Received July 23, 2021, accepted August 24, 2021, date of publication August 27, 2021, date of current version September 7, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3108404

HI-FFT: Heterogeneous Parallel In-Place Algorithm for Large-Scale 2D-FFT

HOMIN KANG, JAEHONG LEE, AND DUKSU KIM, (Member, IEEE)
School of Computer Engineering, Korea University of Technology and Education (KOREATECH), Cheonan 31253, South Korea

Corresponding author: Duksu Kim (bluekdct@gmail.com)

This work was supported in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant through Korean Government [Ministry of Science and ICT (MSIT)] under Grant 2020-0-00594 (Morphable Haptic Controller for Manipulating VR·AR Contents, 50%), and in part 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, 50%).

ABSTRACT Fast Fourier Transform (FFT) is a fundamental operation for 2D data in various applications. To accelerate large-scale 2D-FFT computation, we propose a Heterogeneous parallel In-place 2D-FFT algorithm, HI-FFT. Our novel work decomposition method makes it possible to run our parallel algorithm on the original data (i.e., in-place), unlike prior parallel algorithms that require additional memory space (i.e., out-of-place) to guarantee independence among sub-tasks. Our work decomposition method also removes the duplicated operations on the out-of-place approaches. Using our decomposition method, we introduced an in-place heterogeneous parallel algorithm that utilizes both multi-core CPU and GPU simultaneously. To maximize the utilization efficiency of the computing resources, we also propose a priority-based dynamic scheduling method. We compared the performance of seven different 2D-FFT algorithms, including ours, for large-scale 2D-FFT problems whose sizes varied from $20K^2$ to $120K^2$. As a result, we found that our method achieved up to 2.92 and 4.42 times higher performance than the conventional homogeneous parallel algorithms based on the state-of-the-art CPU and GPU libraries, respectively. Also, our method showed up to 2.27 times higher performance than the prior heterogeneous algorithms while requiring two times less memory space. To check the benefit of our HI-FFT on an actual application, we applied it to a CGH (Computer Generated Holography) process. We found that it successfully reduces the hologram generation time. These results demonstrate the advantage of our approach for large-scale 2D-FFT computation.

INDEX TERMS 2D-FFT, heterogeneous, parallel, CPU, GPU, in-place.

I. INTRODUCTION

The Discrete Fourier Transform (DFT) is one of the fundamental operations in the scientific and engineering domains [1]. A Fast Fourier Transform (FFT) is an algorithm for computing DFT efficiently, and 2D-FFT is widely employed in various applications, including image processing, machine learning, and digital holography [2]–[4]. There have been numerous attempts to accelerate the performance of FFT. Early studies focused on minimizing the number of arithmetic operations in the FFT process. For example, the Cooly-Tukey algorithm reduces the computational complexity from $O(N^2)$ to $O(NlogN)$ [5].

One of the widely employed strategies for accelerating mathematic computation is utilizing parallel computing hardware, including FPGA, multi-core CPU [6]–[8], and GPU [9]–[11]. Recent FFT acceleration approaches have also actively employed parallel processing approaches. A state-of-the-art algorithm utilizing multi-core CPUs (e.g., FFTW [7] and MKL [8]) improved the FFT performance by several times compared with using a single CPU core. Also, the state-of-the-art GPU FFT library (i.e., cuFFT [11]) improved performance by several orders of magnitude thanks to its massive parallelism, which meets the compute-intensive nature of FFT computation. Despite the impressive performance of the GPU-based algorithm, the limited GPU memory size restricts the use of GPU for large-scale FFT problems such as ultra-high-resolution hologram (e.g., $100K^2$ double complex matrix) generation.

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

Recently, most computing systems include both multi-core CPU and GPU in a system. To fully exploit the computing capability of such heterogeneous systems, there have been attempts to utilize different types of parallel computing architectures at once [12]. Such heterogeneous parallel techniques achieved impressive performance improvement in various applications [13]–[17]. Hybrid CPU/GPU algorithms have also been proposed, and they solved the limited GPU memory issue by maintaining the all of data in the CPU memory and sending parts of them to the GPU memory [18]. They also utilized both multi-core CPU and GPU for the computation, to obtain further performance improvements [19], [20].

Our HI-FFT improves the prior heterogeneous parallel approaches for 2D-FFT computation in three aspects: memory usage, redundant operations, and utilization efficiency of the heterogeneous computing system. In the prior parallel 2D-FFT algorithms, a typical work distribution unit is a line (or row) of two-dimensional data (hereafter referred to as a 'matrix' for simplicity). However, a line-based work distribution requires a copy of the matrix to guarantee independence for parallel processing, and this also leads to redundant operations. To solve these memory and computational overheads of the line-based approach, we propose a novel work decomposition method that divides the 2D-FFT computation work into sub-tasks whose workspaces are disjoint (Sec. III). Based on the decomposition method, we introduce an in-place parallel 2D-FFT algorithm that does not require additional memory space or redundant operations (Sec. IV-A and Sec. IV-B). Then, we extend our in-place parallel algorithm to utilize both multi-core CPU and GPU. Unlike prior hybrid parallel approaches that allocate a region of the matrix with a static scheduling scheme, we distribute the sub-tasks to the available computing resources (e.g., the CPU cores and GPU). Also, we propose a priority-based dynamic scheduling method to maximize the utilization efficiency of the heterogeneous computing system (Sec. IV-C).

To check the benefits of our method, we implemented seven different 2D-FFT algorithms, including two versions of our methods. On three different heterogeneous computing systems, we tested the performance of the 2D-FFT algorithms with large-scale matrices from $20K^2$ to $120K^2$ in both single- and double-precision (Sec. V). Compared with the conventional homogeneous parallel algorithms based on the state-of-the-art CPU (i.e., FFTW) and the state-of-the-art GPU (i.e., cuFFT) libraries, our HI-FFT achieved up 2.92 and 4.42 times higher performance, respectively. Our method also showed better (e.g., 69% on average) performance than prior heterogeneous parallel algorithms, while using less CPU memory space. Also, we found that our method, working in an in-place manner, could successfully handle up to two times larger 2D-FFT problems than prior out-of-place parallel algorithms. In addition, the multi-core CPUs-only version of our method (i.e., $HI\text{-}FFT_{CPU}$) achieved better performance than the conventional CPU parallel algorithms using a line-based work decomposition approach. Also, we found that $HI\text{-}FFT_{CPU}$ had better performance than heterogeneous algorithms in some cases. These results demonstrate the advantages of our approach in terms of both memory usage and computational performance.

II. RELATED WORK

Various parallel algorithms have been proposed to accelerate 2D-FFT computation, including FPGA-based [21]–[24], multi-core CPU-based [7], [8], [25]–[28], and GPU-based [18], [29]–[35] approaches.

FFTW (Fastest Fourier Transform in the West) [7] is one of the most well-known multi-core CPU-based FFT libraries. FFTW utilizes the SIMD (Single Instruction, Multiple Data) units in CPU cores to maximize DFT (Discrete Fourier Transform) performance. The MKL (Math Kernel Library) [8] is a math library optimized for Intel's CPU, and it supports FFT computation. By using multi-core CPUs, both libraries provide much higher performance than a serial algorithm for FFT computation. Khokhriakov *et al.* [28] employed these libraries for 2D-FFT computation. They also took a load-imbalancing parallel computing method to optimize 2D-FFT computation on multi-core CPUs (e.g., Intel Haswell CPUs), and it achieved up to 9.4 times higher performance.

The GPU has also been actively employed to accelerate FFT computation [18], [29], [30], [32], [33]. cuFFT (CUDA Fast Fourier Transform library) [11] is one of the state-of-the-art GPU-based FFT libraries introduced by Nvidia. Although such GPU-based FFT algorithms generally show much higher performance than a CPU, it has two points of limitations. First, it is hard to handle large-scale data since the GPU has limited memory space (e.g., 2-24GB). The second limitation is memory transfer overhead. Although the memory transfer bandwidth between the host and device has been improved, it is still one of the performance bottlenecks for the GPU algorithm, especially for handling large-scale data. Gu *et al.* [18] decomposed the work of 2D-FFT into a set of 1D-FFTs based on the Cooley-Turkey algorithm [5] to perform 2D-FFT with the GPU's limited memory space. They also proposed a blocked buffer method for 1D-FFT computation, to optimize data transfer overhead. As a result, their algorithm achieved up to 2.11 times higher performance than the CPU-based algorithm for a large-scale 2D-FFT problem.

Unlike algorithms using one of the parallel computing resources (e.g., CPU or GPU), recent works have proposed utilizing different types of processors at once [19], [20], [36]–[38]. Wu and JaJa [36] employed both CPU and GPU to perform 3D-FFT for large-scale three-dimensional data. They distributed the workload to the CPU and GPU while hiding the data transfer overhead with multiple streams. For 2D-FFT computation, Ogata *et al.* [19] and Chen and Li [20] utilized a heterogeneous computing system with multi-core CPU and GPU. These works designed performance models to predict the processing time of the 2D-FFT computation on the CPU and GPU, and they tuned the models empirically. Then, they allocated the appropriate workload for each computing resource depending on its computing capability, so that the entire processing time was minimized.

To handle large-scale data with limited GPU memory, they set a line (e.g., row or column) as the basic work unit. Then, they computed 2D-FFT results by performing 1D-FFTs for columns and rows. To avoid synchronization overhead among parallel processing units (e.g., threads), the line-level work distribution approach used additional memory space (up to two times of data). Ogata *et al.* [19] transposed the matrix before the column-wise 1D-FFT and used a row-wise 1D-FFT module to improve processing performance. Since the column elements are placed discontinuously in the memory, it leads to low performance from the perspectives of cache utilization efficiency and data transfer between host and device [25], [39]. Therefore, this transposition strategy improved the performance over the naive method. As a result, they achieved up to 1.50 times performance improvement compared with using a CPU only. Our method also adopted this transposition strategy.

Like the prior heterogeneous algorithms, we utilized both multi-core CPUs and GPU. However, instead of line-level decomposition, we propose a novel decomposition method that does not require additional memory space, while guaranteeing independence among tasks (Sec. III). Based on the work decomposition method, our parallel algorithm performs in-place processing. Also, we propose a simple dynamic work scheduling algorithm that does not require empirical parameter tuning (Sec. IV).

III. WORK DECOMPOSITION

In this section, we first briefly explain the general process of 2D-FFT computation. Then, we introduce the work decomposition method in our approach.

A. GENERAL PROCESS OF 2D-FFT COMPUTATION

A Fast Fourier transform (FFT) is an acceleration algorithm that computes the Discrete Fourier transform (DFT). Let $f(x, y)$ be an element of an $N \times N$ complex matrix where x and y are row and column indices, and the two-dimensional DFT result of the matrix is defined by Eq. 1.

$$F(O_x, O_y) = \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} \mathrm{f}(x, y) \times \mathrm{e}^{-j2\pi(O_x \frac{x}{N} + O_y \frac{y}{N})}, \quad (1)$$

where j is $\sqrt{-1}$. To reduce the high computational cost ($O(N^4)$), it usually employs the row-column decomposition method, which computes the 2D-DFT using a series of 1D-FFTs [39]. We can summarize this approach mathematically using Eq. 2.

$$\begin{aligned} F(O_x, O_y) &= \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} \mathrm{f}(x, y) \times \mathrm{e}^{-j2\pi(O_x \frac{x}{N} + O_y \frac{y}{N})} \\ &= \sum_{x=0}^{N-1} \left[\sum_{y=0}^{N-1} \mathrm{f}(x, y) \times \mathrm{e}^{-j2\pi(O_y \frac{y}{N})} \right] \mathrm{e}^{-j2\pi(O_x \frac{x}{N})} \\ &= \sum_{x=0}^{N-1} \mathrm{f}'(x, y) \mathrm{e}^{-j2\pi(O_x \frac{x}{N})} \end{aligned} \quad (2)$$

With this row-column decomposition method, the computational cost is reduced to $O(N^2 log(N))$.

The computational process consists of two steps, including column-wise FFT and row-wise FFT. Since a matrix has a row-major layout in the memory, the column-wise FFT accesses the discontinuous memory region. This type of memory access pattern affects cache utilization efficiency, leading to lower processing performance than the row-wise 1D-FFT cases. This discontinuous memory layout is also inefficient when using a GPU. It requires multiple data copy API calls to send data (e.g., column) in discontinuous memory space, and we need to send the data to the GPU memory first to use GPU computation. By transposing the column before the column-wise 1D-FFT and processing the transposed column with the row-wise 1D-FFT process, we can improve the processing efficiency of the column-wise 1D-FFT step. Although this strategy requires additional data transposition steps before and after the column-wise 1D-FFT computation, it provides better performance than the naive column-wise processing. Therefore, this transposition strategy has been widely employed in many prior works, especially when handling a large complex matrix with a GPU [19], [38], [40]. The process of the transposition-based 2D-FFT is *Transpose* $\rightarrow$ *FFT* $\rightarrow$ *Transpose* $\rightarrow$ *FFT*. Our method is based on this process, and we define the combination of *Transpose* and *FFT* as a stage. Therefore, our algorithm consists of two stages, including column- and row-wise FFT stages.

B. OUR WORK DECOMPOSITION METHOD

In previous parallel algorithms for 2D-FFT, the basic work distribution unit is usually a line, like a row or a column [19], [20], [38]. The lines are allocated to available computing resources depending on the particular load-balancing strategies, and each computing resource processes the given line. When we employ the transpose-based 2D-FFT process, a computing resource performs both transposition and 1D-FFT computation for the line. However, we found two issues with this process.

- *Out-of-Place Computation and Memory Overhead:* Since the transposition operation exchanges two elements in different rows and columns (e.g., $f(i, j) \leftrightarrow f(j, i)$), transposition for a line affects other lines. To process multiple lines in parallel, we need to guarantee the independence among the lines in the workspace. We can guarantee it by making a copy of the matrix. However, such out-of-place computation requires up to two times more memory space than an in-place algorithm, which uses no auxiliary space. This memory overhead can significantly lower computational performance when we handle a large-scale 2D-FFT problem that needs to use virtual memory space (e.g., when the data size is more than half of the system memory space). In other words, the out-of-place 2D-FFT algorithm halves the maximum matrix size that the algorithm can handle efficiently.

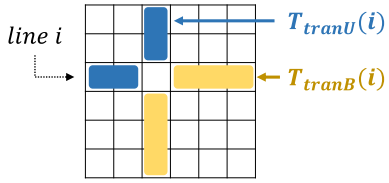


FIGURE 1. The two types of transposition task in our work decomposition method.

- *Redundant Transposition Operations:* When we use a copy of the input matrix for parallel processing, we perform transposition for each line independently. However, not all lines are actually independent, and transposing a line includes the same transposition operations for an element of other lines. For instance, transposing the first row ($f(0, j)$) is the same work as transposition of the first element of other lines ($f(j, 0)$, $j = \{1, \ldots, N-1\}$). Therefore, handling each line independently for transposition does the same work two times. In other words, it requires twice as many operations as it actually needs.

To avoid such memory and computational overhead from the line-level work distribution approach, we decompose the transposition work for a line in two ways (Fig. 1).

- $T_{transB}(i)$ is a set of transposition operations for $f(i, j)$, $\forall j \in \{i+1, \ldots, N-1\}$.
- $T_{transU}(i)$ is a set of transposition operations for $f(i, j)$, $\forall j \in \{0, \ldots, i-1\}$.

When i is the line (or row) number, the criteria that defines the two task types is the accessing direction from the line. $T_{transB}(i)$ accesses only the region of the matrix below the line. Also, it performs the transposition operation from the $(i+1)$-th element since its former elements are handled by the $T_{transB}(k)$s, $k < i$. On the other hand, we can define the transposition task that accesses the region of the matrix above the line, like $T_{transU}(i)$. In $T_{transU}(i)$, it performs transposition tasks from the first element to the (i-1)-th element, while the latter elements are transposed by the $T_{transU}(k)$s, $k > i$. Based on the task definitions, we can get the matrix transposition result without duplicate operations by performing a set of $T_{transB}(i)$s or $T_{transU}(i)$s for all lines.

In some cases, our algorithm needs to perform $T_{transB}(i)$ after $T_{transU}(k)$ $(k > i)$ is done. We define a subset task of $T_{transB}(i)$ to handle such cases.

- $T_{transB}(i, k)$ is a set of transposition operations for $f(i, j)$, $\forall j \in \{max(i+1, k), \ldots, N-1\}$.

Finally, we define the FFT computation task,

- $T_{FFT}(i)$ is FFT computation for line i.

Therefore, we have four types of task for 2D-FFT computation, $T_{transB}(\cdot)$, $T_{transU}(\cdot)$, $T_{transB}(\cdot, \cdot)$, and $T_{FFT}(\cdot)$.

IV. HI-FFT ALGORITHM

In this section, we first introduce an overview of our HI-FFT (**H**eterogeneous parallel **I**n-place 2D-**FFT**) framework. Then, we explain the details of our algorithm.

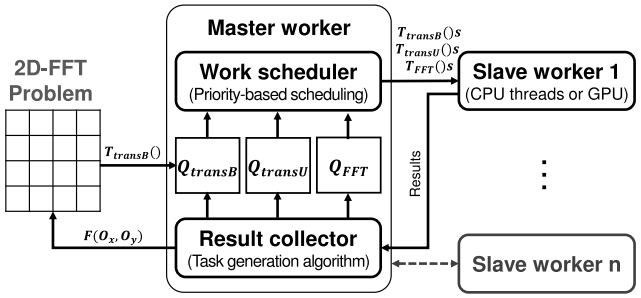


FIGURE 2. HI-FFT framework overview.

A. HI-FFT FRAMEWORK

Fig. 2 shows an overview of the HI-FFT framework. It consists of a master worker and slave workers. The *master worker* consists of three components: 1) work scheduler, 2) three task queues, and 3) result collector. There are three task queues, Q_{transB}, Q_{transU}, and Q_{FFT}. Each task queue manages each task type, while Q_{transB} handles both $T_{transB}(\cdot)$ and $T_{transB}(\cdot, \cdot)$ because $T_{transB}(i)$ is equal to $T_{transB}(i, i+1)$. When our HI-FFT framework gets an input matrix, it pushes $T_{transB}(\cdot)$ for all lines to Q_{transB} from the first line. At this time, there is no task in the other task queues. The *work scheduler* pops tasks from the queues and allocates them to the slave workers based on our scheduling algorithm (Sec. IV-C). Each *slave worker* has processing modules for the four task types. In our framework, a GPU or a set of CPU threads can compose a slave worker. Once a slave worker gets a task, it processes the task and returns the results to the result collector in the master worker. Then, the *result collector* takes the result and checks whether it meets a condition for generating other tasks (Sec. IV-B). If it meets one of the conditions, it pushes available tasks to the task queues. The HI-FFT framework repeats these processes until there are no tasks anymore and all the slaver workers have finished their given tasks. Then, we finally get the 2D-FFT result for the input matrix.

B. TASK GENERATION ALGORITHM

With an out-of-place parallel algorithm, a naive approach distributes the lines to available computing resources (e.g., threads) because all the lines are independent from each other. In this case, the straightforward processing order for line i is $T_{trans_1}(i) \rightarrow T_{FFT_1}(i) \rightarrow T_{trans_2}(i) \rightarrow T_{FFT_2}(i)$, where $T_{trans}(i)$ is a set of transposition operations for all elements in line i. Note that we have numbered each task (e.g., T_{OO_1} and T_{OO_2}) to distinguish the column-wise$_1$ and row-wise$_2$ stages. Similarly, for line i, the processing order of our in-place algorithm can be $T_{transB_1}(i) \rightarrow T_{FFT_1}(i) \rightarrow T_{transB_2}(i) \rightarrow T_{FFT_2}(i)$. However, all the lines are dependent on each other in our in-place algorithm. We need to consider the dependencies among tasks to design a parallel algorithm. As an in-place algorithm, there are three dependencies among tasks of different lines,

- *D1:* $T_{FFT_1}(i)$ should be processed after completing $T_{transB_1}(j)$s for all $j \leq i$.

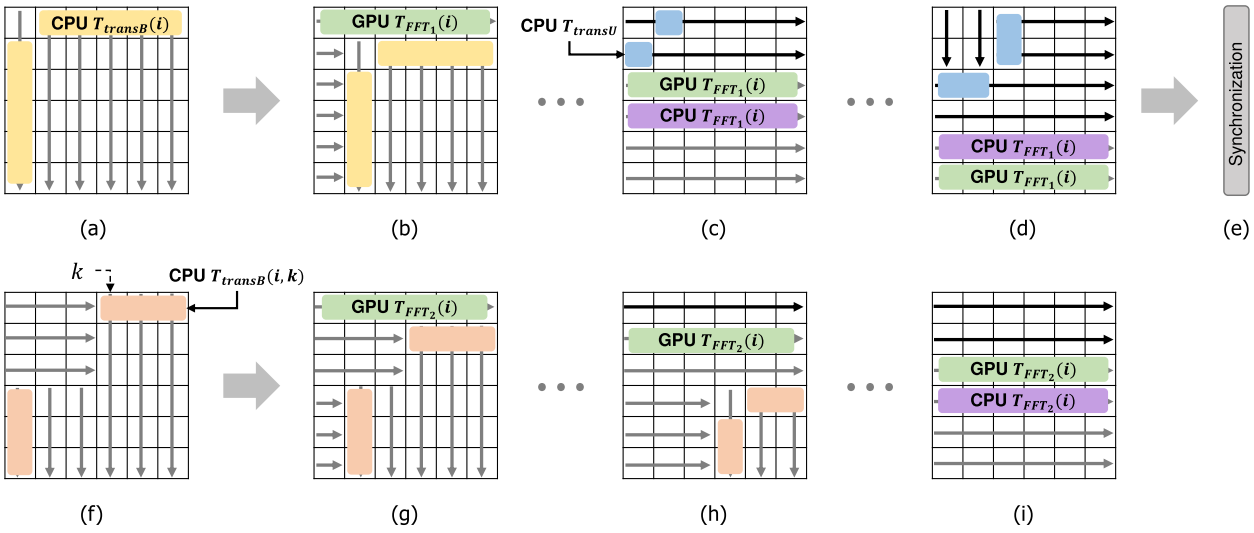


FIGURE 3. This figure shows the process of our HI-FFT algorithm when it has two CPU slave workers and one GPU worker.

- *D2:* $T_{transB_2}(i)$ should be processed after completing $T_{FFT_1}(\cdot)$s for all lines.
- *D3:* $T_{FFT_2}(i)$ should be processed after completing $T_{transB_2}(j)$s for all $j \leq i$.

The first and third dependencies are caused by the fact that $T_{FFT}(i)$ requires all the elements of line i, while the transposition operation for $f(i,j)$ is performed by $T_{transB}(j)$s for all $j \leq i$. The second dependency is defined as the $T_{transB}(i)$ accesses the i-th elements of other lines below line i while we process $T_{FFT_1}(\cdot)$s from the first line.

Tasks with dependencies require sequential processing, and it is crucial to remove or reduce the dependency to improve the degree of parallelism [41]. We can relax the second dependency by employing $T_{transU}(i)$ and $T_{transB}(i,k)$. Since $T_{transU}(\cdot)$ accesses only the lines on the upper side of line i, **D2** is decomposed into two sub-dependencies, and **D3** is changed.

- *D2-1:* $T_{transU}(i)$ should be performed after $T_{FFT_1}(j)$s are finished for all $j \leq i$.
- *D2-2:* $T_{transB}(\cdot,k)$ can be processed after completing $T_{FFT_1}(\cdot)$ for all lines.
- *D3′:* $T_{FFT_2}(i)$ can be performed after $T_{transU}(i)$ and $T_{transB}(i,k)$s where $k = i+1$ (or $k > (i+1)$ if $T_{transU}(j)$s are finished, $\forall j \in \{i+2, \ldots, k-1\}$).

Based on these dependencies, we define four task generation conditions.

- *C1:* If $T_{transB}(j)$s for all $j \leq i$ are finished (**D1**), it generates $T_{FFT_1}(i)$
- *C2:* If $T_{FFT_1}(j)$s for $j \leq i$ (**D2-1**) are finished, it generates $T_{transU}(i)$
- *C3:* If $T_{FFT_1}(\cdot)$ for all lines are finished (**D2-2**), it pops all the remaining $T_{transU}(i)$s from Q_{transU} and generates $T_{transB}(i,k)$s where $k-1$ is the last line $T_{transU}()$ is performed.
- *C4:* If $T_{transB}(i,k)$ is finished (**D3′**), it generates $T_{FFT_2}(i)$.

C1 and **C2** are intuitive according to the **D1** and **D2-1**. In our HI-FFT algorithm, the GPU's role is an accelerator for FFT computation (Sec. IV-C), and it needs to generate FFT tasks (i.e., $T_{FFT}(\cdot)$) as soon as possible to utilize the GPU intensively. We designed **C3** and **C4** to meet this demand. Because our framework uses queues, $T_{transU}(\cdot)$s are sequentially popped from the first line. This means that all the $T_{transU}(i)$ for $i < k$ are processed if we wait until all the slave workers finish their current tasks. To make sure of this, our framework has a synchronization step when it meets **C3**. After synchronization, finishing $T_{transB}(i,k)$ means that line i is ready for $T_{FFT_2}(i)$ (**C4**).

Fig. 3 shows the process overview of the HI-FFT algorithm with our task generation algorithm when it has two CPU slave workers and one GPU worker. Initially, we have $T_{transB}(\cdot)$s for all lines. CPU slave workers processes $T_{transB}(\cdot)$s from the first line (Fig. 3-(a)). As $T_{transB}(\cdot)$s are processed, $T_{FFT_1}(\cdot)$s are generated according to **C1**, and the GPU slave worker process them from the first line (Fig. 3-(b)). After the GPU worker completes $T_{FFT_1}(j)$ for $j \leq i$, $T_{transU}(i)$ is generated depending on **C2**. Once CPU slave workers process all the $T_{transB}(\cdot)$s, they start to process $T_{transU}(\cdot)$s (Fig. 3-(c) and (d)). Meanwhile, CPU slave workers also participate in processing $T_{FFT_1}(\cdot)$s since the workload for $T_{transU}()$ is much smaller than $T_{FFT}()$. When all $T_{FFT_1}(\cdot)$s were finished, it pops remaining $T_{transU}(\cdot)$s from Q_{transU} and generates $T_{transB}(\cdot,\cdot)$s according to **C3** (Fig. 3-(e)). Then, CPU slaver workers process $T_{transB}(\cdot,\cdot)$s (Fig. 3-(f)). As $T_{transB}(\cdot,\cdot)$s are processed, $T_{FFT_2}(\cdot)$s are generated (**C4**) while the GPU slave worker takes them (Fig. 3-(g) and (h)). Once CPU slave workers complete all $T_{transB}(\cdot,\cdot)$s, they start to process $T_{FFT_2}(\cdot)$s (Fig. 3-(i)).

C. PRIORITY-BASED DYNAMIC SCHEDULING

The work scheduler in the master worker distributes tasks in the task queues to the slave workers. We employ a

priority-based dynamic scheduling strategy. From a high-level, there are two types of tasks, transposition ($T_{transB}()$ and $T_{transU}()$) and FFT computation ($T_{FFT}()$). Also, we have two types of computing resources, multi-core CPU and GPU. The transposition tasks' major operation is exchanging the two elements which reside in different memory regions. In other words, those tasks are memory-bounded work having a non-regular memory access pattern. In contrast, $T_{FFT}()$ is a compute-intensive task that accesses the continuous memory region. The properties for each task type are matched with the architectural characteristics of the CPU and GPU.

At first, the CPU has a well-organized cache hierarchy, which improves the memory access efficiency for random memory access patterns [16], [25], [42]. Also, the CPU can access the system memory directly, unlike the GPU which requires copying data from the system memory to the device memory. Therefore, we give a higher priority than $T_{FFT}()$ to the transposition tasks for slave workers consisting of CPU threads. Between transposition tasks, $T_{transB}()$s have a higher priority, to generate $T_{FFT}()$s as soon as possible. In summary, the task priority for CPU is $T_{transB}() > T_{transU}() \geq T_{FFT}()$.

Unlike the CPU, the GPU has many cores (e.g., hundreds or thousands), and it has an optimized architecture for regular streaming floating-point operations [16], [43], [44]. This architectural property matches the characteristic of $T_{FFT}()$, and the GPU has a much higher performance than a multi-core CPU for FFT computation. On the other hand, the transposition tasks have a non-regular memory access pattern that is not suitable for the GPU architecture. Also, using GPU for the transposition tasks is rather a loss, due to the overhead for transferring data between host and device memories. We found that the transfer time itself is larger than the time needed for transposition on the CPU-side. As a result, we employ the GPU as an FFT computation accelerator. The workload for FFT computation is much larger than the transpositions, and we can utilize GPU intensively if it generates $T_{FFT}()$s as soon as possible. The task priority of the CPU slave worker and the task generation conditions (**C3** and **C4**) reflect this. Algorithm 1 is the pseudo-code of our HI-FFT algorithm that the priority-based dynamic scheduling algorithm is applied.

Scheduling Granularity: To reduce the scheduling overhead, including communication overhead between master and slaves, we distribute tasks as a unit of a block which is a set of tasks with the same task type. For example, a block of FFT computation is $\{T_{FFT}(i), \ldots, T_{FFT}(i+b-1)\}$, where b is the block size. A larger block leads to better GPU computational efficiency, but it can waste the GPU's computing power while transferring the block to the device memory [45], [46]. To hide the data transfer time and fully utilize the GPU's computing capability, we overlap the data transfer and computation using multiple streams [43]. In this way, we use the maximum number of lines the device memory can hold for all streams as the block size.

Algorithm 1: Pseudo Code of the HI-FFT Algorithm

```
1  k ← 0
2  Q_transB ← T_transB(i), ∀i ∈ {1, ..., (N : matrix size)}
3  Run all slave workers in parallel
4      [CPU slave worker]
5          while Q_transB is not empty do
6              T_transB(i) ← Q_transB
7              process T_transB(i)
8              Q_FFT ← T_FFT1(i)                    // C1
9          while Q_FFT is not empty do
10             if Q_transU is not empty then
11                 T_transU(i) ← Q_transU
12                 process T_transU(i)
13                 k++
14             if Q_FFT is not empty then
15                 T_FFT1(i) ← Q_FFT
16                 process T_FFT1(i)
17                 Q_transU ← T_transU(i)           // C2
18     [GPU slave worker]
19         while Q_FFT or Q_transB is not empty do
20             T_FFT1(i) ← Q_FFT
21             process T_FFT1(i)
22             Q_transU ← T_transU(i)               // C2
23
24 Global synchronization                           ▷ Fig.3-(e)
25 T_transU(i) ← Q_transU
26 Q_treansB ← T_transB(i, k)                       // C3
27
28 Run all slave workers in parallel
29     [CPU slave worker]
30         while Q_transB is not empty do
31             T_transB(i, k) ← Q_transB
32             process T_transB(i, k)
33             Q_FFT ← T_FFT2(i)                    // C4
34         while Q_FFT is not empty do
35             T_FFT2(i) ← Q_FFT
36             process T_FFT2(i)
37     [GPU slave worker]
38         while Q_transB or Q_FFT is not empty do
39             if Q_FFT is not empty then
40                 T_FFT2(i) ← Q_FFT
41                 process T_FFT2(i)
```

V. RESULTS AND ANALYSIS

We implemented our HI-FFT algorithm on three different heterogeneous machines consisting of multi-core CPU(s) and a GPU (Table 1). The three machines had different GPUs. The GTX 1060 in Machine 1 is a low-end GPU with the lowest performance and smallest device memory space. The two GPUs in Machine 2 and 3 have the same device memory size, but the RTX 3090 has a higher computational performance. Machine 1 and 2 share the same base system, and they have larger host memory space than Machine 3. On the other hand, Machine 3 has a more powerful CPU with PCIe 4.0 and supports higher bandwidth between host and device memories than Machine 1 and 2. We used OpenMP [47] and

TABLE 1. Configurations of the three heterogeneous computing machines used in our experiments.

	Machine 1	Machine 2	Machine 3
CPU	Two Intel Xeon CPUs (2×8 cores, 2.10 Ghz)		AMD Ryzen CPU (24 cores, 3.80Ghz)
System memory	386 GB		256 GB
GPU	GTX 1060 (6 GB)	Titan RTX (24 GB)	RTX 3090 (24 GB)
PCI	PCIe 3.0 x16		PCIe 4.0 x16

TABLE 2. This table shows the block sizes used in our experiment on each machine depending on the matrix size for single- and double-precision cases.

Matrix size	$20K^2$	$40K^2$	$60K^2$	$80K^2$	$100K^2$	$120K^2$
Single-precision						
Mach. 1	1,000	1,000	500	500	250	100
Mach. 2&3	1,000	1,000	1,000	1,000	1,000	1,000
Double-precision						
Mach. 1	1,000	500	500	250	100	100
Mach. 2&3	1,000	1,000	1,000	1,000	1,000	500

CUDA 11.0 [10] to implement the parallel algorithms on the multi-core CPU and GPU. To perform $T_{FFT}()$, we employed the FFTW 3.3 [7] and cuFFT 11.0 [11] libraries for the slave workers in the multi-core CPU and GPU, respectively. We used four streams, and the block sizes are shown in Table 2. We implemented two versions of our method.

- *HI-FFT* is the implementation of our algorithm with two CPU slave workers and one GPU slaver worker. Each CPU slave worker uses sixteen threads on Machine 1 & 2, and twenty-four threads on Machine 3.
- *HI-FFT*$_{CPU}$ is an alternative version of our method. It uses only two CPU slaver workers without a GPU slave worker. The configuration for CPU slaver workers is the same as the *HI-FFT*.

To check the benefits of our method, we also implemented five alternative algorithms based on prior approaches.

- CPU_{FFTW} is based on the state-of-the-art CPU-based FFT library, FFTW [7], respectively. We applied the transposition-based 2D-FFT process, which uses the line-level work decomposition method. To make it an in-place algorithm, we put a synchronization step between the transposition and FFT computation steps. For this method, we used sixteen and twenty-four CPU threads for Machine 1 and 2, and Machine 3, respectively.
- CPU_{MKL} is an alternative implementation of a CPU-based 2D-FFT algorithm based on the MKL library [8]. The only difference from CPU_{FFTW} is that it uses the FFT functions of the MKL 2021.3.0 library instead of the FFTW library.
- GPU_{CUFFT} is based on cuFFT, which is a GPU-based FFT library highly optimized for Nvidia GPUs [11]. If the matrix is smaller than the device memory size (marked by the asterisk in Table. 3), it loads the entire matrix onto the GPU first and runs the 2D-FFT module of the cuFFT library. Otherwise, it employs the row-column decomposition strategy. For columns, it transfers the lines to the device memory by cudaMemcpy2D() API instead of using the transposition-based approach. We used the same block size with our method and four streams to minimize the data transfer overhead.
- *Ogata* is an implementation of the heterogeneous parallel algorithm presented by Ogata *et al.* [19]. This method uses the line-level work decomposition method and out-of-place computation with a transposition-based 2D-FFT strategy. Although they reported that allocating a 60% workload (i.e., lines) to the GPU achieved the best performance, we found that it depended on the system configurations. Therefore, we measured the performance while changing the GPU workload from 10% to 90% (in 10% increments) and took the best performance for comparison.
- *Chen* is an implementation of the heterogeneous parallel algorithm proposed by Chen and Li [20]. Like *Ogata*, it is an out-of-place algorithm and the basic work unit is a line. This algorithm needs to set the entire workspace on the host memory as page-locked (or pinned) memory, and the maximum size of the input matrix it can handle is smaller than *Ogata*. Like *Ogata*, we recorded the best performance among different conditions for the workload distribution between CPU and GPU.

For *Ogata* and *Chen*, we used the same number of CPU threads as the number of CPU cores on each machine.

Benchmarks: To measure the performance of the different 2D-FFT algorithms, we generated a set of complex matrices in both single- and double-precision. We randomly generated complex numbers, and the matrix sizes are varied from $20K^2$ to $120K^2$ while the matrix's data sizes varied from 3 GB to 215 GB. For each matrix, we performed the 2D-FFT computation five times and reported the average processing time.

A. RESULTS

Table 3 shows the processing times of the seven different 2D-FFT algorithms. Since the out-of-place approach requires up to two times more memory space of the input matrix, the two out-of-place algorithms failed to process the large-scale 2D-FFT problems. For the double-precision complex matrices, *Ogata* failed to handle a $120K^2$ matrix on Machine 1 & 2 and $100K^2$ on Machine 3 because the required memory space (e.g., 2 × 215 GB for $120K^2$) exceeded the system memory size. The problem size that *Chen* could process was much smaller than *Ogata*, since only a part of the system memory is allowed to be a page-locked memory. Therefore, *Chen* failed to process the $100K^2$ and $120K^2$ complex matrices in single precision on Machine 1 & 2 and Machine 3, respectively. Also, it could not treat the $80K^2$ complex matrix in double precision. In contrast, the in-place algorithms, including our *HI-FFT*, successfully processed the large-scale 2D-FFT problems.

TABLE 3. This table shows the 2D-FFT computation times (seconds) of seven different algorithms on three machines for different input matrices in single- and double-precision. For Machine 2, the processing times of the CPU_{MKL}, CPU_{FFTW} and $HI\text{-}FFT_{CPU}$ are the same as Machine 1 since they used the same CPUs. An asterisk(*) denotes that the GPU_{CUFFT} performed 2D-FFT after loading all of the data on the device memory. A dash(-) denotes that the algorithm failed to process, due to out-of-memory. The bold font and the bold-italic font mark the best and the second-best performance among the seven algorithms, respectively.

Single-precision							
Input matrix size (data size)		$20K^2$ (2.98 GB)	$40K^2$ (11.92 GB)	$60K^2$ (26.82 GB)	$80K^2$ (47.68 GB)	$100K^2$ (74.50 GB)	$120K^2$ (107.28 GB)
Machine 1	CPU_{MKL}	1.71	6.54	17.02	34.95	60.05	105.12
	CPU_{FFTW}	2.17	8.30	19.47	35.02	58.97	104.76
	$HI\text{-}FFT_{CPU}$	2.15	7.94	15.96	31.83	51.43	***88.11***
	GPU_{CUFFT}	3.45	14.05	39.15	58.52	105.74	171.61
	Ogata	1.69	6.21	14.29	35.18	***45.06***	89.24
	Chen	***1.54***	***5.41***	***13.22***	***24.06***	-	-
	HI-FFT	**1.25**	**4.28**	**9.04**	**15.49**	**25.83**	**38.76**
Machine 2	GPU_{CUFFT}	1.95*	10.76	27.49	44.46	64.26	116.63
	Ogata	1.57	6.22	13.96	33.17	***42.28***	***81.26***
	Chen	***1.28***	***4.57***	***10.82***	***20.46***	-	-
	HI-FFT	**1.24**	**4.10**	**8.88**	**15.40**	**24.21**	**35.88**
Machine 3	CPU_{MKL}	1.22	4.45	11.33	18.96	30.49	41.20
	CPU_{FFTW}	1.11	4.65	9.29	15.94	26.06	41.28
	$HI\text{-}FFT_{CPU}$	0.95	3.49	7.84	13.25	21.08	32.15
	GPU_{CUFFT}	***0.70****	4.67	10.76	21.35	31.89	49.16
	Ogata	0.78	3.03	6.85	***11.85***	***19.23***	***30.07***
	Chen	0.71	***2.61***	***6.24***	12.03	-	-
	HI-FFT	**0.63**	**2.23**	**4.76**	**8.13**	**12.56**	**18.43**

Double-precision							
Input matrix size (data size)		$20K^2$ (5.96 GB)	$40K^2$ (23.84 GB)	$60K^2$ (53.64 GB)	$80K^2$ (95.37 GB)	$100K^2$ (149.01 GB)	$120K^2$ (214.57 GB)
Machine 1	CPU_{MKL}	3.66	15.12	37.24	59.21	101.71	194.98
	CPU_{FFTW}	5.48	15.01	24.55	38.48	56.31	94.38
	$HI\text{-}FFT_{CPU}$	5.13	13.12	***21.93***	***34.42***	***50.15***	***73.59***
	GPU_{CUFFT}	6.73	27.06	68.24	112.88	197.23	265.34
	Ogata	3.03	***11.65***	25.85	53.42	80.63	-
	Chen	***3.03***	12.4	27.96	-	-	-
	HI-FFT	**2.35**	**7.41**	**15.59**	**27.63**	**46.7**	**61.44**
Machine 2	GPU_{CUFFT}	4.43*	22.02	46.86	91.52	144.33	197.94
	Ogata	2.72	11.43	24.38	51.57	66.72	-
	Chen	***2.54***	***10.02***	25.35	-	-	-
	HI-FFT	**2.16**	**7.22**	**14.78**	**24.46**	**36.69**	**54.33**
Machine 3	CPU_{MKL}	2.04	7.88	17.80	32.30	53.74	69.44
	CPU_{FFTW}	3.68	10.45	18.21	28.59	43.19	61.64
	$HI\text{-}FFT_{CPU}$	3.35	9.02	16.67	26.11	***41.36***	***57.54***
	GPU_{CUFFT}	1.54*	8.97	18.5	37.61	57.90	82.53
	Ogata	1.72	6.39	***13.83***	***25.21***	-	-
	Chen	***1.45***	***5.89***	15.11	-	-	-
	HI-FFT	**1.41**	**4.82**	**10.29**	**18.79**	**30.04**	***43.54***

CPU_{MKL} shows a comparable or a little higher performance than CPU_{FFTW} for small matrices. However, for large-scale matrices, CPU_{FFTW} generally achieved a better performance (e.g., 10% on average) than CPU_{MKL}, especially for the double-precision case. $HI\text{-}FFT_{CPU}$ generally showed higher performance than two alternative CPU-based algorithms. It achieved up to 33% and 28% (18% and 12% on average) better performance than the CPU_{FFTW} for single- and double-precision cases, respectively. Compared with CPU_{MKL}, $HI\text{-}FFT_{CPU}$ showed up to 44% and 165% (19% and 35% on average) higher performance for single- and double-precision cases, respectively. For a larger matrix, $HI\text{-}FFT_{CPU}$ tended to show more performance gaps from these two CPU-based algorithms. Since our method requires fewer operations for the transposition task and has higher parallelism than CPU_{FFTW} and CPU_{MKL}, $HI\text{-}FFT_{CPU}$ achieved higher performance than these two alternatives with the same computing resources in most cases.

GPU_{CUFFT} achieved higher performance than the two CPU-based parallel algorithms, including CPU_{FFTW} and $HI\text{-}FFT_{CPU}$, when the input matrix was small enough to load the entire data into the device memory (denoted by the

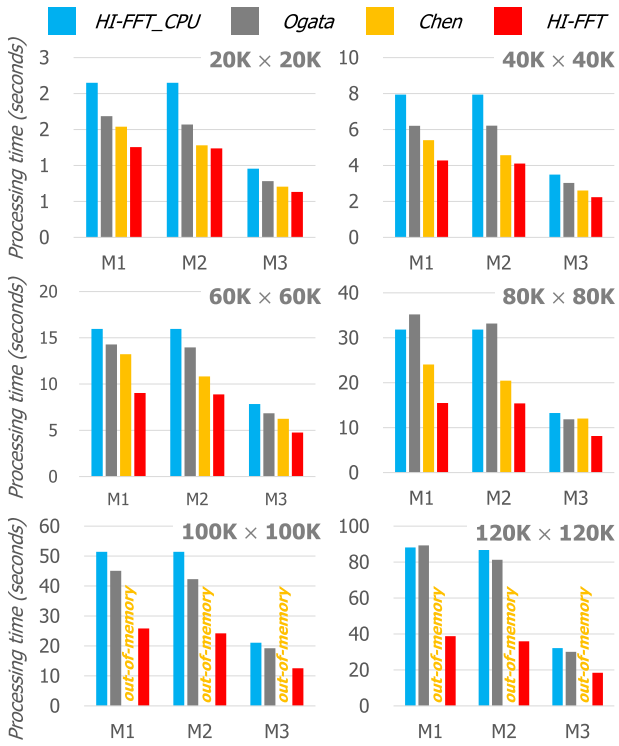


FIGURE 4. This graph compares the performance of two prior heterogeneous algorithms and ours for the single-precision case.

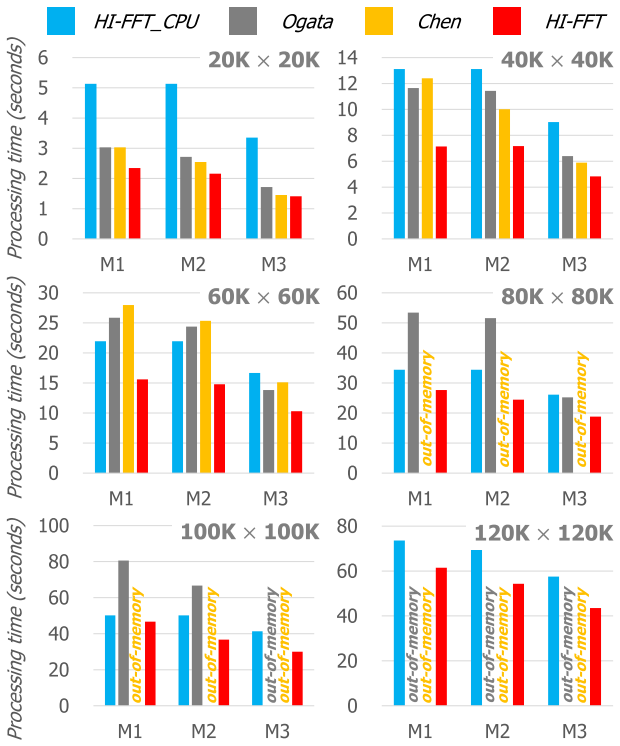


FIGURE 5. This graph compares the performance of two prior heterogeneous algorithms and ours for the double-precision case.

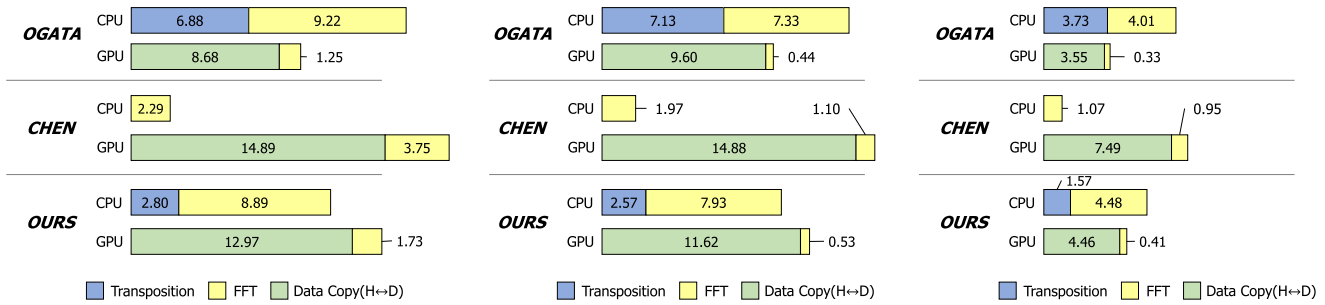
asterisk in Table 3). Also, it showed comparable performance with CPU_{MKL}. However, when the input matrix's data size was larger than the device memory and the GPU was unable to process that in an in-core manner, GPU_{CUFFT} showed a much lower performance than the CPU-based methods. For large-scale data, GPU_{CUFFT} required frequent data transfer between the host and device memories, and this communication overhead lowered performance significantly. We found that the data communication took 89% of the GPU_{CUFFT}'s processing time on average.

By employing multi-core CPU(s) in addition to a GPU, the heterogeneous 2D-FFT algorithms not only decreased the workload on the GPU but also the data communication overhead as well. Therefore, they achieved much higher performance than GPU_{CUFFT}. For single-precision, *Ogata*, *Chen*, and *HI-FFT* achieved up to 2.74, 2.96, and 4.43 times higher performance than GPU_{CUFFT}, respectively. For double-precision, they also achieved up to 2.64, 2.44, and 4.38 times higher performance than GPU_{CUFFT}, respectively.

Fig. 4 and Fig. 5 compare the 2D-FFT computation times of three heterogeneous algorithms, including two prior works and our *HI-FFT*. On average, *Chen* showed 16% and 2% higher performance than *Ogata* for single- and double-precision cases, respectively. However, *Chen* requiring large page-locked memory space failed to handle a large 2D-FFT problem that *Ogata* could process. In some cases, especially for double-precision, *Chen* showed a comparable or even worse (e.g., double-precision $60K^2$) performance than *Ogata*. We found that *Chen* had a higher reliance on the GPU than *Ogata* (Table 4). Therefore, the increase in data transfer overhead for the large 2D-FFT problems affected *Chen*'s processing performance more than *Ogata*'s (Sec. V-B).

Our *HI-FFT* achieved the best performance for all the cases, and it successfully processed large-scale 2D-FFT problems the prior two algorithms failed to handle. Compared with *Ogata*, *HI-FFT* showed up to 2.27 and 1.93 times (1.66 and 1.56 times on average) higher performance for single- and double-precision cases, respectively. Also, *HI-FFT* achieved up to 1.33 and 1.79 times (1.26 and 1.41 times on average) higher performance than *Chen* for single- and double-precision cases, respectively.

We also found that $HI\text{-}FFT_{CPU}$ achieved a similar to or even higher performance than two prior heterogeneous algorithms for large-scale 2D-FFT problems (e.g., larger than double-precision $60K^2$ on Machine 1 and 2). The data communication overhead increased as the problem size increased, but the prior static workload distribution approach was hard to correspond to such changes. Also, our in-place algorithm has a lower workload for transposition tasks than the out-of-place algorithms. As a result, $HI\text{-}FFT_{CPU}$ surpassed those prior heterogeneous algorithms for large-scale 2D-FFT problems. On the other hand, *HI-FFT* robustly showed higher performance than $HI\text{-}FFT_{CPU}$ since our scheduling algorithm dynamically controlled the workload between the CPU and GPU slave workers (Sec. V-B).



(a) Machine 1 (b) Machine 2 (c) Machine 3

FIGURE 6. This figure shows the summation of the processing times for each task type and data copy for the three heterogeneous algorithms on three machines. For this profile, we used a $60K^2$ complex matrix in single-precision.

B. PERFORMANCE ANALYSIS

To ascertain the factors that allowed the proposed approach to achieve higher performance than the prior algorithms, we measured each task's total processing time in the three heterogeneous algorithms, including ours. Fig. 6 has stacked column charts that show the total time for processing each task type on the three machines. For this profiling, we used $60K^2$ complex matrices in the single-precision. Please note that the times in the charts are the summation of the processing times of all slave workers and on all the streams, while some of them run concurrently at runtime.

For the CPU workloads, the total processing time for the transposition task in our *HI-FFT* was 2.54 and 4.24 times shorter on average than *Ogata* for single- and double-precision cases, respectively. This result was possible because the *HI-FFT* requires two times fewer operations for transposition tasks than *Ogata*. Also, our in-place algorithm has a higher spatial locality, which leads to a higher cache hit ratio than the out-of-place algorithms. Since *Chen* does not take the transposition strategy [20], there is no transposition time on the CPU. Instead, we found that *Chen* had greater overhead for data copy than the other algorithms.

The common ground for all the methods is that data copy was the most time-consuming job, which governed the GPU's utilization efficiency. Therefore, performance improvements with Machine 2 over Machine 1 were not impressive for all algorithms because their base system was the same, although Machine 2 had a much more powerful GPU. Nonetheless, on Machine 2, all of the algorithms using GPU showed better performance. This is because the larger block increases the efficiencies of both computations on GPU and data communication. Machine 2 used a larger block size than Machine 1, depending on the device memory size.

Machine 3 supports PCIe 4.0 (32 GB/s), which supports up to two times higher bandwidth than the PCIe 3.0 (16 GB/s) on Machine 2. As shown in Fig. 6, the data transfer overhead decreased on Machine 3 compared with Machine 2. Consequently, all of the algorithms achieved much higher performance on Machine 3 than on Machine 2. This phenomenon stood out more for *Ogata* and *Chen* than *HI-FFT*, which means that the efficiency of those methods is affected by the computing environment. On the other hand, our *HI-FFT* stably brought out the best performance from the given computing resources.

We found that our scheduling algorithm, which dynamically controls the workload for available computing resources, allowed *HI-FFT* to achieve stable and high performance regardless of system configuration. Table 4 shows the ratio of $T_{FFT}(\cdot)$s processed by the GPU in three different heterogeneous algorithms for the single-precision $60K^2$ complex matrix case. As we mentioned, we measured the processing time for *Ogata* and *Chen* by changing the ratio from 10% to 90% (in 10% increments) manually. Then,

TABLE 4. This table shows the ratio of $T_{FFT}(\cdot)$s processed by the GPU slaver worker in three heterogeneous algorithms.

		Single-precision			Double-precision		
	Size	*Ogata*	*Chen*	*HI-FFT*	*Ogata*	*Chen*	*HI-FFT*
M1	$20K^2$	60%	90%	77%	60%	90%	75%
	$40K^2$	60%	90%	74%	60%	80%	60%
	$60K^2$	60%	90%	66%	40%	90%	53%
M2	$20K^2$	70%	90%	80%	60%	90%	78%
	$40K^2$	60%	90%	75%	60%	80%	64%
	$60K^2$	60%	90%	70%	50%	90%	57%
M3	$20K^2$	60%	90%	75%	70%	90%	70%
	$40K^2$	50%	90%	66%	60%	80%	52%
	$60K^2$	50%	90%	63%	50%	90%	45%

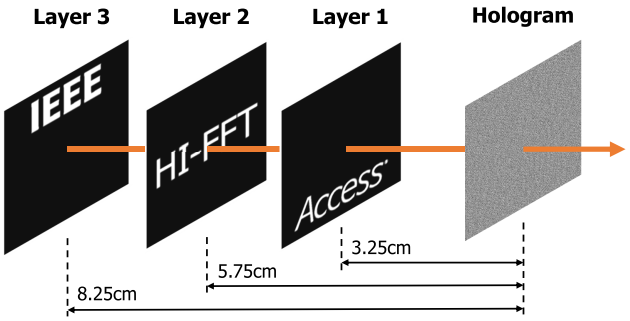


FIGURE 7. Images and distance from the hologram used in the layer-based hologram process.

(a) Focus on the layer 1 (b) Focus on the layer 2 (c) Focus on the layer 3 (d) Viewpoint: Left (e) Viewpoint: Right

FIGURE 8. These images show numerical reconstruction results of the 80K^2 hologram generated by applying our method. The first three images (a, b, c) show the reconstruction results with a different focal point. The last two images (d, e) are the reconstruction results with different viewpoints.

TABLE 5. This table shows the processing times (seconds) for ASM, applying off-axis, and other steps of the layer-based CGH. For this analysis, we used Machine 2 and generated hologram consists of three layers (Fig. 7). The bold font denotes the steps that include 2D-FFT computation.

Single-precision (hologram resolution: $80K^2$)			
2D-FFT algorithm	*Ogata*	*Chen*	*HI-FFT*
ASM	324.27	272.31	232.822
Applying off-axis	117.10	114.84	105.26
Other steps	316.54		
Total processing time	757.91	703.69	654.62
Double-precision (hologram resolution: $60K^2$)			
2D-FFT algorithm	*Ogata*	*Chen*	*HI-FFT*
ASM	209.79	237.67	171.83
Applying off-axis	92.06	98.77	80.15
Other steps	306.75		
Total processing time	608.6	643.19	558.73

we reported the best one as their performances, and Table 4 shows the ratio. On the other hand, our scheduling algorithm in *HI-FFT* does not require any empirical tuning or manual control, and it dynamically determined the appropriated work distribution at runtime depending on the processing status, as shown in Table 4. One interesting example is that Machine 2 has a much more powerful GPU than Machine 1, but *HI-FFT* distributed a little more $T_{FFT}(\cdot)$s to the GPU slave worker on Machine 2 than Machine 1. This deviates from the usual approach of allocating much more tasks to the GPU when we have a much more powerful GPU. We found that the different block sizes in Machine 1 and 2 changed the processing efficiency for each task type on each device (e.g., better efficiency for transposition with larger block size). Our dynamic scheduling algorithm coped with such organic change suitably, and these results validate the robustness of our approach.

C. APPLICATION TO COMPUTER GENERATED HOLOGRAPHY

To verify the benefit of our *HI-FFT* in an actual application, we applied three heterogeneous algorithms into the layer-based CGH (Computer Generated Holography) [48]–[50].

The layer-based CGH process consists of seven steps: reading image, random phase, Angular Spectrum Method (ASM), accumulation, applying off-axis, phase encoding, and saving the resulting hologram. Among them, the ASM computation includes 2D-FFT calculation two times for each layer, and the off-axis computation requires calculating 2D-FFT two times. These two steps are the most time consuming part of the layer-based CGH. Fig. 7 shows the layers and configuration used in our experiments. We generated $80K^2$ and $60K^2$ holograms in single- and double-precision, respectively, and used Machine 2. Fig. 8 is numerical reconstruction of $80K^2$ hologram generated with single-precision computation, and it validates that our algorithm works accurately in the application.

Table 5 shows the processing time of ASM, applying off-axis, and other steps of layer-based CGH. We applied three different 2D-FFT algorithms for ASM and applying off-axis steps. In the double-precision case, our *HI-FFT* reduced the processing time of ASM by 18% and 28% compared with using *Ogata* and *Chen*, respectively. Also, *HI-FFT* achieved 1.15 and 1.23 times higher performances than *Ogata* and *Chen*, respectively, for applying off-axis. For the single-precision case, *HI-FFT* achieved similar performance improvement for ASM and applying off-axis with the double-precision case. It needs to note that they also include other computations like element-wise multiplication, etc. As a result, our *HI-FFT* reduced hologram generation time by 14% and 7% for $80K^2$ in single-precision and 8% and 13% for $60K^2$ in double-precision, compared with using *Ogata* and *Chen*, respectively. These results demonstrate the usefulness of our approach in actual applications.

VI. CONCLUSION AND FUTURE WORK

We have presented a heterogeneous parallel in-place algorithm, *HI-FFT*, for large-scale 2D-FFT. We first figured out the memory and computational overhead in the out-of-place computation and line-level work distribution approaches used in prior parallel 2D-FFT algorithms. To solve these issues, we proposed a novel work decomposition method, in which the workspaces of the decomposed tasks are disjointed from each other, and do not have duplicated operations. Based on our work decomposition method, we introduced an in-place parallel 2D-FFT algorithm that can handle up to two times larger 2D-FFT problem compared with the

out-of-place algorithms. Then, we extended it to use both multi-core CPUs and GPU. We also proposed a priority-based dynamic scheduling algorithm to maximize the utilization efficiency of the computing resources. We found that our scheduling algorithm achieved good performance stably for a given computing machine without any empirical tuning or manual control. We implemented our method on three different heterogeneous computing systems and compared the performance with five alternative methods based on prior approaches for large-scale 2D-FFT problems. Overall, our *HI-FFT* showed the best performance among all the algorithms. Compared with the CPU-based and GPU-based parallel algorithms based on state-of-the-art libraries, *HI-FFT* achieved up to 2.92 and 4.42 times higher performance. Also, *HI-FFT* showed up to 2.27 times higher performance than the prior heterogeneous algorithms while successfully handling a large-scale 2D-FFT problem the prior approaches failed to process.

With our in-place algorithm, we could handle a much larger 2D-FFT problem than the prior out-of-place algorithms. However, for a massive 2D-FFT problem whose input matrix size exceeds the system memory size, it is hard to apply our method. As future work, we plan to extend our method to support out-of-core processing to efficiently handle massive-scale 2D-FFT. From the perspective of processing performance, copying data between host and device memories is still the performance bottleneck of heterogeneous algorithms, although we have PCIe 4.0. Therefore, we would like to design an algorithm which minimizes data copy overhead for 2D-FFT computation on GPU.

REFERENCES

[1] A. V. Oppenheim, *Discrete-Time Signal Processing*. London, U.K.: Pearson, 1999.
[2] B. S. Reddy and B. N. Chatterji, "An FFT-based technique for translation, rotation, and scale-invariant image registration," *IEEE Trans. Image Process.*, vol. 5, no. 8, pp. 1266–1271, Aug. 1996.
[3] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through FFTs," 2013, *arXiv:1312.5851*. [Online]. Available: http://arxiv.org/abs/1312.5851
[4] P. Duhamel and M. Vetterli, "Fast Fourier transforms: A tutorial review and a state of the art," *Signal Process.*, vol. 19, no. 4, pp. 259–299, 1990.
[5] 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.
[6] Y. Zhang, F.-Z. He, N. Hou, and Y. M. Qiu, "Parallel ant colony optimization on multi-core SIMD CPUs," *Future Gener. Comput. Syst.*, vol. 79, pp. 473–487, May 2018.
[7] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Proc. Int. Conf. Acoust., Speech, Signal Process.*, vol. 3. May 1998, pp. 1381–1384.
[8] Intel. (2020). *Intel® Math Kernel Library*. [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html
[9] Y. Zhou, F. He, and Y. Qiu, "Dynamic strategy based parallel ant colony optimization on GPUs for TSPs," *Sci. China Inf. Sci.*, vol. 60, no. 6, Jun. 2017, Art. no. 068102.
[10] *CUDA Runtime API :: CUDA Toolkit Documentation*. Accessed: Jul. 15, 2021. [Online]. Available: https://docs.nvidia.com/cuda/archive/11.0/
[11] NVIDIA. *CUFFT Libraries*. Accessed: Jul. 15, 2020. [Online]. Available: https://docs.nvidia.com/cuda/cufft/index.html
[12] S. Mittal and J. S. Vetter, "A survey of CPU-GPU heterogeneous computing techniques," *ACM Comput. Surv.*, vol. 47, no. 4, pp. 1–35, Jul. 2015.
[13] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli, "State-of-the-art in heterogeneous computing," *Sci. Program.*, vol. 18, no. 1, pp. 1–33, Jan. 2010.
[14] D. Kim, J.-P. Heo, J. Huh, J. Kim, and S.-E. Yoon, "HPCCD: Hybrid parallel continuous collision detection using CPUs and GPUs," *Comput. Graph. Forum*, vol. 28, no. 7, pp. 1791–1800, Oct. 2009.
[15] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core CPU and GPU," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, Oct. 2011, pp. 78–88.
[16] D. Kim, J. Lee, J. Lee, I. Shin, J. Kim, and S.-E. Yoon, "Scheduling in heterogeneous computing environments for proximity queries," *IEEE Trans. Vis. Comput. Graphics*, vol. 19, no. 9, pp. 1513–1525, Sep. 2013.
[17] N. Hou, F. He, Y. Zhou, and Y. Chen, "An efficient GPU-based parallel tabu search algorithm for hardware/software co-design," *Frontiers Comput. Sci.*, vol. 14, no. 5, pp. 1–18, 2020.
[18] L. Gu, J. Siegel, and X. Li, "Using GPUs to compute large out-of-card FFTs," in *Proc. Int. Conf. Supercomput. (ICS)*, 2011, pp. 255–264.
[19] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka, "An efficient, model-based CPU-GPU heterogeneous FFT library," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, Apr. 2008, pp. 1–10.
[20] S. Chen and X. Li, "A hybrid GPU/CPU FFT library for large FFT problems," in *Proc. IEEE 32nd Int. Perform. Comput. Commun. Conf. (IPCCC)*, Dec. 2013, pp. 1–10.
[21] K. Underwood, K. Hemmert, and C. Ulmer, "Architectures and APIs: Assessing requirements for delivering FPGA performance to applications," in *Proc. ACM/IEEE SC Conf. (SC)*, Nov. 2006, p. 111.
[22] A. Saeed, M. Elbably, G. Abdelfadeel, and M. Eladawy, "Efficient fpga implementation of fft/ifft processor," *Int. J. circuits, Syst. Signal Process.*, vol. 3, no. 3, pp. 103–110, 2009.
[23] 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," *Microprocessors Microsyst.*, vol. 60, pp. 96–106, Jul. 2018.
[24] 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," in *Nanoelectronics, Circuits and Communication Systems*. Singapore: Springer, 2021, pp. 613–623.
[25] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," *ACM Trans. Algorithms*, vol. 8, no. 1, pp. 1–22, Jan. 2012.
[26] 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.
[27] M. Pippig, "PFFT: An extension of FFTW to massively parallel architectures," *SIAM J. Sci. Comput.*, vol. 35, no. 3, pp. C213–C236, Jan. 2013.
[28] 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*, vol. 6, pp. 64202–64224, 2018.
[29] K. Moreland and E. Ange, "The FFT on a GPU," in *Proc. SIGGRAPH/Eurographics Workshop Graph. Hardw.*, 2003, pp. 112–119.
[30] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," in *Proc. ACM SIGGRAPH Papers (SIGGRAPH)*, 2004, pp. 777–786.
[31] O. Fialka and M. Cadik, "FFT and convolution performance in image filtering on GPU," in *Proc. 10th Int. Conf. Inf. Visualisation (IV)*, 2006, pp. 609–614.
[32] Y. Chen, X. Cui, and H. Mei, "Large-scale FFT on GPU clusters," in *Proc. 24th ACM Int. Conf. Supercomput. (ICS)*, 2010, pp. 315–324.
[33] L. Gu, X. Li, and J. Siegel, "An empirically tuned 2D and 3D FFT library on CUDA GPU," in *Proc. 24th ACM Int. Conf. Supercomputing (ICS)*, 2010, pp. 305–314.
[34] A. Gholami, J. Hill, D. Malhotra, and G. Biros, "AccFFT: A library for distributed-memory FFT on CPU and GPU architectures," 2015, *arXiv:1506.07933*. [Online]. Available: http://arxiv.org/abs/1506.07933
[35] Z. Zhao and Y. Zhao, "The optimization of FFT algorithm based with parallel computing on GPU," in *Proc. IEEE 3rd Adv. Inf. Technol., Electron. Autom. Control Conf. (IAEAC)*, Oct. 2018, pp. 2003–2007.
[36] J. Wu and J. Jaja, "High performance FFT based Poisson solver on a CPU-GPU heterogeneous platform," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, May 2013, pp. 115–125.

[37] V. H. Naik and C. S. Kusur, "Analysis of performance enhancement on graphic processor based heterogeneous architecture: A CUDA and MATLAB experiment," in *Proc. Nat. Conf. Parallel Comput. Technol. (PARCOMPTECH)*, Feb. 2015, pp. 1–5.
[38] H. Khaleghzadeh, R. R. Manumachu, and A. Lastovetsky, "A novel data-partitioning algorithm for performance optimization of data-parallel applications on heterogeneous HPC platforms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 10, pp. 2176–2190, Oct. 2018.
[39] D. Takahashi, *Fast Fourier Transform Algorithms for Parallel Computers*. Springer, 2019.
[40] B. Van de Wiele, A. Vansteenkiste, B. Van Waeyenberge, L. Dupré, and D. De Zutter, "Fast Fourier transforms for the evaluation of convolution products: CPU versus GPU implementation," *Int. J. Numer. Model., Electron. Netw., Devices Fields*, vol. 27, no. 3, pp. 495–504, May 2014.
[41] M. J. Quinn, *Parallel Computing Theory and Practice*. New York, NY, USA: McGraw-Hill, 1994.
[42] U. Drepper, *What Every Programmer Should Know About Memory*, vol. 11. Raleigh, NC, USA: Red Hat, Inc., 2007.
[43] J. Cheng, M. Grossman, and T. McKercher, *Professional Cuda C Programming*. Hoboken, NJ, USA: Wiley, 2014.
[44] S. Asano, T. Maruyama, and Y. Yamaguchi, "Performance comparison of FPGA, GPU and CPU in image processing," in *Proc. Int. Conf. Field Program. Log. Appl.*, Aug. 2009, pp. 126–131.
[45] D.-K. Chen, H.-M. Su, and P.-C. Yew, "The impact of synchronization and granularity on parallel systems," *ACM SIGARCH Comput. Archit. News*, vol. 18, no. 2SI, pp. 239–248, Jun. 1990.
[46] D. Lustig and M. Martonosi, "Reducing GPU offload latency via fine-grained CPU-GPU synchronization," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2013, pp. 354–365.
[47] ARB. (2020). *OpenMP*. [Online]. Available: https://www.openmp.org/
[48] H. Zhang, L. Cao, and G. Jin, "Computer-generated hologram with occlusion effect using layer-based processing," *Appl. Opt.*, vol. 56, no. 13, p. F138, 2017.
[49] C. Slinger, C. Cameron, and M. Stanley, "Computer-generated holography as a generic display technology," *Computer*, vol. 38, no. 8, pp. 46–53, Aug. 2005.
[50] Y. Zhao, L. Cao, H. Zhang, D. Kong, and G. Jin, "Accurate calculation of computer-generated holograms using angular-spectrum layer-oriented method," *Opt. Exp.*, vol. 23, no. 20, 2015, Art. no. 25440.

HOMIN KANG is currently pursuing the M.S. degree with the School of Computer Engineering, Korea University of Technology and Education (KOREATECH). His research interests include heterogeneous parallel computing using CPU and GPU, and GPGPU computing for matrix operation, including 2D-FFT and matrix multiplication.

JAEHONG LEE is currently pursuing the M.S. degree with the School of Computer Engineering, Korea University of Technology and Education (KOREATECH). His research interests include high-performance computing and large-scale computer-generated holography.

DUKSU KIM (Member, IEEE) received the B.S. degree from Sungkyunkwan University, in 2008, and the Ph.D. degree in computer science from Korea Advanced Institute of Science and Technology (KAIST), in 2014. He spent several years as a Senior Researcher with KISTI National Supercomputing Center. He is currently an Assistant Professor with the School of Computer Engineering, Korea University of Technology and Education (KOREATECH). His research interests include designing heterogeneous parallel computing algorithms for various applications, including proximity computation, scientific visualization, and machine learning. He is a Young Professional Member of IEEE and a Professional Member of ACM. Some of his work received the Distinguished Paper Award at Pacific Graphics, in 2009, and an ACM Student Research Competition Award, in 2009. He was selected as the Spotlight Paper for the September Issue of IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS (TVCG), in 2013.