefits of our approach, we implemented our method on a system with an Nvidia RTX 3090 with 24 GB memory. We then tested the performance of our 3D-FFT algorithm with a large-scale 3D complex data problem, whose size was much larger than the GPU memory size (Section 5). Compared with a state-of-the-art multicore, CPU-based FFT library (i.e., FFTW), our method achieved higher performance (up to 2.89 times). In addition, we found that the performance gap increased as the data size increased. We also implemented the state-of-the-art GPU-based 3D-FFT algorithm (i.e., *Chen*) [10] and compared its performance with that of our method. *Chen* performs better than our method for small-scale data (e.g., 16 GB). However, our algorithm achieved comparable performance to that of *Chen* for large-scale benchmarks (e.g., 32 GB–64 GB). Specifically, for massive-scale data (e.g., 128 GB), the performance of *Chen* decreased significantly, whereas our method worked stably for the same data. As a result, our method achieved higher performance than *Chen* for massive-scale data (up to 8.78 times). These results validate the efficiency of our method and show that our 3D-FFT algorithm is appropriate for large-scale 3D-FFT problems.

2 | RELATED WORK

Various parallel algorithms have been proposed that employ various types of parallel computing resources, including field-programmable gate arrays (FPGAs) [12, 13], multicore CPUs [4, 5, 14], and GPUs [9, 15–17].

The multicore CPU has traditionally been one of the most extensively adopted hardware for FFT computations. The FFTW [4] is a state-of-the-art FFT library that runs on a multicore CPU. The Math Kernel Library (MKL) [5] also supports high-performance FFT computation on Intel's CPU. Khokhriakov and others [14] proposed an optimized two-dimensional (2D)-FFT algorithm using these libraries and achieved performance improvements of up to 9.4 times with multicore CPUs compared with single-core CPUs. Most modern computing systems have one or more multicore CPUs. Generality is one of the main advantages of multicore CPU-based FFT algorithms. However, from the perspective of performance, using an accelerator (e.g., an FPGA or GPU) yields a considerably improved performance.

Recently, the GPU has also been actively employed to accelerate FFT computations. cuFFT [9] is a state-of-the-art GPU-based FFT library. Typically, it achieves much higher performance than CPU-based libraries. AMD also released the rocFFT library that runs on the Radeon Open Computing Platform (ROCm) [18]. Hu and others [19] optimized the memory-access pattern for FFT computations on ROCm. The proposed implementation achieved speed enhancements ranging from 25% to 250% compared with the speed of rocFFT.

However, GPU-based FFT libraries can handle problem sizes that are much smaller than CPU-based algorithms using the system memory because the size of the device memory (e.g., 2 GB–24 GB) is typically much smaller than the system memory (e.g., 16 GB to 1 TB). Gu and others [16] solved this issue using the

Cooley–Turkey algorithm [2]. These researchers reduced the size of the work unit that a GPU could process at a specific time period by decomposing the 1D-FFT problem into a set of sub-FFT problems using the Cooley–Tukey algorithm. They also decomposed multidimensional FFT problems into multidimensional sub-FFT problems and gathered the subarrays in contiguous memory space for efficient data transfer. As a result, they achieved performance improvements up to 2.3 and 1.53 times compared with CPU-based algorithms for 2D-FFT and 3D-FFT problems, respectively, in cases wherein problem sizes were larger than the device memory.

Modern Nvidia's GPUs include tensor cores—special-purpose processing units for four-by-four matrix-multiply-and-accumulate. Usually, they are employed to accelerate deep-learning processes, but recent studies have attempted to utilize the tensor core for FFT computations [20–23]. Sorna and others [23] divided an input vector into multiple vectors such that the length of each vector was equal to four. Four-by-four matrices were constructed, and FFT was executed with tensor cores. They also decreased the precision error associated with the four-by-four matrix-based FFT by combining two half-precision numbers into a single-precision number. Although they succeeded in using a tensor core for FFT computations, their performance was lower than that achieved using the cuFFT library. Durrani and others [20], Li and others [21], and Pisha and Ligowski [22] presented tensor-core-based FFT methods and achieved improved performance compared with those achieved using the cuFFT library. However, these approaches have several limitations on the precision [21] or the supported vector size [20, 22]. Therefore, using standard GPU cores is a more general solution.

Ogata and others [24] proposed a hybrid approach using both a CPU and GPU for 2D-FFT computations. They decomposed the 2D-FFT computation into a set of 1D-FFTs and distributed the FFT problems to the CPU and GPU. Additionally, to decrease the overhead for data transfer between the host and device memories, they performed matrix transposition before sending data. Chen and Li [10] extended the approach of Gu and others [16] and used both a GPU and CPU for FFT computations, similar to Ogata and others [24]. Unlike Gu and others [16], they used a 2D data-copy application programming interface (API) instead of gathering multiple subarrays before sending them to transfer multidimensional data.

Similar to Ogata and others [24], we also employed 1D-FFT-based, high-dimensional FFT computations and used the data-transposition approach to achieve efficient data communication between the CPU and GPU. However, we handled 3D-FFT problems instead of 2D-FFT. In addition, we propose a data-transposition method for 3D data that depends on the target dimensions.

The data communication medium between the system and device memories is the peripheral component interconnect express (PCIe). Most prior studies used pinned (or page-locked) memory regions to maximize the bandwidth of PCIe. However, only part of the system can be a page-locked region, which limits the problem size that previous algorithms could treat. Lee and others [25] proposed a pinned-memory buffer approach to handle large-scale data that exceeded the maximum size of the page-locked memory. Instead of managing the entire input data in the pinned-memory space, they constructed small-sized pinned-memory buffers were constructed. Subsequently, only the target work unit (e.g., rows or columns) was placed in the pinned buffer for data communication between the CPU and GPU. This pinned-memory buffer approach requires additional data transposition and data-copy overheads. The authors of this study found that the benefit of using pinned-memory buffers was considerable compared with that associated with overhead. As a result, they successfully handled a large-scale 2D-FFT problem, which could not be handled by the prior approach, and achieved improved performance (up to 3.24 times) than the conventional implementation (which did not use pinned-memory buffers).

We employed the pinned buffer concept of Lee and others [25] and adapted it to 3D-FFT problem. We also present a novel data rearrangement (transposition) method for the 3D-FFT case to efficiently utilize the pinned buffer.

Distributed computing environments are suitable candidates for handling large-scale FFT problems [26, 27]. Excellent analyses based on recent studies are available [28, 29]. The primary issue in a distributed system is the data commutation overhead on distributed memory. Cayrols and others [30] proposed a data compression approach to reduce the communication overhead. However, there is a trade-off between the accuracy and computing performance for 3D-FFT. Unlike distributed FFT algorithms, we focused on optimizing the processing speed of 3D-FFT on a GPU node.

3 | PRELIMINARIES

In this section, we first define a 3D FFT operation. We then introduce GPU-based computational characteristics that are required to understand the proposed method.

3.1 | 3D-FFT

3D-FFT is represented by (1), where $F(f_x, f_y, f_z)$ is the distribution of the Fourier domain and $f(x, y, z)$ is the 3D input data.

$$F(f_x, f_y, f_z) = \sum_{x=0}^{N_x-1}\sum_{y=0}^{N_y-1}\sum_{z=0}^{N_z-1} f(x,y,z)\exp\left(-2\pi i\left(\frac{xf_x}{N_x}+\frac{yf_y}{N_y}+\frac{zf_z}{N_z}\right)\right). \quad (1)$$

x,y, and z are indices of the input signal, and f_x, f_y, and f_z are indices of the Fourier domain. N_x, N_y, and N_z are the sizes of each dimension of the signal, and i is $\sqrt{-1}$. In (1), $\exp(-2\pi i(\frac{xf_x}{N_x}+\frac{yf_y}{N_y}+\frac{zf_z}{N_z}))$ is the twiddle factor, which can be decomposed in each dimension. For example, the twiddle factor of the X-coordinate is $\exp(xf_x/N_x)$. Based on this property, we can represent the 3D-FFT as a set of 1D-FFT for each dimension.

As the 1D-FFT-based, high-dimensional FFT can handle each dimension separately, it requires considerably less memory than handling all data simultaneously. Therefore, this approach facilitates the use of systems with limited memory space. In this study, we employed 1D-FFT to perform 3D-FFT computations to efficiently utilize GPUs that have limited device memory.

3.2 | Characteristics of GPU computation

To utilize GPU for computations, the input data must first be copied to the GPU memory (i.e., device memory). Typically, the PCIe communication interface is used, and the bandwidth of PCIe (e.g., 31.75 GB/s) is lower than the bandwidths of the system memory (e.g., 95.37 GB/s) and device memory (e.g., 936.2 GB/s). Therefore, frequent communication between the host and device is a performance bottleneck in a GPU algorithm. Additionally, to maximally utilize the PCIe bandwidth, a pinned-memory (i.e., page-locked) region must be used, which is guaranteed to not be swapped-out by the operating system.

However, we could only set a part of the system memory as a pinned-memory region, and it is difficult to place the entire large-scale data into the pinned memory.

In our method, we constructed fixed and small-sized buffers in the pinned-memory region to handle a large-scale 3D-FFT problem with GPU efficiently (Section 4.2).

GPU and CPU are independent processing units that can run asynchronously. In addition, data copies between two devices can be performed during the time the devices perform computations. Such concurrent execution can be realized by pipelining computation and data communication through multiple streams, which is a medium for issuing tasks to the GPU. The concurrent execution of computation and data transfer is a well-known strategy for hiding data communication overhead. We employ this approach in our method.

4 | GPU-BASED 3D-FFT ALGORITHM

4.1 | Algorithm overview

To handle a large-scale 3D-FFT problem with limited device memory space, we employed the 1D-FFT-based 3D-FFT calculation approach (Section 3.1). This implies that our method performs 1D-FFTs for each dimension. Note that the processing order of the dimensional data did not affect the final 3D-FFT result. The basic work unit of 1D-FFT is a line along a specific dimension (such as a matrix row or column). Figure 1 shows an overview of the proposed GPU-based 3D-FFT algorithm that processes each 1D vector (i.e., line) in five steps.

The first step is data transposition (i.e., *Transpose*). The Y- and Z-dimensional data are generally noncontinuous in the system memory, according to (2).

$$\text{Index}(x, y, z) = zN_xN_y + yN_x + x. \quad (2)$$

As nonsequential data access is slower than sequential access, 1D-FFT computations for the Y and Z dimensions become a performance bottleneck. For the Y and Z dimensions, we transposed the 1D vectors in a continuous memory form to improve the processing efficiency. The transposed data were then placed into the pinned buffer, and a small pinned-memory space was used to send the data to the GPU (Section 4.2). As X-dimensional data are continuously placed in the

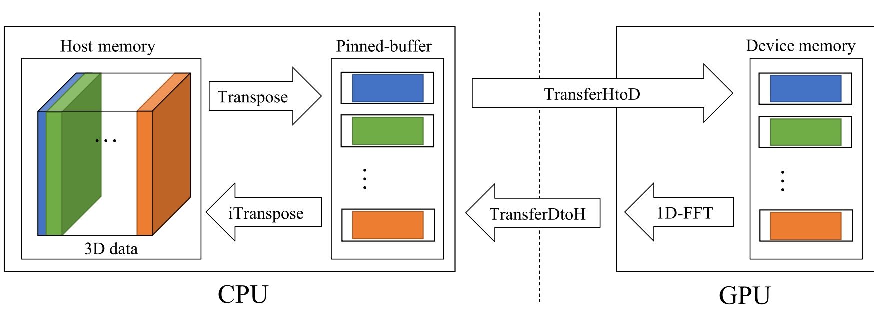

FIGURE 1 Process overview of the proposed 3D-FFT algorithm.

memory, the pinned buffer is copied, thus bypassing the transposition step.

The second step is data copying from the host and device memories (i.e., *TransferHtoD*). This step transfers the data in the pinned buffer to the GPU's device memory. Once the data are transferred to the device memory, GPU performs 1D-FFT operations (i.e., the *1D-FFT* step). After the 1D-FFT computation is completed, the results are transferred from the device memory to the pinned buffer on the host memory, which is the fourth step, commonly referred to as *TransferDtoH*. The last step is inverse transposition (i.e., *iTranspose*). This step transposes the result vector in the pinned buffer in the reverse manner of the first step and places the results in the original memory space of the data.

Basic work unit: As the data size increases, the data transfer efficiency improves. Additionally, to utilize the GPU's massive parallelism capability to the maximum extent, many data need to be processed simultaneously [10, 11]. To achieve a high efficiency in data transfer and computation, we constructed a set of 1D datasets (rows or columns) in the form of a data part that constituted the basic work unit of our algorithm. We also handled multiple parts (e.g., four parts) simultaneously, using different streams. Given that tasks in different streams can overlap (e.g., data transfer and computation on GPU), we can hide the data communication overhead while improving GPU's utilization.

4.2 | Data transposition and generation of data parts

Figure 2 shows the memory layout of 3D data. Unlike the X-dimensional data allocated continuously, the Y- and Z-dimensional data were placed discontinuously and strided. This discontinuous data layout leads to nonsequential memory access and lowers the cache utilization efficiency (e.g., hit ratio) while requiring more data access latency than sequential memory access. Additionally, continuous data can be handled with a single data-copy API for communication between the CPU and GPU. However, data with a discontinuous layout require multiple API calls, thereby decreasing the data transfer efficiency (e.g., the time required per data element).

To improve the data transfer efficiency, we gathered data of the target dimension into a contiguous memory space (i.e., pinned buffer) by data transposition. For a Y-dimensional vector, the distance in memory between two neighboring elements is the length of the X dimension (i.e., N_x). Furthermore, there is a finite space equal to $N_x \times N_y$ between two neighboring elements in the case of a Z-dimensional vector. Equations (3) and (4) are the equations for calculating the transposed positions of

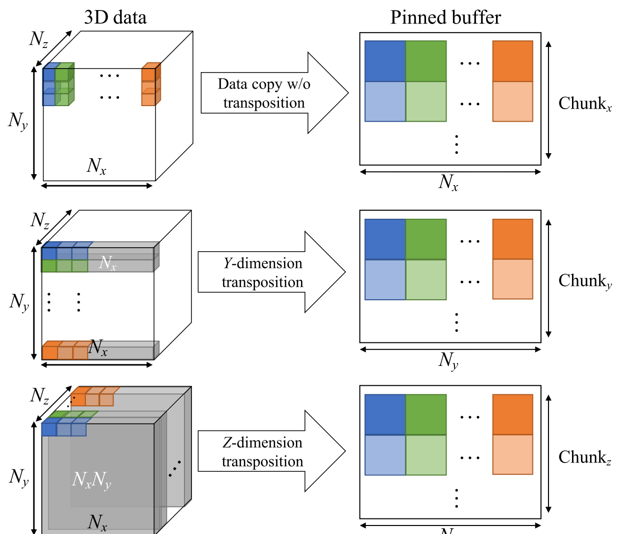

FIGURE 2 The shape of 3D data in memory and data-transposition example for each dimension.

element (x, y, z) along the Y and Z dimensions, respectively, where (x', y', z') is the index of the first element of the data part of the 3D data.

$$\mathrm{Trans}_y(x,y,z) = (z-z')N_xN_y + (x-x')N_y + y, \quad (3)$$

$$\mathrm{Trans}_z(x,y,z) = (y-y')N_xN_z + (x-x')N_z + z. \quad (4)$$

The target data part was copied into the pinned buffer and rearranged using transposition equations. These were then transferred to the device memory for 1D-FFT computations. Moreover, after 1D-FFT computations, GPU returned the result to the pinned buffer. Note that the data communication between the pinned buffer and device memory required only a data-copy API call, as we placed them in a continuous memory space.

We need to write the 1D-FFT results must be written to the original data region for subsequent 1D-FFT processes (e.g., 1D-FFTs for other dimensions). This requires another data-transposition step in the inverse direction with respect to the direction of the first transposition. Equations (5) and (6) present the inverse transposition equations for the Y and Z dimensions, respectively.

$$\mathrm{iTrans}_y(x_b, y_b, z_b) = (z_b + z')N_xN_y + x_bN_x + (y_b + x'), \quad (5)$$

$$\mathrm{iTrans}_z(x_b, y_b, z_b) = x_bN_xN_y + (z_b + y')N_x + (y_b + x'). \quad (6)$$

In (5) and (6), (x_b, y_b, z_b) is the data index in the pinned buffer. Although our approach requires additional transposition steps before and after processing each part, we found that the advantage of data transposition on data

communication efficiency outweighs the overhead. For example, for a vector with a size of 2048 in the Z dimension, the time required to copy the vector with two data-transposition steps (time of transpositions and data transfer) is approximately 56% of the time required to send data without transposition.

4.3 | Optimization with multiple streams

The CPU and GPU are independent computing units, as explained in Section 3.2. It is possible to run multiple operations (e.g., computations in each computing unit and data communication) in different streams asynchronously and concurrently.

To minimize the data transfer overhead and maximize GPU's utilization efficiency, we constructed multiple streams and allocated data parts evenly to the streams in a round-robin manner. We also allocate a dedicated pinned buffer to each stream. As each stream handles different data parts from each other, they can process the given part independently while exploiting the concurrent execution capability of the system. Prior studies have indicated that the use of four streams yielded good performance [31, 32]. In addition, we found that four streams worked best for our algorithm. Therefore, we designed our algorithm to use four streams.

Data-part size: The data transfer between CPU and GPU memories is capable of achieving increased bandwidth as the amount of data sent in one transmission is increased [10, 11]. Therefore, increasing the data-part size can improve the performance of the proposed algorithm. To maximize the chunk size, we determine the chunk size based on (7), where C_d is the chunk size (the number of lines) in d dimensions, V_d is the size of the 1D vector for 1D-FFT, $|\text{Device memory}|$ is the available GPU memory size, N_{stream} is the number of streams, and k (>1) is a coefficient for holding the memory space for 1D-FFT computations.

$$C_d \leq \frac{|\text{Device memory}|}{k V_d N_{\text{stream}}}. \tag{7}$$

We set k to 3 because we needed space for the input data, output data, and temporal space for the computation.

4.4 | Memory footprint

Our FFT algorithm did not require additional memory space for the output, as GPU handled the 1D-FFT computation. The results were applied directly to the pre-allocated space of the original (or input) data. Therefore, we only needed additional space for the pinned buffers on the host side. The size for pinned buffers was $C_d V_d N_{\text{stream}}$, which was bounded by the device memory size (e.g., $|\text{Device memory}|/k$). Therefore, the memory footprint of the host (Mem_{host}) is defined by (8), where $|\text{Data}|$ denotes the size of the input data.

$$\begin{aligned} \text{Mem}_{\text{host}} &= |\text{Data}| + C_d V_d N_{\text{stream}} \\ &= O(|\text{Data}| + |\text{Device memory}|/k). \end{aligned} \tag{8}$$

Our method uses as much device memory as possible to improve data communication and computational efficiencies, as mentioned in Section 4.3. Therefore, the memory footprint of the device is bounded by the device memory size for large-scale FFT problems.

5 | RESULTS AND ANALYSIS

We implemented our 3D-FFT algorithm on a system equipped with an Nvidia RTX 3090, with a device memory of 24 GB. We employed CUDA 11.6 and cuFFT libraries to implement GPU algorithms. To verify the benefits of our method compared with prior approaches, we implemented two alternative methods based on state-of-the-art algorithms.

- *FFTW* is a state-of-the-art FFT library which uses a multicore CPU. We used the ESTIMATE plan option and set the number of threads to 32 (equal to the number of physical cores in the CPU).
- *Chen* is the implementation of Chen and Li [10] of a state-of-the-art GPU-based FFT algorithm supporting 3D-FFT. For this implementation, we used cuFFT and FFTW for the GPU and CPU modules, respectively. To implement 3D-FFT, we divided the Z dimension into the $Z1$ and $Z2$ segments, the Y dimension into the $Y1$ and $Y2$ segments, and computed the 5D-FFT of $Z1 \times Z2 \times Y1 \times Y2 \times X$. We set the $Z2$ and $Y2$ to 32 because this setting yields the best performance based on our benchmark. *Chen* performed 3D-FFT in two phases; the first phase was for $Z1 \times Y1 \times Y2$ and the second for $Z2 \times X$. Therefore, it needed two cuFFT plans for each phase and required more device memory space than *Ours*. Their publication reported that the best workload distribution between CPU and GPU was 2:8 in their test. However, we found that only GPU achieved the best performance in our system in all the cases. Additionally, we found that the best number of streams depended on the input data size unlike

the reported value (i.e., eight streams) in their publication. We tested various numbers of streams to compare the outcomes with those obtained with our method. For each input data, we used the best one for the result of *Chen*.

Benchmarks: We generated a set of 3D data with random complex numbers to measure the performance of three algorithms (*FFTW*, *Chen*, and *Ours*). The data sizes were varied from $2^{10} \times 2^{10} \times 2^{11}$ to $2^{12} \times 2^{12} \times 2^{10}$ in the single-precision benchmark and from $2^{9} \times 2^{10} \times 2^{11}$ to $2^{11} \times 2^{11} \times 2^{11}$ in the double-precision benchmark. Tables 1 and 2 show information about the benchmarks used in our experiments.

5.1 | Results

Tables 1 and 2 show the 3D-FFT computation times using the algorithms *FFTW*, *Chen*, and *Ours* for each benchmark. For single-precision benchmarks, *Ours* achieved improved performances (up to 3.11 times [2.73 times on average]) than *FFTW* (Figure 3). We found that the performance gap between *FFTW* and *Ours* increased as the data size increased. For example, *Ours* yielded improvement performances equal to 3.11 times and 2.25 times than *FFTW* for the $2^{11} \times 2^{12} \times 2^{11}$ and $2^{10} \times 2^{10} \times 2^{11}$ cases, respectively. For data sizes equal to or less than 64 GB (2^{33}), *Chen* achieved the best performance among the three algorithms. However, the performance gap between *Chen* and *Ours* decreased as the data size increased. For the largest benchmarks (2^{34}, 128 GB), *Chen*

TABLE 1 This table shows three algorithms' processing times (seconds) for the benchmarks in single-precision complex numbers.

Data shape ($2^X \times 2^Y \times 2^Z$)	Single precision			
	Data size (GB)	FFTW	Chen	Ours
(10, 10, 11)	16	31.80	7.30	14.16
(10, 11, 10)	16	31.47	7.30	13.47
(11, 10, 10)	16	31.99	7.20	14.05
(10, 11, 11)	32	74.03	14.40	26.94
(11, 10, 11)	32	66.03	14.08	28.04
(11, 11, 10)	32	63.32	14.05	26.14
(10, 11, 12)	64	131.47	33.63	54.67
(10, 12, 11)	64	150.98	33.16	51.54
(11, 10, 12)	64	133.95	33.27	57.22
(11, 11, 11)	64	148.67	32.62	51.45
(11, 12, 10)	64	133.29	32.89	43.67
(12, 10, 11)	64	159.74	32.44	56.91
(12, 11, 10)	64	139.32	32.51	46.05
(10, 12, 12)	128	318.48	609.47	105.70
(11, 11, 12)	128	320.98	647.60	108.63
(11, 12, 11)	128	269.66	630.57	86.76
(12, 10, 12)	128	333.32	680.20	114.35
(12, 11, 11)	128	278.78	613.87	91.18
(12, 12, 10)	128	287.72	576.50	96.79

TABLE 2 This table shows three algorithms' processing times (seconds) for the benchmarks in double-precision complex numbers.

Data shape ($2^X \times 2^Y \times 2^Z$)	Double precision			
	Data size (GB)	FFTW	Chen	Ours
(9, 10, 11)	16	17.80	2.40	9.30
(9, 11, 10)	16	16.33	2.46	9.18
(10, 9, 11)	16	15.21	2.17	9.39
(10, 10, 10)	16	14.06	2.20	9.19
(10, 11, 9)	16	16.88	2.25	8.92
(11, 9, 10)	16	15.13	2.15	9.63
(11, 10, 9)	16	17.63	2.19	9.19
(9, 11, 11)	32	33.49	15.33	18.34
(10, 10, 11)	32	36.79	14.85	18.44
(10, 11, 10)	32	34.63	14.76	17.69
(11, 9, 11)	32	32.90	14.49	19.31
(11, 10, 10)	32	30.58	14.56	18.30
(11, 11, 9)	32	35.38	16.24	16.40
(10, 11, 11)	64	68.75	35.07	35.53
(11, 10, 11)	64	72.70	34.56	36.75
(11, 11, 10)	64	69.64	34.50	32.54
(11, 11, 11)	128	134.06	576.17	65.63

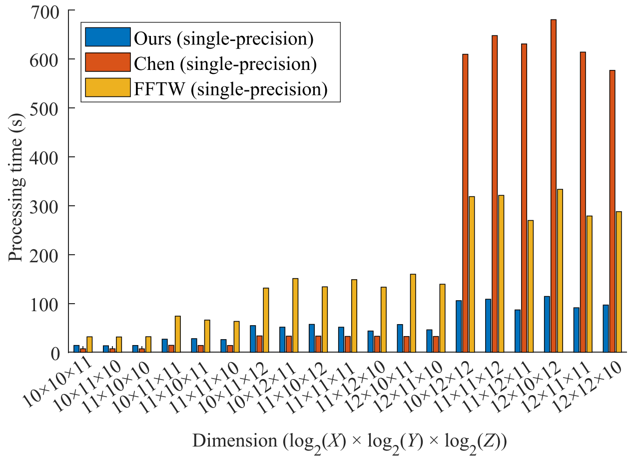

FIGURE 3 Processing time comparison among three algorithms for benchmarks in single-precision complex numbers.

yielded significantly lower performance (e.g., 7.27 times) than *Ours* and even lower than *FFTW*. Because *Chen* is an out-of-place algorithm that requires a memory space that is more than twice the input data size, the required memory space exceeds the system memory size (i.e., 256 GB) of the testing system. The high memory usage of *Chen* leads to significant performance degradation. Unlike *Chen*, *Ours* handled large-scale data robustly, because it is a near-in-place algorithm. We discuss this memory-usage issue in Section 5.3

In the double-precision case, *Ours* achieved improved performance up to 2.16 times (1.86 times on average) than that of *FFTW* (Figure 4). Additionally, as in the single-precision case, *Ours* yielded a higher performance improvement than *FFTW* as the data size increased. However, the overall performance improvement compared to *FFTW* was less than that in the single-precision case. This is because the performance difference between the CPU and GPU is larger for single-precision than for double-precision computations. Moreover, the overhead for data transposition and communication in the single-precision case was lower than that in the double-precision case, as the data size was smaller than that of the double-precision data. Nonetheless, we found that *Ours* achieved better performance than *FFTW* for all benchmarks at both precision levels. *Chen* also shows a performance trend similar to that of single-precision cases. However, *Ours* achieved a performance comparable to that of *Chen* for large-scale data (e.g., 2^{31}–2^{32}, 32 GB–64 GB). Specifically, for the largest benchmarks (2^{33}, 128 GB), *Chen* yielded significantly lower (i.e., 8.78 times) performance improvements than *Ours*, owing to the huge memory usage of *Chen*.

These results confirm that our GPU-based 3D-FFT algorithm is appropriate for handling large-scale 3D-FFT problems.

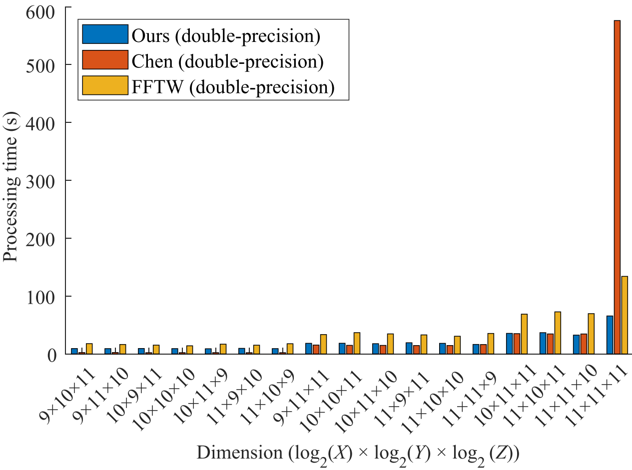

FIGURE 4 Processing time comparison among three algorithms for benchmarks in double-precision complex numbers.

5.2 | Performance analysis

Our method handles each data part in five steps (Section 4.1). We outline an in-depth analysis of the performance of our 3D-FFT algorithm, and Figure 5 is a stacked bar chart that shows the processing time for each step. For this profiling, we used $2^{11} \times 2^{11} \times 2^{11}$ data in single-precision format. Please note that to measure the time required for each step, we implemented synchronization at the beginning and end of each task. However, in an actual implementation, these steps would be executed concurrently without synchronization (Section 4.3).

Among the five steps, data transposition (*Transpose*) and inverse transposition (*iTranspose*) are the most time consuming. *Transpose* and *iTranspose* require approximately 45% and 29% of the total workload, respectively (i.e., the total summation of the processing time for each step). Our method employed 1D-FFT-based 3D-FFT computations, and the transposition methods differed according to the target dimension (Section 4.2). Consequently, different times were required for *Transpose* and *iTranspose* depending on the target dimension. For the benchmark $2^{11} \times 2^{11} \times 2^{11}$ in the single-precision case (Figure 5), *Transpose* required 6.1 and 19.9 s for Y and Z dimensions. Because the X dimension did not require data transposition, it only required time to copy the data to the pinned buffer, which was equal to 2.9 s. This means that the data-transposition (*Transpose*) overhead for the Y and Z dimensions are approximately 6.7 and 2.1 times higher than those required to copy the data (e.g., for the X dimension). Although data transposition requires this overhead, we found that the summation of

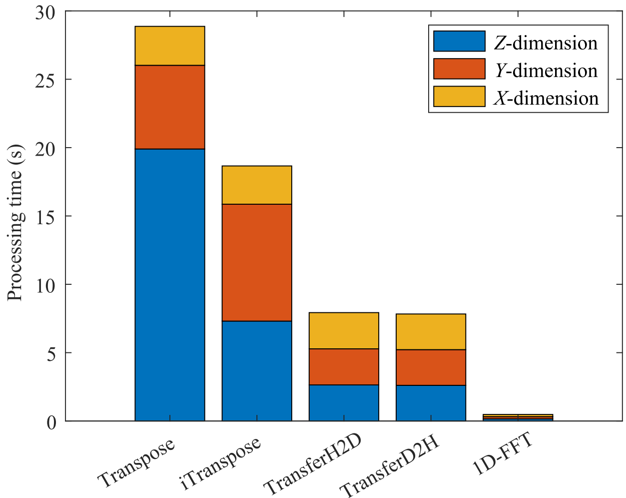

FIGURE 5 This stacked bar chart shows the total processing time for each step. We used $2^{11} \times 2^{11} \times 2^{11}$ data in single precision for this profiling. Each bar stacks the processing time for each dimension.

the time for the two transposition steps and CPU–GPU data communications required considerably less time (e.g., 56%) than the time required for data transfer between the CPU and GPU without transposition.

The transposition for the Z dimension required 3.3 times more computation time than that for the Y dimension. The elements along the Z dimension have greater memory distances with neighboring elements than the Y-dimensional vector. The longer the memory distance, the more disadvantageous the cache utilization efficiency (e.g., hit ratio). To check the cache efficiency in the transposition step for each dimension, we measured the last level cache (LLC) misses using Intel's Vtune analyzer. Because Vtune does not work with AMD's CPU in our testing system, we used another system equipped with an Intel CPU (i.e., Intel i7-9700k) for this analysis. For this analysis, we used $2^{11} \times 2^{11} \times 2^{11}$ data in single precision, and the numbers of LLC misses for the Z, Y, and X dimensions were 461 M, 395 M, and 1.9 M, respectively. This result explains why the transposition for the Z dimension required more time than other dimensions.

Conversely, *iTranspose* along the Y and Z dimensions yielded a similar performance. As the data in the pinned buffer are transposed data, continuous data are read efficiently, unlike the *Transpose* case, wherein each element in the original 3D data is read across the dimension. Although the *Transpose's* writing operation to the pinned buffer accesses the sequential region, the preceding work (i.e., reading from the original data) delays it because it is a cache-inefficient memory operation. Therefore, we found that *iTranspose* requires less time than *Transpose* for both the Y and Z dimensions.

With the exception of the two transposition steps, data communication between the CPU and GPU accounted for the highest part of the overall processing time. In Figure 5, the data transfer from CPU to GPU (i.e., *TransferH2D*) required 7.9 s, and data copy from GPU to CPU (i.e., *TransferD2H*) required 7.8 s. These values are approximately 12% of the total workload. The unit of data transfer is the data part of the same size, and the only difference between *TransferH2D* and *TransferD2H* is the transfer direction. This implies that the workload was almost the same as the profiling result. For $2^{11} \times 2^{11} \times 2^{11}$ single-precision data, 64 GB of data was transferred along each dimension. In Figure 5, the data transfer time of a dimension is approximately 2.6 s, and its transfer speed is 24.6 GB/s. The system used in our experiment had PCIe4.0x16, and its theoretical maximum bandwidth was 31.51 GB/s. Therefore, our algorithm achieved 78% of the theoretical maximum bandwidth. We can achieve this high utilization rate of the communication interface bandwidth because the algorithm maximizes the data-part size while considering the size of the device's memory and the number of streams.

The 1D-FFT steps required the smallest number of steps (approximately 3% (1.5 s) of the total workload). This is because the 1D-FFT computation fit the GPU's parallel architecture, and the GPU yielded a much higher performance than the CPU for these data parallelism tasks.

As stated above, the transpose step requires more time than the total time required by the data transfer and 1D-FFT steps. At runtime, in cases in which we used multiple streams, the time for data transfer and GPU computation was hidden by the times for data transposition. Consequently, we can see that the actual total processing time (e.g., 51.45 s) is almost the same as the summation of the processing times for *Transpose* and *iTranspose* (e.g., 47.53 s).

5.3 | Memory usage

The FFTW library supports both in-place and out-of-place computations. The in-place algorithm does not require additional memory space, and the memory usage of *FFTW* in in-place mode is almost the same as the input data size (i.e., $|\text{Data}|$). *Chen* is an out-of-place algorithm that requires more than $2 \times |\text{Data}|$ because it requires memory space for output. Therefore, *Chen* requires memory space in excess of 256 GB for the largest double-precision benchmark. This is larger than the test system's physical memory space (256 GB). The virtual memories of operating systems make it possible to run a program that requires larger space than the physical memory size. However, using virtual memory spaces can degrade the processing performance considerably because the virtual memory method uses a part of the swap device (or disk), which is much slower than physical memory [33]. This is the reason for Chen's abnormally low performance on large-scale data.

Conversely, as mentioned in Section 4.4, the memory footprint of Ours is $O(|\text{Data}| + |\text{Device memory}|/k)$. Because we set k to three, our method used approximately 136 GB (i.e., $128 + 8$ GB) of system memory for the largest benchmark (i.e., $2^{11} \times 2^{11} \times 2^{11}$) in a double-precision format. It can be inferred that *Ours* is a near-in-place algorithm. Therefore, *Ours* yielded a more stable performance in terms of data-size increments than *Chen*. These results demonstrate the robustness of the proposed approach for large-scale 3D-FFT computations.

6 | CONCLUSION AND FUTURE WORK

We presented a novel GPU-based 3D-FFT algorithm for large-scale 3D data whose sizes were larger than the GPU's device memory. Our method employed a 1D-FFT-

based, high-dimensional FFT approach to handle large-scale data using limited device memory. To improve the data communication efficiency between the CPU and GPU memories, we proposed a 3D transposition method to place the 1D vector of the target dimension in a contiguous memory region. We also employed the pinned buffer instead of placing all the data in the pinned memory, as only a part of the system memory can serve as page-locked memory, thus limiting the maximum data size that can be handled. To optimize the performance of our algorithm, we utilized multiple streams that were associated with overlapped computations on GPU and data transfer between the host and device memories. Additionally, we maximized the part (i.e., the work unit of our method) by considering the size of the device memory and the number of streams to achieve the maximum utilization of the bandwidth of the communication interface (i.e., PCIe).

We implemented our method on a GPU system and compared its performance with an alternative implementation using the state-of-the-art FFT library for large-scale 3D-FFT problems. As a result, our GPU-based algorithm achieved improved performance by up to 2.89 times compared with those of alternative implementations (i.e., *FFTW*). Furthermore, our algorithm exhibited an improved performance for all benchmarks. These results demonstrate the advantages of the proposed approach for large-scale 3D-FFT problems.

Utilizing GPU for FFT computations significantly reduced the processing time of 1D-FFT. Consequently, in the proposed method, additional data-transposition tasks that arise when a GPU is employed become a performance bottleneck. In future work, we would like to design an efficient data-transposition algorithm for large-scale 3D data. In addition, we would like to extend our algorithm to a heterogeneous parallel 3D-FFT algorithm utilizing all available computing resources, including multicore CPUs and different types of GPUs, as in Kim and others [34]. This work assumes that the system memory is adequate to hold the input data. However, this may not be valid for extreme-scale problems. As another future research direction, we would like to design an efficient out-of-core FFT algorithm that can handle massive-scale data sizes exceeding the system memory size.

CONFLICT OF INTEREST STATEMENT

The authors declare that there are no conflicts of interest.

ORCID

Jaehong Lee https://orcid.org/0000-0002-8311-5975
Duksu Kim https://orcid.org/0000-0002-9075-3983

REFERENCES

1. R. N. Bracewell and R. N. Bracewell, *The Fourier transform and its applications*, Vol. **31999**, McGraw-Hill, New York, 1986.
2. J. W. Cooley and J. W. Tukey, *An algorithm for the machine calculation of complex Fourier series*, Math. Comput. **19** (1965), no. 90, 297–301. https://doi.org/10.1090/s0025-5718-1965-0178586-1
3. M. Frigo and S. G. Johnson, *The design and implementation of FFTW3*, Proc. IEEE **93** (2005), no. 2, 216–231.
4. M. Frigo and S. G. Johnson, *FFTW: an adaptive software architecture for the FFT*, (Proc. 1998 IEEE Int. Conf. Acoust. Speech Signal Process. Seattle, WA, USA), 1998, pp. 1381–1384.
5. Intel, *Intel® Math Kernel Library*, 2020. https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html
6. K. Matsushima, *Introduction to computer holography: creating computer-generated holograms as the ultimate 3D image*, Springer Nature, London, UK, 2020.
7. D. Takahashi, *Fast Fourier transform algorithms for parallel computers*, Springer, Berlin, Heidelberg, Germany, 2019.
8. MathWorks, Matlab, Web Page, https://mathworks.com/, 2022.
9. NVIDIA, *CUFFT libraries*. https://docs.nvidia.com/cuda/cufft/index.html
10. S. Chen and X. Li, *A hybrid GPU/CPU FFT library for large FFT problems*, (IEEE 32nd Int. Perform. Comput. Commun. Conf. (IPCCC), San Diego, CA, USA), 2013, pp. 1–10.
11. T. M. John Cheng, *Professional CUDA C programming*, John Wiley & Sons, Hoboken, New Jersey, United States, 2014.
12. A. Kumar, S. Gavel, and A. S. Raghuvanshi, *FPGA implementation of radix-4-based two-dimensional FFT with and without pipelining using efficient data reordering scheme*, *Nanoelectronics, circuits and communication systems*, Springer, Republic of Singapore, 2021, pp. 613–623.
13. N. H. Nguyen, S. A. Khan, C.-H. Kim, and J.-M. Kim, *A high-performance, resource-efficient, reconfigurable parallel-pipelined FFT processor for FPGA platforms*, Microprocess. Microsyst. **60** (2018), 96–106.
14. S. Khokhriakov, R. R. Manumachu, and A. Lastovetsky, *Performance optimization of multithreaded 2D fast Fourier transform on multicore processors using load imbalancing parallel computing method*, IEEE Access **6** (2018), 64202–64224.
15. L. Gu, X. Li, and J. Siegel, *An empirically tuned 2D and 3D FFT library on CUDA GPU*, (Proc. 24th ACM Int. Conf. Supercomput., Association for Computing Machinery, New York, NY, USA), 2010, pp. 305–314.
16. L. Gu, J. Siegel, and X. Li, *Using GPUs to compute large out-of-card FFTs*, (Proc. Int. Conf. Supercomput., Association for Computing Machinery, New York, NY, USA), 2011. https://doi.org/10.1145/1995896.1995937
17. Z. Zhao and Y. Zhao, *The optimization of FFT algorithm based with parallel computing on GPU*, (Proc. IEEE 3rd Adv. Inf. Technol. Electron. Autom. Control Conf. (IAEAC), IEEE, Chongqing, China), 2018, pp. 2003–2007.
18. Advanced Micro Devices, *rocFFT*. https://rocfft.readthedocs.io/en/rocm-5.3.1/, Accessed: 2022-11-04.

19. Y. Hu, L. Lu, and C. Li, *Memory-accelerated parallel method for multidimensional fast Fourier implementation on GPU*, J. Supercomput. **78** (2022), no. 16, 18189–18208.
20. S. Durrani, M. S. Chughtai, M. Hidayetoglu, R. Tahir, A. Dakkak, L. Rauchwerger, F. Zaffar, and W. M. Hwu, *Accelerating Fourier and number theoretic transforms using tensor cores and warp shuffles*, (30th Int. Conf. Parallel Archit. Compilation Tech. (PACT), Atlanta, GA, USA), 2021. https://doi.org/10.1109/PACT52795.2021.00032
21. B. Li, S. Cheng, and J. Lin, *tcFFT: a fast half-precision FFT library for NVIDIA Tensor Cores*, (IEEE Int. Conf. Cluster Comput. (CLUSTER), Portland, OR, USA), 2021. https://doi.org/10.1109/Cluster48925.2021.00035
22. L. Pisha and L. Ligowski, *Accelerating non-power-of-2 size Fourier transforms with GPU tensor cores*, (IEEE Int. Parallel Distrib. Process. Symp. (IPDPS), Portland, OR, USA), 2021. https://doi.org/10.1109/IPDPS49936.2021.00059
23. A. Sorna, X. Cheng, E. D. Azevedo, K. Won, and S. Tomov, *Optimizing the fast Fourier transform using mixed precision on tensor core hardware*, (IEEE 25th Int. Conf. High Perform. Comput. Workshops (HiPCW), Bengaluru, India), 2018, pp. 3–7.
24. Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka, *An efficient, model-based CPU-GPU heterogeneous FFT library*, (IEEE Int. Symp. Parallel Distrib. Process., Miami, FL, USA), 2008. https://doi.org/10.1109/IPDPS.2008.4536163
25. J. Lee, H. Kang, H. J. Yeom, S. Cheon, J. Park, and D. Kim, *Out-of-core GPU 2D-shift-FFT algorithm for ultra-high-resolution hologram generation*, Opt. Express **29** (2021), no. 12, 19094–19112.
26. A. Gholami, J. Hill, D. Malhotra, and G. Biros, *AccFFT: a library for distributed-memory FFT on CPU and GPU architectures*, arXive preprint, 2015. https://doi.org/10.48550/arXiv.1506.07933
27. D. Sharp, M. Stoyanov, S. Tomov, and J. Dongarra, *A more portable HeFFTe: implementing a fallback algorithm for scalable Fourier transforms*, (IEEE High Perform. Extreme Comput. Conf. (HPEC), Waltham, MA, USA), 2021, pp. 1–5.
28. A. Ayala, S. Tomov, M. Stoyanov, A. Haidar, and J. Dongarra, *Performance analysis of parallel FFT on large multi-GPU systems*, (IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW), Lyon, France), 2022. https://doi.org/10.1109/IPDPSW55747.2022.00072
29. A. Ayala, S. Tomov, M. Stoyanov, and J. Dongarra, *Scalability issues in FFT computation*, *Parallel computing technologies*, Lecture Notes in Computer Science, Springer, Cham, Switzerland, 2021, pp. 279–287.
30. S. Cayrols, J. Li, G. Bosilca, S. Tomov, A. Ayala, and J. Dongarra, *Lossy all-to-all exchange for accelerating parallel 3-D FFTs on hybrid architectures with GPUs*, (IEEE Int. Conf. Cluster Comput. (CLUSTER), Heidelberg, Germany), 2022, pp. 152–160.
31. H. Kang, H. C. Kwon, and D. Kim, *HPMaX: heterogeneous parallel matrix multiplication using CPUs and GPUs*, Computing **102** (2020), no. 12, 2607–2631.
32. N. V. Sunitha, K. Raju, and N. N. Chiplunkar, *Performance improvement of CUDA applications by reducing CPU-GPU data transfer overhead*, (Int. Conf. Inventive Commun. Comput. Technol. (ICICCT), Coimbatore, India), 2017, pp. 211–215.
33. H. Kang, J. Lee, and D. Kim, *HI-FFT: heterogeneous parallel in-place algorithm for large-scale 2D-FFT*, IEEE Access **9** (2021), 120261–120273.
34. D. Kim, J. Lee, J. Lee, I. Shin, J. Kim, and S.-E. Yoon, *Scheduling in heterogeneous computing environments for proximity queries*, IEEE Trans. Visual. Comput. Graphics **19** (2013), no. 9, 1513–1525.

AUTHOR BIOGRAPHIES

Jaehong Lee received his BS degree in Computer Science and Engineering from the Korea University of Technology and Education (KOREATECH), Cheonan, Rep. of Korea, in 2020, and his MS degrees in Engineering from the KOREATECH, in 2022. His main research interests are high-performance computing and computer-generated holography.

Duksu Kim is currently an assistant professor in the School of Computer Engineering at Korea University of Technology and Education (KOREATECH), Cheonan, Rep. of Korea. He received his BS from SungKyunKwan University, Suwon, Rep. of Korea, in 2008. He received his PhD from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Rep. of Korea, in Computer Science in 2014. He spent several years as a senior researcher at KISTI National Supercomputing Center, Daejeon, Rep. of Korea. His research interest is designing high-performance algorithms for various applications, including computer graphics, robotics, and computer-generated holography.

How to cite this article: J. Lee and D. Kim, *Large-scale 3D fast Fourier transform computation on a GPU*, ETRI Journal **45** (2023), 1035–1045. DOI 10.4218/etrij.2022-0297