The Journal of Supercomputing
https://doi.org/10.1007/s11227-022-04570-9

Memory-accelerated parallel method for multidimensional fast fourier implementation on GPU

Yichang Hu[1] · Lu Lu[1] · Cuixu Li[2]

Accepted: 27 April 2022
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2022

Abstract

Fast Fourier transform (FFT) is a well-known algorithm that calculates the discrete Fourier transform (DFT) of discrete data and is an essential tool in scientific and engineering computation. Due to the large amounts of data, parallelly executing FFT in graphics processing unit (GPU) can effectively optimize the performance. Following this approach, FFTW and some other FFT packages were designed, but the fixed computation pattern makes it hard to utilize the computing power of GPU. Additionally, the memory access pattern is not optimized to alleviate the bottleneck of data exchange. Motivated by these challenges, we propose an efficient GPU-accelerated multidimensional FFT library to achieve better performance in this paper. We present a detailed and clear implementation strategy and optimize FFT by having as few memory transfers as possible. The data will be reshuffled on the CPU, and the access mode is also optimized to coordinate with the GPU memory access pattern. Several optimizations are also demonstrated to enhance the performance of our approach for varying FFT sizes, and the evaluation shows that our approach consistently outperforms rocFFT with a speedup of about 25% to 250% on average in AMD Instinct MI100 GPU.

Keywords FFT · DFT · GPU · ROCm · Memory optimization

✉ Lu Lu
lul@scut.edu.cn

Yichang Hu
sxy05100415@126.com

Cuixu Li
1827192140@qq.com

[1] School of Computer Science and Engineering, South China University of Technology, Guangzhou 510000, Guangdong, China

[2] Guangdong MeiWeiXian Flavoring Food Co., LTD., Zhongshan 528437, Guangdong, China

Published online: 02 June 2022

1 Introduction

DFT is essential in many scientific and engineering applications, including large-scale simulations [1], time series [2], waveform analysis [3], electronic structure calculations [4], and image processing [5]. Of particular interest has been FFT, a staple of mathematical computing and an efficient algorithm of DFT as it has been named one of the most important algorithms in the 20th century [6]. Numerically, it was first calculated relying on the DFT algorithm. However, due to its $O(N^2)$ computational complexity, FFT is used as a less time-consuming alternative for computing FTs in a divide-and-conquer fashion with an $O(N\log N)$ complexity.

Because of its wide range of applications, improving the performance of FFT is of great significance, and many efforts have been made from algorithm and hardware aspects. Plenty of optimized, production-level implementations of the FFT are developed across nearly all platforms, including FFTW [7], FFTPACK [8], cuFFT, and rocFFT. Nowadays, GPUs have become one of the most popular platforms for high-performance computing (HPC). Combining raw computation power with programmability, GPUs provide parallel architecture, which can execute programs on thousands of threads simultaneously and possess much larger bandwidth. Since the heterogeneous platform is gradually applied in a variety of fields, large amount of interest has been invested in optimized methodologies on the CPU platform [7], the GPU platform [9, 10], and other accelerator platforms [11].

Although previous works have focused on optimizing FFT with plenty of accelerator platforms, there are still following challenges for FFT optimization on GPU. The first challenge is efficiently utilizing the various types of memory on GPU to support FFT's special operations, because memory can easily become the bottleneck of the FFT algorithm due to its modest arithmetic intensity and distinct memory access pattern. To give full play to the computation performance of the Compute Unit (CU), the algorithm needs to be well-designed in data arrangement. Second, FFTs of different sizes use different kernels, and extensive sequence FFT kernels are more complicated. Therefore, a relatively consistent optimization effect on various and multidimensional FFT may not be gained.

Since there is still considerable room for improvement in the proposed work, we introduce our optimization of FFT on GPU in this paper to deal with the challenges of FFT. The central issues are the utilization of GPU and the memory transfer between threads. We propose a template-based code generation framework that can generate high performance FFT codes automatically on GPU to process FFT of various sizes and dimensions. In addition, the way of data access has been optimized and a novel way based on FFT matrix form is also demonstrated. This work makes the following three primary, novel contributions in optimizing FFT algorithms for efficient execution on GPU:

- A novel template-based FFT library is developed, generating assembly FFT kernels automatically, to accelerate the algorithm on GPU with high performance for multidimensional and mixed radices sequences. The data access

mode is optimized to non-strided layout, so that GPU memory can read the data continuously and less transposition operation is required.
- We exploit several major techniques to further improve its performance. First, a special model for radix-2 FFT based on the matrix form is designed to utilize the computing power of 16×16 GEMM in the CDNA's Matrix Cores. Besides, some other optimized memory mechanism, including in-place and coalesced memory access pattern, is proposed to alleviate the performance loss.
- Our work is evaluated on the AMD instinct GPU. On MI100, it achieves about 2x speedup on multidimensional, mixed radices FFTs over rocFFT in general.

In the remainder of this paper, Sect. 2 introduces FFT algorithm briefly, and Sect. 3 describes some existing related work. Then, in Sect. 4., we propose our novel FFT library and introduce the architecture in detail. Next, some optimization methodologies are presented in Sect. 5. In Sect. 6, we evaluate the library on a MI100 GPU to demonstrate the superiority of our implementation. Finally, we offer the conclusion and discuss the future work in Sect. 7.

2 FFT algorithm

2.1 General FFT algorithm

FFT is an efficient way of computing DFT for a set of signals. DFT converts a finite sequence of samples from the time or space domain to the frequency domain and vice versa. The forward transform is expressed as:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk} \tag{1}$$

where N is the input length, n is the spatial index, and k is the frequency index.

The DFT incurs a high computation cost with an asymptotic complexity of $O(N^2)$. In cases where N is significantly large, direct computation affects the performance of the system adversely. A faster approach, known as FFT, introduced by Cooley and Tukey, reduces this complexity to $O(N \log N)$. The efficient algorithm proposed in 1965 for computing DFT was a significant turning point in the development of digital signal processing [12]. The FFT butterfly operation, which is the fundamental calculation element in the FFT process, takes two complex points and converts them into two other complex points. In the 2-point (radix-2) Cooley-Tukey algorithm, the butterfly is simply a DFT of size-2 that takes two complex inputs (x_0, x_1) and gives two complex outputs (y_0, y_1) by using Eqs. 2 and 3. The twiddle factors in FFT algorithms are trigonometric coefficients that multiply the data [13, 14].

$$y_0 = x_0 + x_1 w^k \tag{2}$$

$$y_1 = x_0 - x_1 w^k \tag{3}$$

where the twiddle factor, w^k, is given as:

$$w^k = e^{\frac{-2\pi i}{N}nk} = \cos\left(\frac{2\pi}{N}nk\right) - i\sin\left(\frac{2\pi}{N}nk\right) \tag{4}$$

And for a radix-2, the DFT from Eq. 1 can be rewritten as:

$$X_k = \sum_{n=0}^{N/2-1} x_{2n} w_N^{2nk} + w^k \sum_{n=0}^{N/2-1} x_{2n+1} w_N^{2nk} \tag{5}$$

2.2 FFT in matrix form

The complete FFT algorithm consists of multiple processes of combining N_1 subsequences to get the DFT of the original N-point sequence. This process can be expressed in the form of a matrix as Eq. 6. It is rewritten from Eq. 1 according to the periodicity of W^k.

$$X_{out} = F_{N_1} \cdot (T_{N_1 N_2} \odot X_{in}) \tag{6}$$

where $\cdot$ denotes matrix product, $\odot$ denotes element-wise product, and N_2 equals N/N_1. X_{in} is the input sequence as follows:

$$X_{in} = \begin{bmatrix} X_0[0] & X_0[1] & \cdots & X_0[N_2-1] \\ X_1[0] & X_1[1] & \cdots & X_1[N_2-1] \\ \vdots & \vdots & \vdots & \vdots \\ X_{N_1-1}[0] & X_{N_1-1}[1] & \cdots & X_{N_1-1}[N_2-1] \end{bmatrix} \tag{7}$$

X_{out} represents the matrix form of the output N-point DFT of the original sequence. The matrix is $N_1 \times N_2$ and is shown as follows:

$$X_{out} = \begin{bmatrix} X[0] & X[1] & \cdots & X[N_2-1] \\ X[N_2] & X[N_2+1] & \cdots & X[2N_2-1] \\ \vdots & \vdots & \vdots & \vdots \\ X[(N_1-1)N_2] & X[(N_1-1)N_2+1] & \cdots & X[N_1N_2-1] \end{bmatrix} \tag{8}$$

F_{N_1} denotes the radix-N_1 DFT matrix. $T_{N_1N_2}$ is an $N_1 \times N_2$ twiddle factor matrix, and it has the same size as X_{out} and X_{in}. F_{N_1} and $T_{N_1N_2}$ are shown as follows:

$$F_{N_1} = \begin{bmatrix} W_{N_1}^0 & W_{N_1}^0 & \cdots & W_{N_1}^0 \\ W_{N_1}^0 & W_{N_1}^1 & \cdots & W_{N_1}^{N_1-1} \\ \vdots & \vdots & \vdots & \vdots \\ W_{N_1}^0 & W_{N_1}^{N_1-1} & \vdots & W_{N_1}^{(N_1-1)(N_1-1)} \end{bmatrix} \tag{9}$$

$$T_{N_1 \times N_2} = \begin{bmatrix} W_N^0 & W_N^0 & \cdots & W_N^0 \\ W_N^0 & W_N^1 & \cdots & W_N^{N_2-1} \\ \vdots & \vdots & \vdots & \vdots \\ W_N^0 & W_N^{N_1-1} & \vdots & W_N^{(N_1-1)(N_2-1)}, \end{bmatrix} \tag{10}$$

where X_m denotes the DFT of the m-th $(N_2 - 1)$-point sampling subsequence of original $x \cdot x_m[n] = x[nN_1 + m]$, for all $0 \leqslant n \leqslant N_2 - 1$. X_m is calculated in the previous iteration.

The above content describes how the radix-N_2 FFTs are converted into the FFT of a long sequence through matrix multiplication and element-wise multiplication [15]. And the FFT of the original sequence can be obtained through this process of $\log_{N_1} N$ times.

3 Related work

Methods of FFT acceleration have been widely explored and proposed over the last decades on CPU, GPU, and other accelerator platforms [16, 17]. Yasuhito et al. [18] propose a model-based, adaptive library for 2D FFT that automatically achieves optimal performance using available heterogeneous CPU-GPU computing resources to overcome the problem that the GPU performance can be severely limited by such restrictions as memory size and bandwidth and programming using graphics-specific APIs. Cilasun et al. [19] exploit acceleration opportunities for high-resolution FFTs in spintronic computational RAM (CRAM), which supports proper in-memory processing semantics. Xiaowen Chen [20] et al. propose a variable-size FFT hardware accelerator based on matrix transposition (MT), which can efficiently divide a large size FFT into several small size FFTs that can be executed in parallel. Apart from this, several optimization techniques such as hybrid twiddle factor generation, multibank data memory, block MT, and token-based task scheduling are also proposed. AutoFFT [21] is a code generator that can automatically generate fast Fourier transform (FFT) codes for ARM and x86 CPUs. To further reduce the floating-point butterfly operations, Zhihao Li et. explore more symmetric and periodic properties of the DFT matrix and formulate two optimized calculation templates for prime and power-of-two radices. The template-based auto-tuning mechanism has been verified to be efficient in CPU and is referenceable for the expansibility to GPU.

Apart from the FFT library accelerator, parallel distributed-memory multidimensional FFT is also placed great emphasis [22]. Chen et al. [23] introduce a flexible partitioning scheme that enables concurrent execution of CPU and GPU and integrates several FFT decomposition paradigms to tailor computation and communication. Amir Gholami [24] et al. extend existing FFT libraries for Compute Unified Device Architecture (CUDA) GPUs to distributed memory clusters. They use overlapping communication methods to reduce the overhead of PCIe transfers from/to GPU. To reduce the communication to approximately a single all-to-all transpose, Cris Cecka [25] reformulates an existing algorithm that employs the fast multipole method (FMM).

The requirement for mixed-precision FFT also enhances, while half-precision (FP16) is gaining popularity due to its faster speed and energy saving ability [26]. Moreover, several approaches have been proposed for using Tensor Cores in the computation of FFT. Sorna [27] and Cheng [28] show the theoretical basis and an example implementation that resorted to cuBlas to utilize Tensor Cores. In Durran's poster [29], their implementation outperformed cuFFT with Tensor Core WMMA APIs, but only on the primary small size 1D FFT. These approaches did not handle the memory bottleneck caused by the distinct memory access pattern of large size or multidimensional FFT. These researchers are absorbed in the mixed-precision FFT, which is frequently used in deep learning [30], image restoration, etc. The loss of precision may be intolerable in some HPC applications, but their methodologies are referenceable.

The previous work still has some great challenges, including efficiently utilizing the various types of memory on GPU to support FFT's special operations and complicated kernel design for sequence FFT kernels. Therefore, a relatively consistent optimization effect on various and multidimensional FFT may not be gained. In the light of these challenges, several major techniques are exploited to further improve its performance. The data access mode is optimized to non-strided layout, so that GPU memory can read the data continuously and less transposition operation is required. And a special model for radix-2 FFT based on the matrix form is designed to utilize the computing power of 16×16 GEMM in the CDNA's Matrix Cores.

4 Implementation of FFT computation on GPU

4.1 GPU architecture

GPU can achieve extremely high computational throughput by employing many cores working on a large set of data in parallel. Radeon Open Computing platform (ROCm), developed by AMD, is the open-source software development platform for HPC/Hyperscale-class GPU computing. ROCm is aimed to be an alternative of CUDA and is gradually used programming approach in massively parallel computing applications. ROCm's Heterogenous-compute Interface for Portability (HIP) allows coding in a single-source C++ programming language including features such as templates, C++11 lambdas, classed, namespaces. HIP provides a simple path for users familiar with programming language to develop programs to be executed by GPUs easily.

The AMD GPU architecture consists of multiple compute units (CU) with pipelined cores and instruction dispatch units. The command processor and scheduling logic translate higher-level API commands into compute tasks. Conversely, these computing tasks are implemented as compute arrays and managed by the Asynchronous Compute Engines (ACE). Although HIP provides the possibility to unleash GPU's computational power, several restrictions prevent programmers from achieving peak performance.

CDNA is AMD's latest architecture, and the Instinct MI100 accelerator based on CDNA is the world's fastest HPC GPU. MI100 is the first GPU to break the

10TFLOP/s double precision (FP64) barrier designed as the stepping stone to the next generation of Exascale systems. As Fig. 1 illustrates, 120 CUs of the CDNA architecture are organized into four arrays. They are derived from the earlier GCN architecture and execute wavefronts that contain 64 work-items. However, the CUs are enhanced with new Matrix Core Engines optimized for operating on matrix datatypes, boosting compute throughput and power efficiency. The execution unit reduces the number of register file reads, since many input values are re-used in a matrix multiplication.

As we mentioned in Sect. 2, the FFT algorithm combines radix-N_2 subsequences to get the DFT of the original N-point sequence. Since Matrix Cores of CDNA can provide impressive computing power for matrix datatypes, which can be used in power-of-16 radices and prominently reduce the execution time of FFT.

4.2 Architecture overview of FFT implementation on GPU

In the GPU-based parallel computing, hardware architecture is critical while designing FFT computation algorithm to achieve peak performance. In the algorithm design, there are some major points, including number of stages in butterfly operations, calculation of twiddle factors, and batch size of FFTs that will be computed in parallel. The memory architecture that keeps the intermediate values during the computation is likewise crucial to the time of data exchange [31].

Coalesced global memory has a immense impact on memory-bound applications on heterogeneous systems. In order to achieve higher throughput, the data may be ordered to guarantee coalesced 64 or 126 bit reads. Moreover, FFT input sequence is a 1D array of type *double2* that combines two floating-point elements for the real and complex parts. (This work mainly focuses on the Complex to Complex (C2C) FFT, where the number of elements is $2 \times \mathrm{N}$.)

In the general algorithm, each multidimensional FFT is decomposed into the set of 1D FFTs, which is the typical approach to the algorithm. Our work also follows this rule, and each 1D sequence from the input is uploaded to the shared memory separately. Then, the butterfly operations are performed there entirely. The design has two advantages compared with the previous methods. First, since the access speed of global memory of GPU is relatively slow, shared memory has better performance, and the full

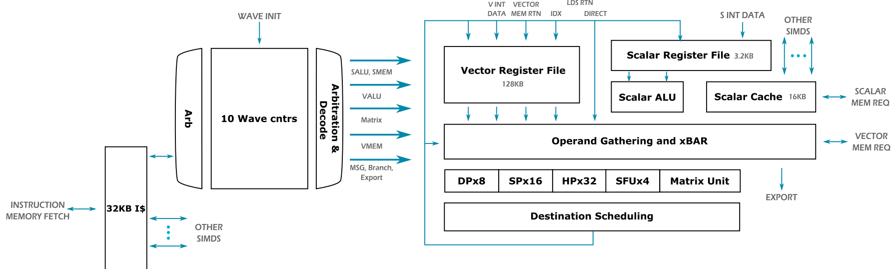


Fig. 1 Block Diagram of an Enhanced Compute Unit (CU) with a detailed SIMD view of the AMD CDNA architecture

utilization of it can improve the evidently. Besides, only one read and write is required to the on-chip memory per axis for butterfly operations.

As we mentioned in Sect. 2, a radix-2 N-point FFT process is performed as a $\log N$ stage computation module, where a 2-point FFT is computed in each stage. Most libraries use $N/2$ independent threads in GPU to achieve the size of N, and the threads are communicated with each other over the shared memory. After the butterfly operation, each thread writes 2 complex values to the shared memory with non-contiguous locations which is calculated by thread number and current stage number. For the next stage, the threads read two complex values with contiguous locations from the shared memory to obtain coalesced access pattern. Figure 2 is the demonstration of 8-point 1D-FFT computation including shared memory usage with 4 independent threads.

Considering the memory access mode of GPU, we optimize the departed implementation in our work. The memory accesses are non-strided so that 32 nearest values are uploaded to GPU at once. However, for higher-dimensional FFT, where the data are strided in directions (y, z), transposition is needed and takes roughly equal to one read and write. And if a sequence is small (a length of ≤ 256), device memory reads 16 nearby complex numbers and does not transpose the data matrix. This design can accelerate the algorithm in the case of small FFTs, benefit from the non-strided access and less data exchange. During 1D-FFT computation along one axis, each line in that axis is considered independent. Threads can be identified using a two-dimensional thread index by a thread block, a two-dimensional block of threads. The second dimension of the thread block, which defines the number of blocks, can be used to implement the other lines of one axis of 3D-FFT computation. The number of threads per block is $N/8$ instead of $N/2$ for an N FFT computation in our design because 16 FFTs instead of 2 FFTs are performed in one thread.

Blocks of the same dimension are computed in parallel by the threads. Then, the threads repeat the exact computation with new indexes N more times to finish all 1D-FFT computations along one axis. The threads write the results of each 1D-FFT computation to global memory so that new FFT kernels will be started in the next dimension. For the next dimension, the start index will be calculated by thread ids for each thread. To complete all the computation, all 1D-FFT computations along three dimensions are executed

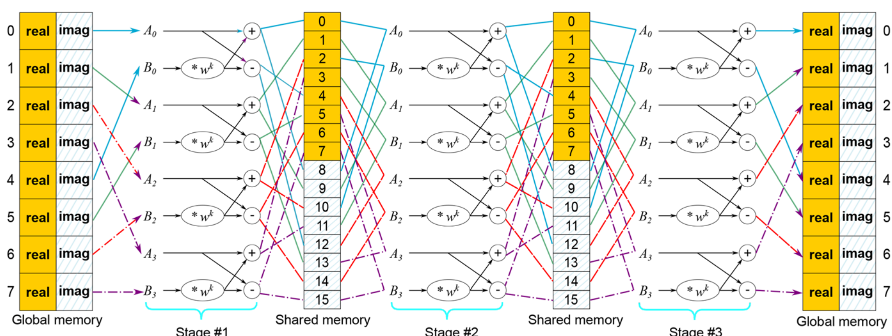


Fig. 2 8-point FFT architecture with 3 stages (Note that; real: real part(solid), imag: imaginary part(stripe)

by $N \times N/8$ threads with transpose illusion which is obtained by indexing the strided sequence and copying data from global memory to share memory.

Moreover, computing twiddle factors by __sinf(x) and __cosf(x) HIP math library functions are slower than reading precalculated values from the host memory in most case. So that, twiddle factors are pre-computed and copied to the device in our work.

4.3 FFT kernel generation

A computational template is designed by extracting the formalized expressions of the patterns to adopt the optimized data access mode in the process of code generation. The designer constructs computational templates, composed of fundamental butterfly templates and integrated templates.

Based on the computational templates, an FFT code generator is designed to autogenerate high-performance FFT kernels for mixed radices. And runtime compilation mechanism is employed. *hipRTC* APIs accept HIP source files as input parameters and create handles of programs by runtime compilation. FFT kernels are compiled by *hipRTC* APIs, providing the possibility of performance improvement compared with regular offline static compilation.

5 Performance optimization

In this section, some optimization methods will be introduced toward the baseline FFT implementation in the last section.

5.1 Incorporating methodologies for matrix-based FFT

As we mentioned in Sect. 2.2, the FFT algorithm combines radix-N_1 subsequences to get the DFT of the original N-point sequence. Since Matrix Cores in CDNA architecture only provide computing power for $16 \times 16 \times 16$ GEMM, which can be used in power-of-16 radices, this part of work supports only radix-2 FFT of large sequences.

Modeled after rocFFT, our implementation uses a universal configuration called a plan. A plan chooses a series of optimal radix-N_1 kernels. Then, when the execution function is called, the actual transform takes place following the plan [29].

Figure 3 shows the complete process of a radix-512 FFT. First, a plan is created according to the dimension and the size of the input data. This plan selects an optimal set of all kernels from the pre-implemented incorporating kernel collection. Then, the execution function is called with the plan and the original data are uploaded to GPU. In the execution function, the previously determined incorporating kernels are executed in turn in multiple iterations. Meantime, multiple types of memory in GPU are fully utilized to increase the reuse of data. After all iterations are executed, the FFT of the original sequence is written to global memory. The reusability of code is improved, and the workload of subsequence performance-related optimization work is also greatly reduced by decomposing a large FFT sequence into a series of incorporating processes.

Algorithm 1: 1D Radix-256 FFT kernels

Input: 256 FFTs of size N
Output: FFTs of size $256N$

```
1  F_{16×16} ← UpLoad Radix-16 DFT Matrix;
2  Tw_{16×N} ← Twiddle Factors are pre-computedl;
3  Take the input data as 16 Matrix_{16×N} matrices;
4  foreach Matrix_{16×N} do
5      foreach InFrag_{16×16} in Matrix_{16×N} do
           /* Parallel by Warps */
6          TwFrag_{16×16} ← load a fragment from the twiddle factors;
7          InFrag_{16×16} = InFrag_{16×16} ⊙ TwFrag_{16×16};
8          OutFrag_{16×16} = F_{16×16} · InFrag_{16×16};
9          Store OutFrag_{16×16} to intermediate data;
10     end
11 end
12 Tw_{16×16N} ← Prepare twiddle factors;
13 Take the intermediate data as Matrix_{16×16N} matrices;
14 foreach InFrag_{16×16} in Matrix_{16×16N} do
       /* Parallel by Warps */
15     TwFrag_{16×16} ← load a fragment from the twiddle factors;
16     InFrag_{16×16} = InFrag_{16×16} ⊙ TwFrag_{16×16};
17     OutFrag_{16×16} = F_{16×16} · InFrag_{16×16};
18     Store OutFrag_{16×16} to output data;
19 end
```

Multidimensional FFT performs 1D FFT on each dimension of sequences in turn and can be implemented by batched FFT. Take a 2D FFT on an $N_1 \times N_2$ row-major matrix as an example, an N_1 batch of N_2-point FFT is applied on the N_1 rows, and then, an N_2 batch of strided N_1-point FFT is applied on the N_1 columns. Therefore, the 2D FFT can be implemented by calling it with proper parameters.

Algorithm 1 shows the radix-256 incorporating kernels, and there are 256 FFT sub-sequences of size N_2 in a large sequence. The kernels contain three sub-incorporating processes. Line 1-11 and line 12-19 are two radix-16 sub-kernels, which is the base of this part of work, because a 16×16 matrix can fill a Matrix Core fragment, significantly increasing the highest computational efficiency.

We treat the X_{out} matrix and the X_{in} matrix as multiple matrices size of 16×16, which will be calculated in parallel threads. A sub-kernel can execute a radix-16 incorporating process described by Eq. 6 in Sect. 2.2. The incorporating process combines FFTs of 16 sequences into an FFT of a sequence of 16 times the length. As shown in line 1-11 in algorithm 1, the sub-kernel firstly loads the radix-16 DFT matrix and calculates twiddle factors while reading input. These matrices are divided into 16×16 fragments and distributed to the GPU threads for parallelization. Then, threads execute matrix multiplication with Matrix Cores and element-wise

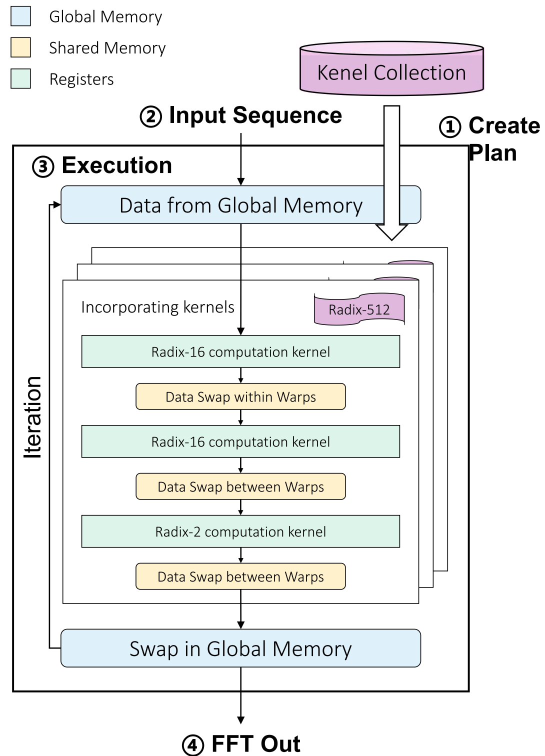


Fig. 3 The complete execution process of our implementation

multiplication on these fragments. After that, all fragments are put together, and 256 FFT sequences of size N are arranged into 16 FFT sequences of size $16N$.

The incorporating strategy by Matrix Core can make global memory accesses performed only when necessary, and only unavoidable synchronizations are performed. In addition, this strategy achieves the purpose of reducing bandwidth requirements and increasing arithmetic intensity.

5.2 Alleviate the memory bottleneck

There are two reasons that memory can easily become the bottleneck of FFT algorithms on GPU. First, the original algorithm requires $\log N$ times of global memory accesses, but the arithmetic intensity of one iteration is relatively tiny. Second, multidimensional FFT needs strided memory access, which will be pretty inefficient on GPUs. For the first problem, we combine 16 FFTs at once, as we mentioned in Sect. 4. The times of global memory accesses will be reduced, and the arithmetic intensity will be increased simultaneously. For the second problem, the data will be reshuffled after the four-step FFT algorithm with an additional buffer (for big sequences). This design and the data layout pattern will be explained in detail in the remainder of this section.

Four-step FFT algorithm. All conventional FFT algorithms require at least n passes through the data set to compute an FFT. The four-step FFT algorithm reduces the number of passes to three.

The four-step algorithm was first presented in a paper by Gentleman and Sande [13], who discussed its implementation on computer systems with hierarchical memory. This algorithm, which shall hereafter be referred to as the four-step FFT algorithm, can be stated very succinctly. Let $n = n_1 n_2$ be the size of the transform. Note that n does not necessarily need to be a power of two. On many systems, the implementation of this algorithm is most efficient when n_1 and n_2 are as close as possible to $\sqrt{n}$. In the following and hereafter, matrices will be assumed to be stored in memory column-wise. The FFT of n complex input data values can then be obtained by performing the following four steps:

(1) Perform n_1 simultaneous n_2-point FFTs on the input data considered as a $n_1 \times n_2$ complex matrix.
(2) Multiply the resulting data, considered as a $n_1 \times n_2$ matrix A_{jk}, by $e^{\pm 2\pi ijk/n}$. The $\pm$ sign is the sign of the transform.
(3) Transpose the resulting $n_1 \times n_2$ complex matrix into a $n_2 \times n_1$ matrix.
(4) Perform n_2 simultaneous n_1-point FFTs on the resulting $n_2 \times n_1$ matrix.

The attractive features of the four-step FFT algorithm can be summarized:

- The four-step algorithm produces an ordered transform vector - it is not necessary to perform a bit reversal permutation, which would lead to performance degradation on many advanced computer systems.
- It is easier to exploit the underlying topology in a distributed transpose than in a distributed butterfly.
- There is less idle waiting time in a distributed transpose than in a sequence of distributed butterflies, where the computation is repeatedly interrupted by communication operation.

All those factors impel us to employ the four-step FFT algorithm in our work. And data will be reshuffled during the algorithm with an additional buffer when the input sequence is large and all transformations are performed in-place. Under that circumstance, there are no additional transposition uploads and no performance loss. Figure 4a shows a example of an 8-point FFT, where the radix is 2. A pair of output elements is stored with a stride length 4, and the uncontinuous data pattern requires twice the size of shared memory, limiting the GPU performance. The data are stored in a improved order, and Fig. 4b shows the rearrangement of the 8-point according to parity, allowing this process to be executed in place with no performance loss.

Data Access Mode. When the data sequence is large, the number of threads that access global memory will increase [32]. Nevertheless, the storage space of share memory is limited. Some corresponding changes are made based on the GPU execution to reach optimum performance when visiting and fetching data.

First, the program divides the data into several pieces and then chooses the appropriate number of threads in one block. Second, the program puts the data in different pieces into shared memory, and the butterfly operation is done in each block. At last, the data in shared memory are put back into the global memory. This is a data exchange. And if the data sequence is bigger, more times exchange will be performed. The differences of the butterfly operation in each level are the span and twiddle factor in the algorithm. So we break the data into pieces to make its span small, and every piece is suitable for computing in the shared memory.

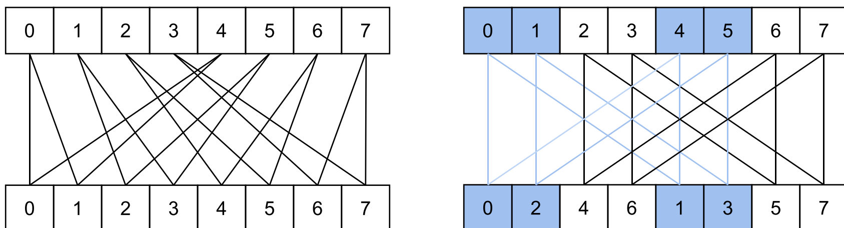
Furthermore, a strategy aimed at the sequence that is full of zeros is also applied in our work. The strategy can omit the sequence-filled zeros and not upload additional memory. When the data have been uploaded to the device memory, zeros will be padded automatically. As we can be seen in Fig. 5, it can specify the range of sequences filled with zeros and the direction when the data are read or written.

6 Evaluation

6.1 Experimental setup

In this section, we measure the performance of our FFT implementation on a heterogeneous system which equipped AMD Instinct MI100 GPU. We compare our work with AMD rocFFT, which is the state-of-the-art FFT library on AMD GPU. The version of ROCm we used is 4.2.0 released. As of the time of this writing, it is the latest version on our testing MI100 platform. We compare them in terms of performance, accuracy, and bandwidth. The performance of multidimensional, mixed-radices FFTs of adequate sizes on MI100 GPU is shown in this section to evaluate the generalization of our algorithm.

We measure the performance of our optimizations on the heterogeneous platform, and the specific configuration is shown in Table 1.



(a) Original out place with a fixed data order

(b) Our work with a improved data order: two adjacent butterflies are grouped

Fig. 4 In place and coalesced memory access pattern: a radix-2 of 8-point sequence as an example

The data are first transferred from CPU to GPU in the performance tests, and a plan is created. Then, the execute function is executed thousands of times (perform one FFT cannot make full use of GPU), and the average performance is reported. The data transfer time between the host and device memory is limited by PCI bus bandwidth. It is independent of the performance of code running time on GPU, so the transfer time is excluded from our experimental, and only the GPU's running time is evaluated. For the input length of N FFT with execution time of t seconds, its performance in GFlops is defined as

$$GFlops = \frac{5 \times N^3 \times \log N^3}{t} \times 10^{-9} \tag{11}$$

As for the precision metric, the result after one FFT and inverse FFT (iFFT) will be compared with the input sequence as Eq. 12.

$$Error = \max_{0 \leqslant i \leqslant N} \sqrt{X_{in}[i]^2 + X_{out}[i]^2} \tag{12}$$

6.2 Speedup

Figure 6 shows the performance comparison between our work and rocFFT on MI100. In the small sequence cases, the execution time on GPU is not long enough, and the calculations cannot be overlapped with memory access. Thus, the performance is mainly limited by the global memory bandwidth. As the FFT size increases, the optimizations will promote the performance more obviously, because the large buffer size will lead to abundant memory access in rocFFT, which causes a longer execution time. In all cases, our work can achieve an average 1.71 × speedup for radix-2 and an average 2.78x speedup for radix-3,5,7 and mixed radices compared with rocFFT.

As we can be seen from Fig. 6, when the FFT size is approximate, our work and rocFFT both perform better under radix-2 than radix-3,5,7 and mixed radices. One reason is that the number of threads and blocks is power-of-two, which can be adapted to the calculation mode of GPU. Another reason is the incorporating methodology introduced in Sect. 5 can fully optimize the utilization of GPU computing power. Radix-2 FFTs are also optimized in rocFFT, because radix-2 FFTs are the typical usage scenario in HPC applications. Our work generates optimized code at

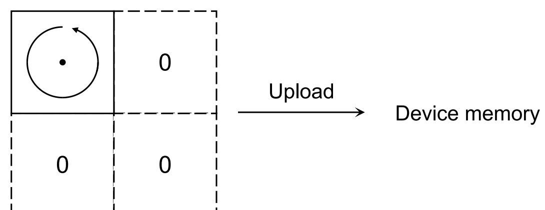


Fig. 5 Cutting off the sequence full of zeros and uploading less memory

Table 1 Platform information

Platform	
CPU	Dual 16-core Intel Xeon Gold 6142 CPUs @2.60 GHz
Host Memory	256GB
GPU	Quad AMD Instinct MI100 GPUs, 32GB RAM
PCI-E	PCI-E Gen3 x16
ROCm	ROCm 4.2.0

run time, and memory bottleneck can be alleviated more notably under the radix-3,5,7 and mixed radices, which are not improved in rocFFT specifically.

In the same experimental condition, Fig. 7 shows the performance comparison of 3D FFT between our work and rocFFT. The same conclusion is obtained with 2D FFT in Fig. 6. Our work and rocFFT both perform better under radix-2 than radix-3,5,7 and mixed radices. The level of improvement is 1.48x and 2.06x for radix-2 and other radices, respectively, because radix-2 FFTs are optimized in rocFFT.

Besides, there are some other interesting discoveries for radix-2. Our work will outperform rocFFT greatly when X, Y, Z are unequal. The first two dimensions (X and Y) are calculated first, the data are strided in directions (y, z), so for the best performance, order dimensions in descendant size may be ordered as $X > Y > Z$ in our work. Nevertheless, when the three dimensions are the same, $X = Y = Z$ is the best case in rocFFT. For example, in the last slide, $512 \times 256 \times 128$ takes much more time than $256 \times 256 \times 256$ in rocFFT (the input size is the same 2^{24}). But in our work, the opposite is the case, and the difference is not great. The reason is that an FFT plan with different X, Y, Z will be computed in different ways in rocFFT. When the X, Y, Z are unequal, rocFFT will launch the kernel functions called *transpose_kernel_scheme*, but not when $X = Y = Z$. The *transpose_kernel_scheme* function is used to handle tiled (row/column) + strided + batched Stockham kernels for arbitrary factorizations. Different X, Y, Z determines how the sequence will be broken down into GPU thread blocks. Therefore, the performance of the same input size (different X, Y, Z) varies too much. But in our work, the transpose scheme is optimized due to the access mode introduced in Sect. 5.

The above results show that although the improvement of our work is not significant in radix-2 cases, our work can outperform rocFFT by a notable margin across different dimensions in other radices cases.

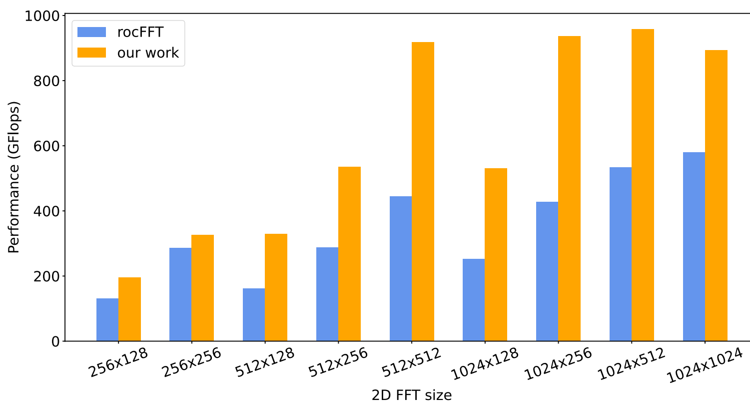
6.3 Precision

Figure 8 shows the average relative error comparison between our work and rocFFT in radix-2 cases. Despite the different GPU kernels, the error of the two libraries is at the same level in FP64. Our library's related error is slightly smaller than rocFFT in a large-scale sequence, demonstrating the high applicability in the HPC applications.

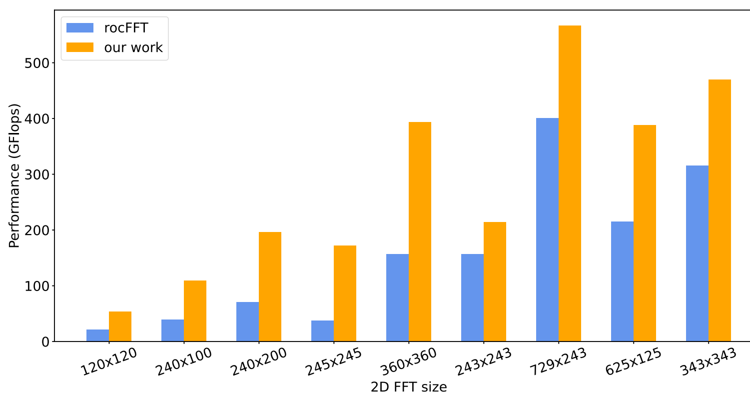
6.4 Memory throughput

Figure 9 is the memory throughput comparison between our work and rocFFT in FP64 on MI100. The experiment configuration below takes multiple 1D FFTs of a supported sequence length from the range of 2 to 4096 and batch them together, so about 1GB of data is uploaded, and multiple consecutive FFTs (different from the experiment in 6.2, which is the single FFT for multiple executions) are performed. After that, the time which one single FFT takes is obtained by averaging the result. Total data size will be divided by the time taken through one upload and download, resulting in the achieved bandwidth.

As we can be seen from Fig. 9, the bandwidth of our work is nearly 2x rocFFT in general, promoted by the memory access pattern and the data arrangement. Only in some cases of radix-2, the performance of rocFFT is close to our work.



(a) 2D radix-2 on MI100

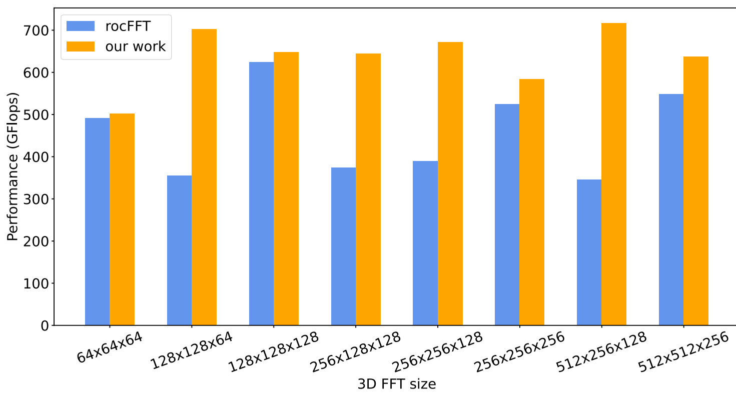


(b) 2D other radices on MI100

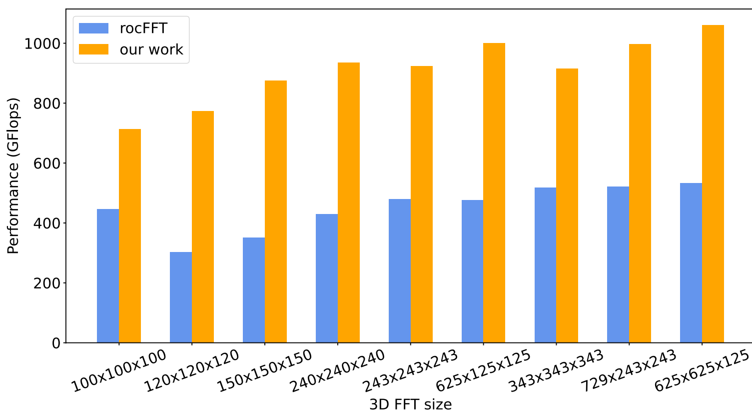
Fig. 6 The performance of different sizes 2D FFT

7 Conclusion

This paper presents the design and implementation of a GPU-based FFT algorithm for mixed radices and multidimensions. Furthermore, we exploit a set of optimizations to support the butterfly operations of FFT efficiently and the data access mode to alleviate the memory bottleneck. Our work is evaluated on the AMD instinct GPU. We measure up to 2x performance increase over rocFFT on MI100 in general, achieving about 1000 GFlops. The precision and bandwidth are also evaluated to demonstrate the great potential to use the HPC GPU to accelerate FFT. The methodologies can be generalized to other FFT scenarios (like R2C and C2R) and can be highly portable across architectures. Despite the novelties of our work, several potential drawbacks may occur, like excessive time of kernel generation on CPU and no further optimization of data transmission between host and device.



(a) 3D radix-2 on MI100



(b) 3D other radices on MI100

Fig. 7 The performance of different sizes 3D FFT

Future work includes exploiting additional optimization of our library to further reduce memory requirements and applying it to some HPC applications. Besides, follow-up efforts can be made to extend the library to multiple nodes and other FFT scenarios (R2C, C2R, etc).

Acknowledgements This work was supported in part by the Major Project on the Integration of Industry, Education and Research of Zhongshan under Grant 210602103890051.

Data availability Data sharing not applicable to this article as no datasets were generated or analyzed during the current study, and source data are provided with the paper in Figs. 6, 7, 8, 9.

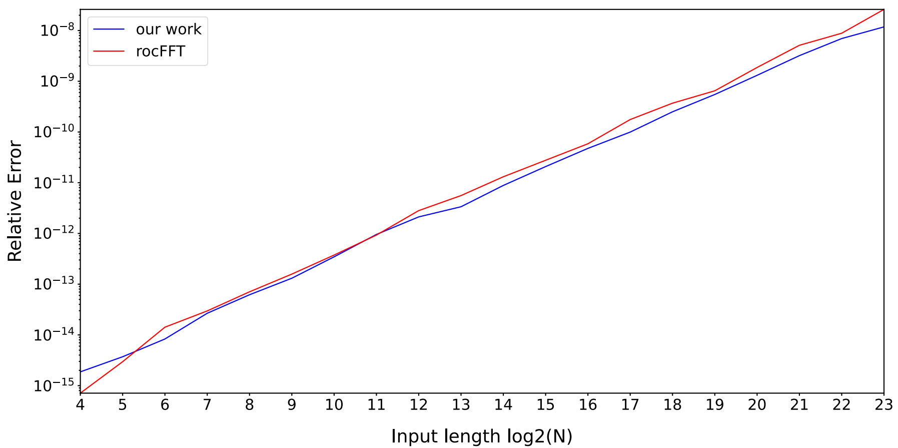


Fig. 8 Average relative error after one FFT and iFFT, comparison between our work and rocFFT (double precision)

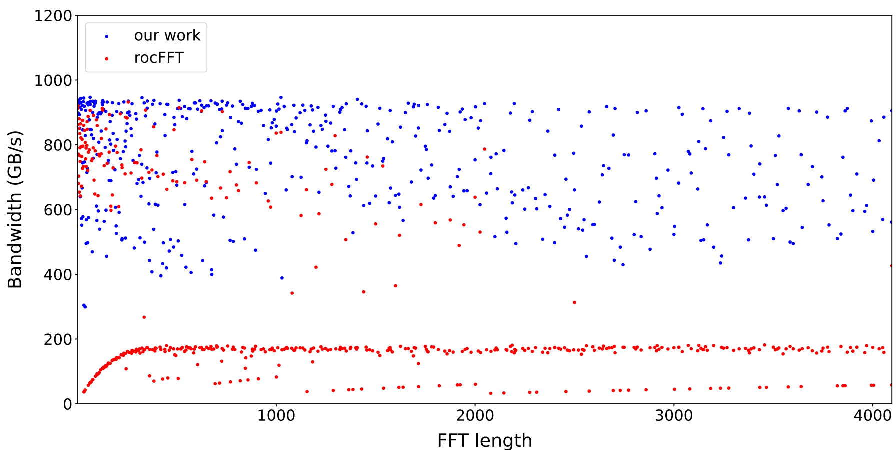


Fig. 9 Bandwidth comparison of batched 1D FFTs in double precision

References

1. Cheng S, Yu H-R, Inman D, Liao Q, Wu Q, Lin J (2020) Cube-towards an optimal scaling of cosmological n-body simulations. In: 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), pp 685–690. https://doi.org/10.1109/CCGrid49817.2020.00-22
2. Watson W, Spedding TA (1982) The time series modelling of non-gaussian engineering processes. Wear 83(2):215–231. https://doi.org/10.1016/0043-1648(82)90178-8
3. Biwer CM, Capano CD, De S, Cabero M, Brown DA, Nitz AH, Raymond V (2019) PyCBC inference: a python-based parameter estimation toolkit for compact binary coalescence signals. Science 131(996):024503. https://doi.org/10.1088/1538-3873/aaef0b
4. Haynes PD, Côté M (2000) Parallel fast fourier transforms for electronic structure calculations. Comput Phys Commun 130(1):130–136. https://doi.org/10.1016/S0010-4655(00)00049-7
5. Després P, Jia X (2017) A review of gpu-based medical image reconstruction. Physica Med 42:76–92. https://doi.org/10.1016/j.ejmp.2017.07.024
6. Cipra BA (2000) The best of the 20th century: editors name top 10 algorithms. SIAM News 33(4):1–2
7. Frigo M, Johnson SG (1998) FFTW: an adaptive software architecture for the FFT. In: Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP ’98 (Cat. No.98CH36181), vol 3, pp 1381–13843. https://doi.org/10.1109/ICASSP.1998.681704
8. Frigo M, Johnson SG (1997) The fastest fourier transform in the west. mit-lcs-tr-728. In: The Proceedings of the 1998 International Conference on Acoustics, Speech, and Signal Processing, ICASSP ’98
9. Nukada A, Sato K, Matsuoka S (2012) Scalable multi-gpu 3-d fft for tsubame 2.0 supercomputer. In: SC ’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp 1–10. https://doi.org/10.1109/SC.2012.100
10. Gu L, Li X, Siegel J (2010) An empirically tuned 2D and 3D fft library on cuda gpu. In: Proceedings of the 24th ACM International Conference on Supercomputing. ICS ’10, pp. 305–314. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1810085.1810127.
11. Schwaller B, Ramesh B, George AD (2017) Investigating ti keystone ii and quad-core arm cortex-a53 architectures for on-board space processing. In: 2017 IEEE High Performance Extreme Computing Conference (HPEC), pp 1–7. https://doi.org/10.1109/HPEC.2017.8091094
12. Cooley JW, Tukey JW (1965) An algorithm for the machine calculation of complex fourier series. Math Comput 19(90):297–301
13. Gentleman WM, Sande G (1966) Fast fourier transforms: For fun and profit. In: Proceedings of the November 7–10, 1966, Fall Joint Computer Conference. AFIPS ’66 (Fall), pp 563–578. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1464291.1464352.
14. Swarztrauber PN (1984) Fft algorithms for vector computers. Parallel Comput 1(1):45–63. https://doi.org/10.1016/S0167-8191(84)90413-7
15. Luo Y, Li Y, Yang J, Ma L, Huang W, Xu B (2021) Optimization of the randomness extraction based on toeplitz matrix for high-speed qrng post-processing on gpu. In: 2021 13th International Conference on Communication Software and Networks (ICCSN), pp 261–264. https://doi.org/10.1109/ICCSN52437.2021.9463613
16. Zhao Z, Zhao Y (2018) The optimization of fft algorithm based with parallel computing on gpu. In: 2018 IEEE 3rd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC), pp 2003–2007. https://doi.org/10.1109/IAEAC.2018.8577843
17. Nejedly P, Plesinger F, Halamek J, Jurak P (2018) Cudafilters: a signalplant library for gpu-accelerated fft and fir filtering. Softw Pract Exp 48(1):3–9. https://doi.org/10.1002/spe.2507
18. Ogata Y, Endo T, Maruyama N, Matsuoka S (2008) An efficient, model-based cpu-gpu heterogeneous fft library. In: 2008 IEEE International Symposium on Parallel and Distributed Processing, pp 1–10. https://doi.org/10.1109/IPDPS.2008.4536163
19. Cılasun H, Resch S, Chowdhury ZI, Olson E, Zabihi M, Zhao Z, Peterson T, Wang J-P, Sapatnekar SS, Karpuzcu U (2020) CRAFFT: high resolution FFT accelerator in spintronic computational RAM. In: 2020 57th ACM/IEEE Design Automation Conference (DAC), pp 1–6. https://doi.org/10.1109/DAC18072.2020.9218673

20. Chen X, Lei Y, Lu Z, Chen S (2018) A variable-size fft hardware accelerator based on matrix transposition. IEEE Trans Very Large Scale Integr Syst 26(10):1953–1966. https://doi.org/10.1109/TVLSI.2018.2846688
21. Li Z, Jia H, Zhang Y, Chen T, Yuan L, Cao L, Wang X (2019) AutoFFT: a template-based FFT codes auto-generation framework for ARM and X86 CPUs. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '19. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3295500.3356138.
22. Ayala A, Tomov S, Luo X, Shaeik H, Haidar A, Bosilca G, Dongarra J (2019) Impacts of multigpu mpi collective communications on large fft computation. In: 2019 IEEE/ACM Workshop on Exascale MPI (ExaMPI), pp 12–18. https://doi.org/10.1109/ExaMPI49596.2019.00007
23. Chen S, Li X (2013) A hybrid gpu/cpu fft library for large fft problems. In: 2013 IEEE 32nd International Performance Computing and Communications Conference (IPCCC), pp 1–10. https://doi.org/10.1109/PCCC.2013.6742796
24. Gholami A, Hill J, Malhotra D, Biros G (2015) AccFFT: a library for distributed-memory FFT on CPU and GPU architectures. arXiv preprint arXiv:1506.07933
25. Cecka C (2017) Low communication fmm-accelerated fft on gpus. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '17. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3126908.3126919.
26. Markidis S, Chien SWD, Laure E, Peng IB, Vetter JS (2018) Nvidia tensor core programmability, performance amp; precision. In: 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp 522–531. https://doi.org/10.1109/IPDPSW.2018.00091
27. Sorna A, Cheng X, D'Azevedo E, Won K, Tomov S (2018) Optimizing the fast fourier transform using mixed precision on tensor core hardware. In: 2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW), pp 3–7. https://doi.org/10.1109/HiPCW.2018.8634417
28. Cheng X, Sorna A, D'Azevedo E, Wong K, Tomov S (2018) Accelerating 2d fft: exploit gpu tensor cores through mixed-precision. In: The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18), ACM Student Research Poster, Dallas, TX
29. Durrani S, Chughtai MS, Dakkak A, Hwu W-m, Rauchwerger L (2021) FFT Blitz: The Tensor Cores Strike Back. In: Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '21, pp 488–489. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3437801.3441623.
30. Abtahi T, Shea C, Kulkarni A, Mohsenin T (2018) Accelerating convolutional neural network with fft on embedded hardware. IEEE Trans Very Large Scale Integr Syst 26(9):1737–1749. https://doi.org/10.1109/TVLSI.2018.2825145
31. Lee J, Kang H, Yeom H-J, Cheon S, Park J, Kim D (2021) Out-of-core gpu 2D-shift-fft algorithm for ultra-high-resolution hologram generation. Opt Express 29(12):19094–19112
32. Kang H, Lee J, Kim D (2021) Hi-fft: Heterogeneous parallel in-place algorithm for large-scale 2D-fft. IEEE Access 9:120261–120273. https://doi.org/10.1109/ACCESS.2021.3108404

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.