Computer Systems
Science and Engineering

Tech Science Press

https://doi.org/10.32604/csse.2026.089793

ARTICLE

A Hybrid Optimization Approach for Large-Size FFT on High-Performance DSP

Junyang Tang[1,2], **Yonghua Hu**[1,2,*], **Yuxiang Gao**[1] **and Jie Luo**[3]

[1]School of Computer Science and Engineering, Hunan University of Science and Technology, Xiangtan, China
[2]Hunan Key Laboratory for Service Computing and Novel Software Technology, Xiangtan, China
[3]Hunan Compiler Information Technology Co., Ltd., Changsha, China

*Corresponding Author: Yonghua Hu. Email: yonghuahu@hnust.edu.cn

Received: 25 July 2026; Accepted: 21 September 2026; Published: 27 October 2026

ABSTRACT: To meet the high-performance requirements of FFT computation under on-chip resource constraints, this paper proposes a cooperative multi-core computation model tailored for deterministic large-size floating-point complex FFT on a multi-core vector digital signal processor (DSP) architecture. Based on an in-depth analysis of FFT computational characteristics and butterfly combination patterns of FFT in the multi-core DSP environment, the model integrates SIMD vectorization, vector-shuffle optimization, and inter-core task coordination. It employs fine-grained on-chip cache management and optimized DMA transfer strategy to significantly reduce data movement overhead and enhance SIMD unit utilization. Unlike existing DSP-specific FFT accelerations that rely on explicit matrix transposition or dynamic task scheduling, the proposed model fuses bit-reversal reordering into the DMA write-back phase, and coordinates the cores through statically mapped task partitioning and hardware barriers. Experiments are conducted on the YHFT-Matrix multi-core vector DSP processor using 8M-point (2^{23}-point) floating-point complex FFT as the primary benchmark sample-size, together with 2M-point and 32M-point cases. The optimized assembly implementation is verified against MATLAB reference outputs. Compared with a single-core vectorized C baseline on the YHFT-Matrix platform, the proposed algorithm achieves an average speedup of 15.3× and, in an engineering comparison on a general-purpose CPU, it also achieves shorter execution times than multi-threaded FFTW (by 1.45×, 1.97×, and 2.11× at 2M, 8M, and 32M points, respectively.

KEYWORDS: Deterministic large-size FFT; butterfly operations; DSP; SIMD

1 Introduction

Fast Fourier Transform (FFT) is an efficient implementation of the Discrete Fourier Transform (DFT). It is one of the most fundamental and core algorithms in engineering fields, including digital signal processing [1,2], image analysis [3,4], and radar and communications [5,6].

With the development of high-resolution and high-sampling-rate real-time systems, such as remote sensing imaging [7,8], radio astronomy [9–11], and high-performance scientific computing [12], the demand for high-performance computation of large-size FFT has increased rapidly in recent years. However, traditional single-core DSPs struggle to address the enormous computational complexity of large-size FFTs and the data throughput pressure at the level of hundreds of MB. Therefore, vector DSP architectures based on multi-core collaboration have become the mainstream solution path [13].

Existing studies have investigated optimization strategies for large-size FFT on various multi-core platforms. Zhan et al. exploited 24-core data parallelism and DMA indexed implicit transposition on the

Copyright © 2026 The Authors. Published by Tech Science Press. This work is licensed under a Creative Commons Attribution 4.0 International License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

FT-X DSP, significantly reducing transposition overhead and achieving a transposition time reduction of up to 91% at the 8K × 8K scale [14]. Guo et al., targeting the "Shenwei 26010", proposed a "column-equal-division row-contiguous" strategy combined with register communication and strided transfer, acheiving a maximum speedup of 65× at the 4M-point scale [15]. Xia et al. compared cached and cacheless transposition implementations on the TI C6678, demonstrating the effectiveness of the space-for-time strategy [16]. Wang et al. extended 2D FFT to 3D through multi-dimensional decomposition, enabling the acceleration of ultra-long FFTs up to 4G points [17]. These studies demonstrate that processor-specific tuning is indispensable, but each method is tightly coupled to a particular platform, lacks portability and scalability, and cannot be directly applied to the fixed-scale, deterministic FFT scenario addressed in this paper.

To address the above challenges, this paper proposes a hybrid optimization method for large-size FFT on a multi-core vector DSP platform. Based on mixed-radix decomposition, the method combines multi-level task partitioning, SIMD (single-instruction, multiple-data) vectorization, vector shuffle optimization, and multi-core task collaboration to jointly optimize computation and memory access. For computation, it employs data preprocessing and layout optimization together with vectorized butterfly operations. For data movement, it adopts a double-buffering strategy and streamlined asynchronous DMA scheduling, thereby fully exploiting the computing resources and memory-transfer bandwidth of the multi-core DSP.

Compared with existing FFT acceleration methods on other DSP platforms, the proposed optimization has three distinctive features: First, it is transposition-free: bit-reversal reordering is fused into the DMA write-back phase through a pre-computed global index table and multi-channel concurrent scattered writes, whereas the FT-X and TI C6678 implementations rely on explicit matrix transposition to restore natural output order. Second, it resolves final-stage SIMD underutilization at the register level. When a butterfly group is smaller than the vector width, vector shuffle instructions pack multiple small butterfly groups to fit the vector register width and reuse radix-4 shuffle templates for radix-2 butterflies, avoiding the padding or scalar tail processing used in other SIMD FFT kernels. Third, it exploits the determinism of fixed large-size FFTs (such as 8M-point FFT), to achieve communication-free multi-core coordination. Task partitioning, data layout, and output address mapping are statically derived from core IDs, and cores synchronize only through hardware barriers, eliminating the message queues and dynamic scheduling required in the Sunway 26010 and 3D FFT implementations.

The main contributions of this paper are as follows: (1) a transposition-free DMA-based bit-reversal mechanism that restores the natural output order during write-back without interrupting the computation pipeline; (2) a register-level shuffle-packing strategy that maintains full vector-unit utilization in the final stages dominated by radix-2 butterflies; and (3) a static and deterministic multi-core mapping with hardware-barrier synchronization model that removes explicit communication overhead.

2 Matrix Architecture

2.1 Introduction of Matrix Kernel

The Matrix kernel is highly optimized for compute-intensive algorithms such as matrix multiplication and FFT. Each core is based on a Very Long Instruction Word (VLIW) architecture, comprising a five-issue scalar processing unit (SPU) and a six-issue vector processing unit (VPU). The two processing units are tightly coupled and support the simultaneous issue of up to 11 instruction words, thereby exploiting both instruction-level parallelism and vector-based data parallelism [18]. The basic framework of Matrix is shown in Fig. 1. The VPU consists of 16 homogeneous vector processing elements (VPEs), which exchange data through a reduction tree and a shuffle network.

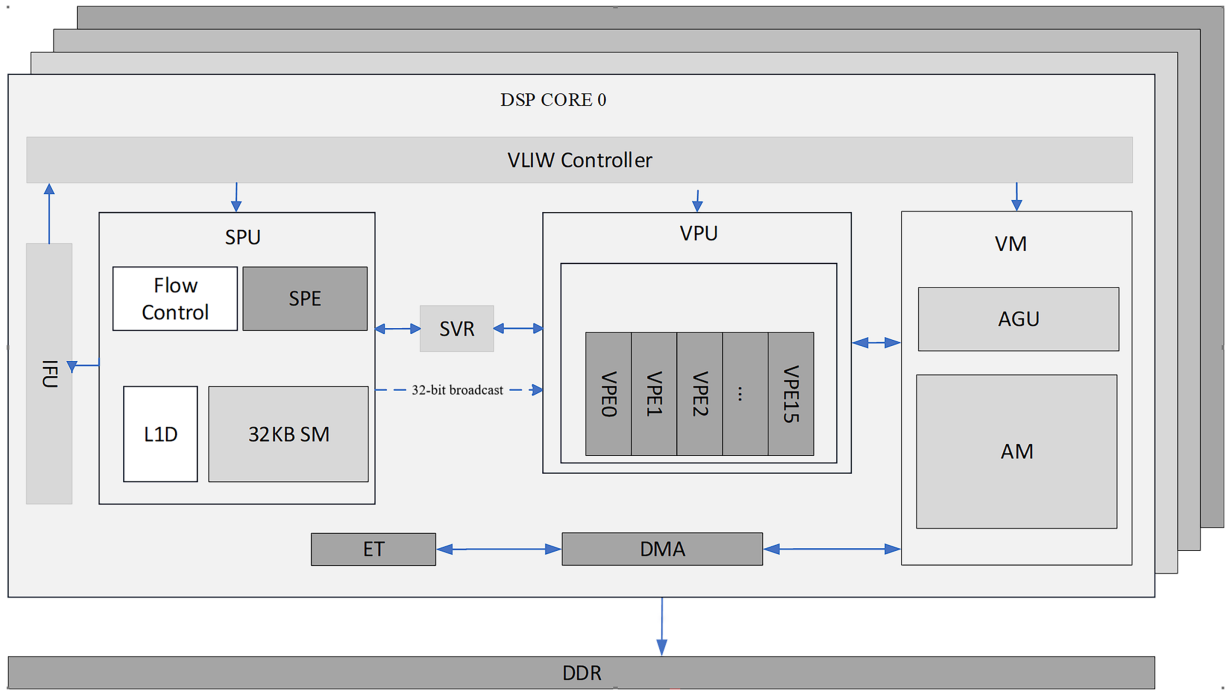


Figure 1: Matrix architecture.

2.2 DMA Performance Analysis

DMA (Direct Memory Access) is a data transfer mechanism executed by a dedicated DMA controller, which efficiently moves data from one address space to another without intervention from the processor core. In parallel computing architectures, reasonable configuration and utilization of DMA are key factors in improving overall program performance. In the Matrix architecture, DMA serves as the central hub for data exchange. Through a high-speed on-chip bus, it internally connects the local memory resources of each DSP core and externally links on-chip peripherals and external memory, thereby providing powerful background data transfer capabilities. Given that large-size FFT algorithms involve frequent movement of massive data volumes, efficient scheduling of DMA is particularly critical.

Each compute core in the Matrix processor integrates a DMA controller. Each DMA controller provides 16 independent logical transfer channels, supporting concurrent data stream management. As shown in Fig. 2, DMA transfer behavior is defined by transfer parameter sets. Each parameter set consists of 8 words and specifies key configurations such as transfer mode, source address, destination address, and data volume. DMA supports data transfers organized as three-dimensional arrays, whose dimensions are data element, frame and channel in sequence. The frame is the minimum unit of DMA transfer. Data elements within a frame must be stored contiguously, and the number of elements is a frame denoted by FCNT. The address offset between adjacent frames is called the frame index, denoted by FIDX.

If the total number of transferred data elements is Num, then when FIDX = 0, the data are transferred continuously in row-major order, as shown in Fig. 3a; when FIDX > 1, the transfer becomes a strided transfer, as shown in Fig. 3b. When FCNT = Num, the transfer corresponds to large-frame transfer, as shown in Fig. 3c.

Transmission control word 1		Word 0
Transmission control word 2		Word 1
source address		Word 2
Source frame count	Source unit count	Word 3
Dst address		Word 4
Dst frame count	Des unit count	Word 5
Dst Frame index	Source Frame index	Word 6
Parameter link address	Res	Word 7

Figure 2: DMA transfer parameter set.

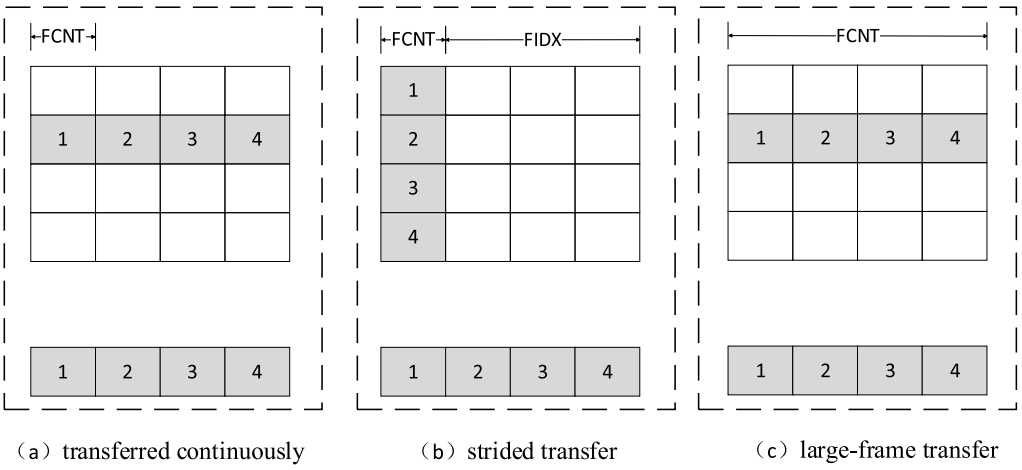


Figure 3: Schematic of DMA transfer modes.

To evaluate the performance characteristics of strided transfers, this paper conducts a dedicated test, with results shown in Fig. 4. In the figure, the primary vertical axis, labeled as “FIDX DMA Transfer Rate”, represents the effect of the frame index on the DMA transfer rate under small-frame transfer (FCNT = 1), while the secondary vertical axis, labeled as “FCNT DMA Transfer Rate”, represents the effect of the frame size on the DMA transfer rate under contiguous transfer (FIDX = 0). The experiment shows that the transfer granularity (i.e., the size of FCNT) is the primary factor affecting the DMA transfer rate, whereas FIDX has a relatively minor impact. For example, when the data volume is 8192, the maximum rate with FCNT = 1 is only 4.3, regardless of how FIDX varies, whereas the rate with FCNT = Num reaches as high as 130.03. This phenomenon can be attributed to the fixed bus width design of the on-chip interconnect: when FCNT is small, the bus bandwidth cannot be fully utilized, resulting in resource underutilization. At the same time, fine-grained accesses exacerbate the non-contiguity of memory accesses, further reducing memory access efficiency. Therefore, in practical applications, large-frame transfers should be used whenever possible to improve DMA transfer efficiency and optimize the overall system performance.

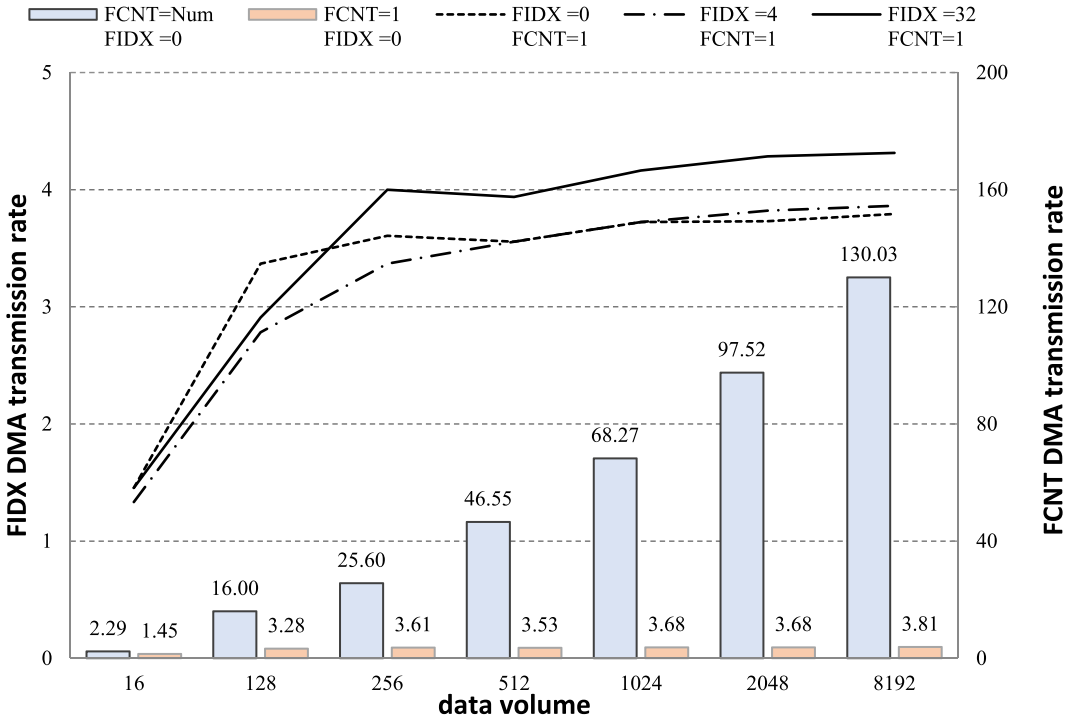


Figure 4: DMA transfer rate.

3 Radix-4 FFT Algorithm Description

FFT is an efficient algorithm for computing the Discrete Fourier Transform (DFT) and its inverse. In spectrum analysis, FFT is widely used for signal processing and frequency-domain analysis. By transforming a signal from the time domain to the frequency domain, we can better understand the characteristics of the signal, such as frequency components and their variations.

Among numerous FFT algorithms, the Cooley-Tukey algorithm [19] is the most classical. Its core idea is to recursively decompose a large-scale DFT into several smaller-scale sub-DFTs using a divide-and-conquer strategy, significantly reducing computational complexity. This paper adopts a mixed-radix strategy combining radix-4 and radix-2 to accommodate large-size FFTs.

Specifically, the radix-4 FFT (Eq. (1)) divides an input sequence of length N into four subsequences of length $N/4$ according to their indices, recursively performing butterfly operations until the decomposition reaches 4-point DFT. When the sequence length is not divisible by 4, it naturally switches to the radix-2 FFT (Eq. (2)), dividing the remaining part into two subsequences for 2-point butterfly operations. This mixed-radix structure not only retains the high computational efficiency of radix-4 FFT but also possesses the flexibility of radix-2 FFT. It can precisely match arbitrary 2^n point sizes, thereby maximizing SIMD parallelism and memory access efficiency while ensuring the integrity of the algorithm.

$$X(k) = \sum_{n=0}^{N/4-1} \left[x(n) + W_N^{(Nk/4)} x\left(n + N/4\right) + W_N^{(Nk/2)} x\left(n + N/2\right) + W_N^{(3Nk/4)} x\left(n + 3N/4\right) \right] \tag{1}$$

$$X(k) = \sum_{n=0}^{N/2-1} \left[x(n) + (-1)^k x\left(n + N/2\right) \right] W_N^{nk} \tag{2}$$

4 Parallel Mapping and Optimization of Large-Size FFT Algorithms

In modern high-performance signal processing systems, the scale of FFT is continuously evolving toward ultra-large point counts. Among them, the 8M-point 32-bit floating-point complex FFT, hereinafter referred to as 8M-point FFT, has become a representative benchmark for evaluating the joint optimization capabilities of processor architectures and algorithms. It corresponds to a typical data block size in key applications such as high-resolution radar, synthetic aperture radar (SAR) imaging, and broadband communications. This scale representative from three perspectives: engineering typicality, as it commonly appears in practical system requirements; technical challenge, as it involves a 64 MB data volume that far exceeds the on-chip cache capacity, resulting in highly coupled computation and memory access bottlenecks; and structural regularity, as its power-of-two size enables efficient decomposition into a mixed-radix butterfly structure, providing a suitable basis for deep vectorization and parallelization.

Based on this, this paper takes the 8M-point FFT as the specific research object and systematically investigates parallel mapping strategies and optimization methods for multi-core DSPs. Since the Matrix architecture is a homogeneous multi-core architecture with multiple identical DSP cores, and since the 8M-point FFT algorithm exhibits good data parallelism, this work adopts a data-parallel approach to exploit multi-core parallelism.

4.1 Single-Core Vectorized Parallel Computing Model

In 8M-point FFT computation, data movement has become a critical performance bottleneck [20]. General-purpose FFT algorithms often fail to fully exploit the parallel computing potential of hardware. Therefore, fully unleashing the available computational resources requires deep, customized optimization tailored to specific hardware architectures and point sizes.

To achieve efficient optimization of the 8M-point FFT algorithm, a systematic processing workflow must be followed, primarily consisting of the following four core steps:

(1) **Data Pre-Processing:** Before computation, the required twiddle factors are generated in advance according to the FFT point size and arranged into specific sequences for subsequent computation stages.
(2) **Data Transfer:** Since core vectorized operations must be performed in vector space AM (Array Memory), the data to be processed must first be efficiently transferred from off-chip memory to the on-chip AM.
(3) **Core Computation Optimization:** Within the AM, the core butterfly operations of the FFT are deeply optimized by exploiting the SIMD instruction set to achieve highly parallelized computation.
(4) **Result Write-Back:** After computation, transfer the final result data from vector space back to scalar memory.

According to the above processing workflow, this study focuses on optimization at the following four key technical levels:

(1) **Data Pre-Processing and Layout Optimization:** Optimize the generation, sorting, and joint layout of twiddle factors and input data to reduce random accesses and data dependencies during computation.
(2) **Vector Parallel Computation Optimization:** Propose an efficient vectorized butterfly computation model to maximize SIMD parallelism and improve the utilization of compute units.
(3) **Data Shuffle and Reordering Optimization:** To address the problem of sparse data distribution in the final stages of FFT, where vector registers cannot be fully filled, employ vector shuffle instructions to reorganize data dynamically within vector registers and maintain sustained full utilization of the compute units.

(4) **DMA-Based Bit-Reversal Optimization:** Adopt a pre-generated bit-reversed address mapping table together with the two-dimensional strided transfer capability of multi-channel DMA to perform bit-reversal reordering concurrently with data write-back.

4.1.1 Data Pre-Processing and Layout

According to the computational principle of FFT, the underlying computational pattern—the butterfly operation—is the same across all stages, making it inherently amenable to SIMD vectorization. However, on the Matrix vector DSP, the data involved in the computation must be carefully prearranged in the AM.

(1) Data Pre-processing

The data required for complex FFT computation include the source data to be transformed and the corresponding twiddle factors. The source data are typically stored in an interleaved real-imaginary mode; however, the DMA unit can reorganize its layout when transferring them to the cache [21]. There are two main data storage modes:

(a) Real-Imaginary Interleaved Storage: The original data layout in memory is preserved. However, vector processing requires the real and imaginary parts to be separated into two vector registers. This will introduce additional overhead.

(b) Real-Imaginary Separated Storage: The data layout will be changed when data enters the cache, in order to reduce load operation time. However, this requires the DMA to perform strided transfer, which takes longer than contiguous transfers.

As discussed in Section 2.2, contiguous transfers are more efficient than strided transfers. Although the interleaved layout requires data separation, this separation can be completed concurrently using the vector modulo-2 load instruction VLDDWM2, whose latency equals that of the conventional vector double-word load instruction VLDDW. Therefore, as shown in Fig. 5, approach (a) requires fewer clock cycles and delivers better overall performance.

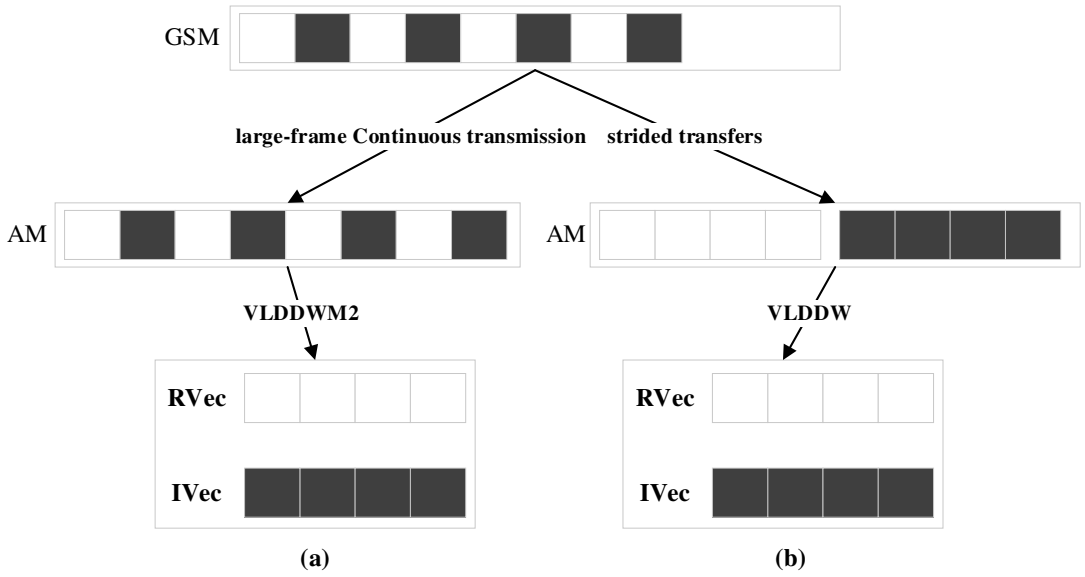


Figure 5: Schematic of VLDDWM2.

The complex multiplications in FFT mainly involve twiddle factors, which are handled in one of two ways. The first approach computes the required twiddle factors on the fly before each butterfly operation. It

is simple and memory-efficient but time-consuming. The second approach pre-computes all twiddle factors, stores them in the same layout as the source data, and loads them into AM via DMA, thereby reducing computation time. The corresponding implementation is shown in Algorithm 1.

Algorithm 1: Calculation of twiddle factors (CalcWData)

input: data volume cxNum
output: twiddle factors Wts

```
1. float *Ws_im = Ws_re + cxNum
2. int n = 3*cxNum/4
3. for i = 0; i < n; i++ do
4.         Wts_re[i] = cos(2*π*i/cxNum)
5.         Wts_im[i] = −sign*sin(2*π*i/cxNum)
6. end for
7. while cxNumInBG >= 8 do
8.         for i = 0; i < bfNumInBG; i = i + 1 do
9.                 for k = 0 to 2 do //three twiddle-factor rotations of a radix-4 butterfly
10.                    Ws_re[offset + bfNumInBG*k + i] = Wts_re[i*(k + 1)*bgNum]
11.                    Ws_im[offset + bfNumInBG*k + i] = Wts_im[i*(k + 1)*bgNum]
12.        offset += bfNumInBG*3
13.        cxNumInBG = cxNumInBG >> 2
14.        bgNum *= 4
15.        bfNumInBG /= 4
16. end while
17. return
```

(2) Data layout

The 8M-point FFT operates on 8,388,608 complex samples; since each single-precision complex sample occupies 8 bytes, the input data F alone requires 64 MB. In addition to the input, the implementation must allocate storage for the output data (64 MB), the pre-computed twiddle factors W (64 MB), and the intermediate buffers used for the stage-by-stage data exchange and the double-buffered DMA transfers (64 MB per buffer). The total storage requirement of the implementation therefore reaches about 256 MB. In contrast, the on-chip AM space of the processor is only 512 KB and the on-chip high-speed buffer GSM is only 8 MB. Considering that GSM must also store program segments and AM must store twiddle factors W, the space partitioning for GSM and AM is shown in Tables 1 and 2, respectively. By adopting a multi-level caching strategy with double-buffering memory management, the limited on-chip resource AM is fully utilized. DMA is employed to overlap computation and data movement, thereby hiding DDR access latency and improving overall throughput [22].

GSM adopts functional partitioning with pre-loading caching. It separates the code and data required for system operation from computation-specific data and reserves an independent pre-loading region for frequently accessed W data. Because W is reused across multiple computation stages and cores, this region stores it for extended periods, while “Buffer C” in AM prepares the W data for the upcoming stage. This design greatly reduces DDR access pressure, allowing AM to retrieve data from GSM quickly without waiting for slow DDR accesses. AM adopts a double-buffering pipeline mode. The 512-KB memory is divided into two functionally equivalent groups: “compute areas” (A/B) and “buffer areas” (C/D). While one group (A/B) performs computation, the other group (C/D) is filled by DMA with data for the next stage. This “foreground

computation, background filling" mechanism effectively overlaps computation and data movement, and it is crucial to improving the performance of large-size FFT.

Table 1: Address space partitioning for GSM (8 M).

Partition	Size	Content	Description
Public Data Space	1 MB	Program segment	Stores system-level shared resources such as program segments, data segments, and code segments. The content of this area is relatively static and is loaded once during system initialization.
Buffer	4 MB	F (output/input data)	Stores the input/output data blocks to be processed. These blocks are pre-loaded from DDR by DMA for use by the compute units in AM.
	3 MB	W (twiddle factors)	Store the twiddle factors required for subsequent multiple computing stages. These data are pre-loaded from DDR by DMA for use by the compute units in AM.

Table 2: Address space partitioning for AM (512 K).

Partition	Size	Content	Description
Compute area A	128 KB	W (twiddle factors)	W data required for the current-stage computation, loaded from GSM to this area by DMA.
Compute area B	128 KB	F (output/input data)	F data required for the current-stage computation, which performs butterfly operations with the W data in "Compute Area A".
Buffer area C	128 KB	W (output/input data)	Pre-load the W data required for the next batch of computation to achieve overlap between computation and DMA transfer.
Buffer area D	128 KB	F (output/input data)	Pre-load the F data for the next batch.

4.1.2 Vector Parallel Computing

To maximize the utilization of the Matrix processor's vector computing capability and AM bandwidth, we propose a hierarchical data computation model and implement an efficient SIMD kernel. Based on the data scale and memory access characteristics of different stages of the 8M-point FFT, the entire computation process is divided into four main stages: L1-layer (first-layer) computation, inter-butterfly-group loop-layer computation, intra-butterfly-group loop-layer computation, and register-based butterfly-group computation [23].

(1) L1-Layer Computation

At the L1 layer, the data comprise a single butterfly group whose size far exceeds the capacity of AM. As shown in Fig. 6, this stage adopts a "sliding data fetch and double-buffering round-robin" strategy: the

butterfly-group data are divided into sliding blocks, each being one-fourth of the AM's maximum source-data capacity. These blocks will be loaded sequentially from the four starting positions of the butterfly group in a sliding-window fashion. AM is logically partitioned into two equal halves—the X and Y regions—which serves as a double buffer. The execution flow is as follows: (1) Initialization: Before the loop begins, the first batch of twiddle factors W and source data F is pre-loaded into the X region via the DMA channel; (2) Round-robin execution: in cycle A, DMA transfers the next batch of W and F into the Y region while the core processes the data already in the X region; in cycle B, DMA writes back the X region results to the GSM buffer and loads the next batch into the X region while the core processes the Y region; (3) Tail processing: after the loop ends, a final computation handles the remaining tail data.

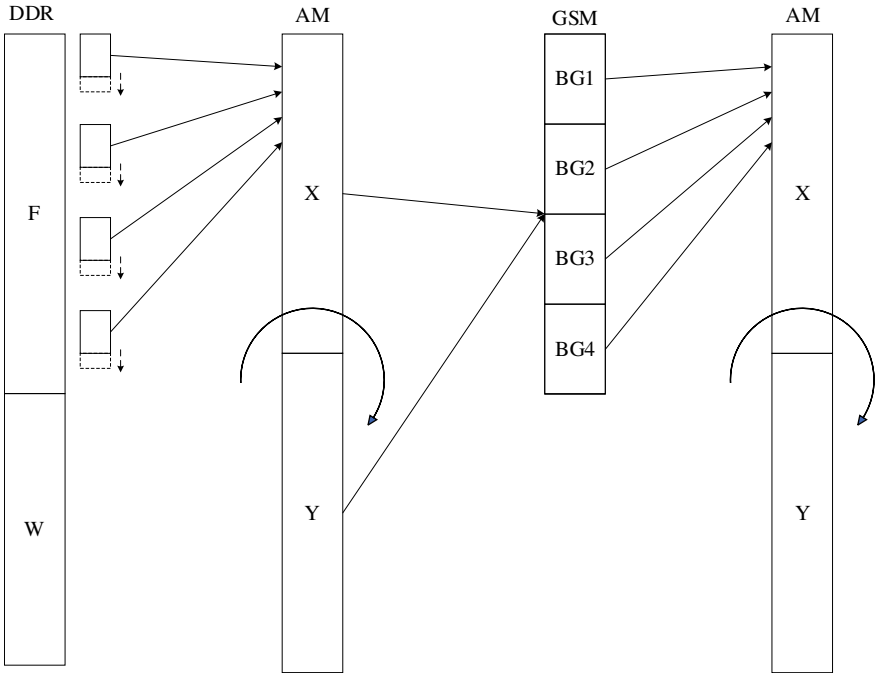


Figure 6: FFT L1-stage computation schematic.

This mechanism fully overlaps computation with data transfer, masking memory-access latency while keeping the compute units utilized and the memory bandwidth saturated.

(2) Inter-Butterfly-Group Loop-Layer Computation

In the middle layers, the data is divided into multiple butterfly groups, each of which still exceeds the capacity of AM. Therefore, the key optimization lies in twiddle-factor reuse. Because all butterfly groups share identical twiddle factors, the required W data is loaded into the X region once, with a duplicate copied into the Y region. Computation then alternates between the two regions, processing blocks at corresponding positions across different groups: while the core processes group A in the X region, DMA loads the next batch of group B into the Y region, and *vice versa* next time. This strategy maximizes twiddle-factor locality, thereby reducing redundant loads of W and memory-bandwidth pressure.

(3) Intra-Butterfly-Group Loop-Layer Computation

After the previous stages, one butterfly group is small enough to fit entirely within the AM. To avoid unnecessary data movement, the data is kept resident in the AM at this stage, and all operations from the current layer to the N3 layer (the $(L-2)$-th layer, where $L = 12$) are completed through round-robin

computation. For the 8M-point FFT, the N3 layer outputs 64-point complex subsequences, which exactly match the vector-register granularity (128 floating-point numbers), enabling fully vectorized computation without padding.

(4) Register-Based Butterfly-Group Computation

In the last three layers of the FFT, the butterfly groups are small enough to fit within vector registers. As shown in Fig. 7, computation at this stage is performed entirely within the registers without accessing the AM or caches, thereby completely eliminating memory access latency and achieving the highest computational efficiency.

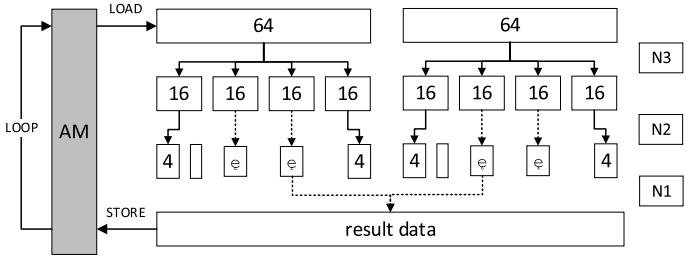


Figure 7: FFT last-three-stage computation schematic.

4.1.3 SIMD Vectorization and Data Shuffling

Across the computation stages of FFT, the number of butterfly groups contained in each stage differs, and the data layout for each butterfly operation changes accordingly. Specifically:

L1 layer: contains only one butterfly group, with data stored contiguously in the AM;

L2 layer: contains 4 butterfly groups, and all groups perform identical operations.

For the parallel processing of multiple butterflies, the corresponding data should be stored in the AM at each stage. According to the Matrix architecture, each vector processing unit (VPU) contains 16 vector processing elements (VPEs). When the butterfly-group data volume is less than 64, not all 16 VPEs can participate simultaneously, so vector shuffle operations must reorganize the data. For the 8M-point FFT, a data volume of 64 corresponds to the N3 layer (the (L − 2)-th layer, where L = 12). Accordingly, this paper divides the 8M-point FFT into two parts: "L < 10" and "L ≥ 10".

For the case where "L < 10", the objective is to maximize register utilization and instruction-level parallelism while minimizing the shuffle count in the final-stage under fixed point-count constraints. The maximum unrolling factor is derived from Eq. (3), and software-pipelining is applied to the resulting code scheduling.

$$U_{\max} = \min\left(\left\lfloor \frac{VR_{total}}{VR_{per\_iter}}, \frac{AR_{total}}{AR_{per\_iter}}, \frac{OR_{total}}{OR_{per\_iter}} \right\rfloor\right) \tag{3}$$

In this large-size FFT implementation, considering that the real and imaginary parts of complex butterfly data are processed independently, a dedicated vector instruction optimization strategy is designed accordingly, where shuffle operations are used in the "L ≥ 10" register-based butterfly-group stage.

Specifically, after the data is loaded from the AM into the vector registers (VR), the butterfly operations are executed at first. Then, unlike the traditional flow, the results are retained in the registers and rearranged through shuffle operations instead of being stored back to memory. This addresses the problem that the butterfly-groups in the final stages of the mixed-radix FFT are too small to fully utilize the SIMD vector

width. For single-precision complex numbers, each VPE stores one float, so 16 VPEs accommodate 8 complex numbers. When the FFT size N is not a power of 4, the last two stages usually form a radix-4 + radix-2 structure (an 8-point FFT followed by a 2-point FFT), leaving only 2 complex numbers per group—far below the capacity of 16 VPEs. Without packing, the 2-point butterfly groups of the final stage would occupy only 4 of the 16 VPEs (25% lane utilization), whereas the proposed packing achieves 100%. The proposed shuffling strategy packs data from multiple small butterfly groups into the same vector register: eight 2-point butterfly groups (split into real and imaginary parts) together fill two full groups of 16 VPEs, achieving fully vectorized computation.

As shown in Fig. 8, the shuffle patterns of radix-2 butterflies differ from those of radix-4; nevertheless, with a reordering sequence designed to align the data layout, radix-2 operations can reuse parts of the radix-4 shuffle templates. For example, before the penultimate stage, four groups of data are first loaded into the vector registers, and a specific shuffle sequence then rearranges the data to meet the input requirements of radix-2 butterflies.

A	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
	32	34	36	38	40	42	44	46	48	50	52	54	56	58	60	62
B	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31
	33	35	37	39	41	43	45	47	49	51	53	55	57	59	61	63

Figure 8: Radix-2 shuffle strategy.

As shown in Fig. 9 (N2-layer shuffle), A, B, C, and D denote the four radix-4 butterflies; after two rounds of shuffling, the data transition from the N3-layer layout to the N2-layer layout is completed. This reuse mechanism reduces the instruction-address-space footprint of dedicated shuffle patterns, thereby improving both instruction-set compactness and hardware-resource utilization.

A	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
B	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

A	0	1	2	3	4	5	6	7	32	33	34	35	36	37	38	39
B	8	9	10	11	12	13	14	15	40	41	42	43	44	45	46	47
C	16	17	18	19	20	21	22	23	48	49	50	51	52	53	54	55
D	24	25	26	27	28	29	30	31	56	57	58	59	60	61	62	63

A	0	1	8	9	16	17	24	25	32	33	40	41	48	49	56	57
B	2	3	10	11	18	19	26	27	34	35	42	43	50	51	58	59
C	4	5	12	13	20	21	28	29	36	37	44	45	52	53	60	61
D	6	7	14	15	22	23	30	31	38	39	46	47	54	55	62	63

Figure 9: Radix-4 shuffle strategy.

4.1.4 DMA Bit-Reversal Optimization

In the decimation-in-time (DIT) 8M-point FFT, the output is inherently in bit-reversed order. To produce naturally ordered results without disrupting the computation pipeline, this paper proposes a DMA-driven bit-reversal reordering mechanism that works closely with the register-level butterfly computation.

To avoid the latency and power overhead of computing bit-reversed addresses at runtime, the system pre-computes a global bit-reversal index array G during initialization, where G[k] is the starting-position offset of the k-th data batch in the sequentially ordered output. The table is then stored in shared memory. Thus, DMA transfers can look up addresses directly, thereby eliminating all runtime address-calculation overhead.

To achieve multi-core scalability, the global batch space is partitioned using a two-dimensional index structure:

$$\text{global index}\,(c, b) = b + B \cdot \text{c} \tag{4}$$

where $c \in [0, 4)$ is the core ID, b is the local batch index within the current core, and B is the fixed inter-core offset, which ensures that each core processes non-overlapping data regions. Correspondingly, the target address for core c processing the b-th batch of data is:

$$\text{Dstaddr} = \text{resDataRe} + \text{G}\left[\text{global index}\,(c, b)\right] \cdot S \tag{5}$$

where $S = 2 \times dataByteNum$ is the byte length of a complex. This design achieves conflict-free, deterministic multi-core scattered writes while fully preserving the semantics of the global bit-reversal permutation.

As shown in Fig. 10, FFT computation is organized into several register-resident butterfly groups in this stage. Each group completes all butterfly operations for a fixed point count at the N3 layer within the registers and performs intra-group shuffling, with its output temporarily stored in the AM.

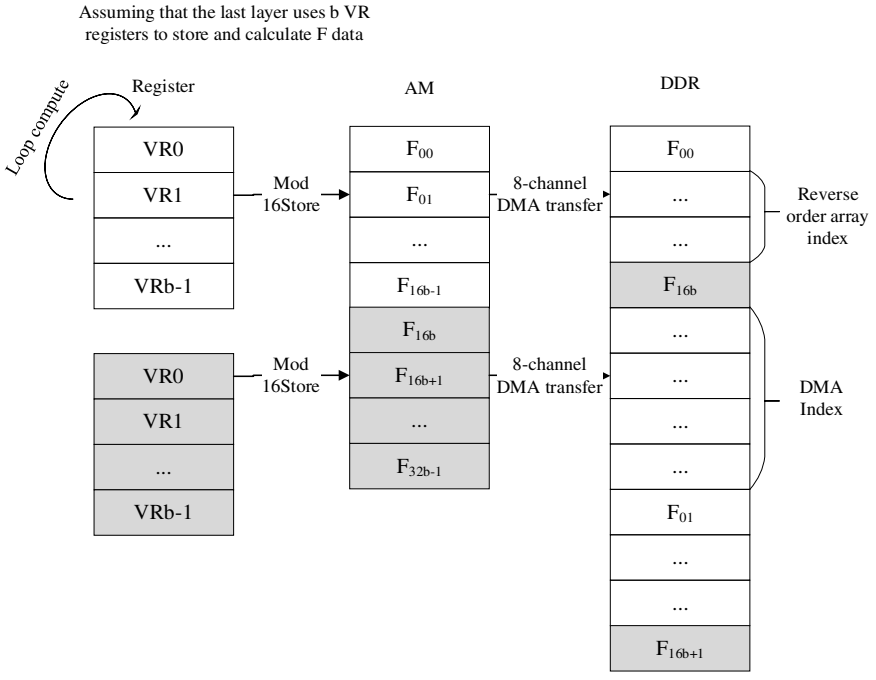


Figure 10: Schematic of DMA Bit-reversal addressing.

Subsequently, DMA performs batch transfers: each DMA transfer operation uses 8 channels to transfer 8 batches of data. The source addresses are linearly distributed in the source data buffer vBRDatas_r, while the target addresses are obtained from the pre-computed index table G and the two-dimensional index mapping, thereby achieving nonlinear yet deterministic bit-reversed writes. As described in Algorithm 2, there are pntBatchNum_am data batches to be transferred, each of size pntBatchByteNum bytes.

Algorithm 2: Batched DMA bit-reversal (DMA-BR)

input: void
output: void

1. **While** i < pntBatchNum_am **do**
2. Wait for DMA channels 0–7 to finish
3. for ch in [0, 1, . . ., 7] do
4. if i >= pntBatchNum_am:
5. break out of all loops
6. Src = vBRDatas_re + pntBatchByteNum*i
7. revIdx = G[b + B*c]
8. Dst = resDatasRe + revIdx*(dataByteNum*2)
9. DMA_ConfigureChannel(ch, src, dst)
10. i++
11. b++
12. end for
13. **end while**
14. Start_DMA_Channels(0xFF)

4.2 Multi-Core Programming Model

The multi-core programming model works as follows: each DSP core has a register that stores its core ID. Before the main program starts, the main program is flashed to every core, and then the master core performs a DDR initialization. Each core determines its data addresses by core ID and process a distinct data portion, thereby achieving data parallelism.

4.2.1 Program Initialization

During the project development phase, the master core is started from an external interface through the emulator and executes the initialization assembly program. After execution completes, the PC is set to the BOOT module. The master core allocates the private space of the slave cores and copies the data in the private space (including the main program) to the private address space of the slave cores, and then remaps the extended address space of the slave cores. In this process, the slave cores wait for the master core to complete initialization.

4.2.2 Multi-Core Synchronization

Because the computation structure and data partitioning of the 8M-point FFT are deterministic, this paper leverages the hardware barriers of the Matrix architecture to construct an implicit inter-core communication mechanism that requires no explicit message passing, enabling low-overhead parallel FFT computation.

The synchronization and mutual-exclusion mechanisms of the Matrix system rely on a shared memory space among cores, with an underlying scalable hardware synchronization mechanism. All synchronization

units are uniformly addressed in the global address space. Thus, a hardware barrier operation is equivalent to a synchronization variable through standard load/store instructions.

To ensure the atomicity of shared-data writes and guarantee that DMA reads the most recent data from GSM, the following protocol is used: before writing results back to GSM, the producer waits on the corresponding barrier variable; the consumer likewise waits on the same variable before reading shared data. After synchronization is complete, the hardware automatically releases the barrier for the next round of shared access.

The barrier unit enables multiple cores to synchronize at specific execution points: when all cores have completed an FFT stage, barrier synchronizaiton is required. Notably, in the recursive decomposition of the multi-stage mixed-radix FFT, a barrier synchronization must follow every computation stage to preserve data consistency and pipeline correctness.

The synchronization process is as follows: when a core arrives at the barrier, the hardware records its core ID and atomically increments the barrier counter. Because arrival signals are collected through hardware interrupts, a core cannot re-enter the barrier before the barrier is released, and each hardware core may arrive only once per synchronization. When the counter reaches the preset number of participating cores, the hardware multicasts a release signal so that all waiting cores exit the wait state simultaneously. Once all cores have exited the barrier, the counter and arrival flags are automatically cleared. The assembly code below illustrates the standard flow of each FFT stage (DMA load → computation → DMA write-back → barrier synchronization):

```
# 1. Read the current core ID (low 4 bits)
READ_COREID    Rcore
# 2. Compute the offset of local core data and initiate DMA loading
MUL            Roffset,Rcore,#N_per_core
ADD            Rsrc,Rsrc_base,Roffset
DMA_LOAD       Rsrc,LocalBuf
# 3. Execute the current layer FFT butterfly operation
FFT_KERNEL     LocalBuf
# 4. DMA write back the result to the corresponding GSM location
ADD            Rdst,Rdst_base,Roffset
DMA_STORE      LocalBuf,Rdst
# 5. Barrier synchronization
SET_BARRIER    #0x30100400
WAIT_BARRIER   #0x30100400
```

All processing cores load an identical program image. After startup, each core reads its core ID from a dedicated hardware register. Based on the preset total point count (8M) and the number of cores (4), the core computes the offset of its data block in the global GSM address space. It then independently issues DMA requests to move its sub-block into AM and executes the same butterfly code as all other cores. Because the point count is fixed, task partitioning, data layout, and output address mapping are all determined statically at compile time or during initialization, eliminating runtime dynamic scheduling.

The essence of this mechanism is to embed task allocation into the mapping between core IDs and data addresses, simplifying communication to coordinated access of the shared address space without message

construction, queue management, or status feedback. Compared with explicit communication models (e.g., IPC message queues, where the master core distributes task descriptions and slave cores parse messages and return status), this design avoids protocol-processing latency as well as queue contention, cache pollution, and frequent control-flow switches—overheads that grow rapidly with the number of cores and FFT stages and significantly erode the benefits of parallelism.

5 Experimental Results and Analysis

Based on the methods described above, 2M-point, 8M-point, and 32M-point FFT implementations were designed for the YHFT-Matrix vector DSP. Single-core and multi-core optimized assembly programs were implemented, incorporating loop optimization techniques such as loop unrolling and software pipelining. The correctness of the FFT assembly algorithm was verified against a MATLAB program (Section 5.1). The 8M-point benchmark is used for the internal stage-by-stage performance analysis (Section 5.2). The 2M-point and 32M-point data, together with the 8M-point data, are used for the comparison with the FFTW library on a general-purpose CPU (Section 5.3). All DSP execution times are converted from clock cycles assuming an operating frequency of 1.6 GHz.

5.1 Correctness Verification

A MATLAB program was used to verify the correctness of the FFT implementation. The input data and twiddle factors generated by MATLAB were loaded onto the YHFT-Matrix DSP processor to execute the FFT assembly program. The output was exported to a text file, which MATLAB then read and visualize, as shown in Fig. 11. Separately, MATLAB called its own FFT function using the same input data and visualized the computation results, as shown in Fig. 12. Since MATLAB is a widely used and well-validated software environment, its FFT output serves as the reference for verifying the assembly implementation. The proposed algorithm is considered correct if the relative error with respect to the output of MATLAB's FFT output is below 0.001. The correctness test results are shown in Table 3.

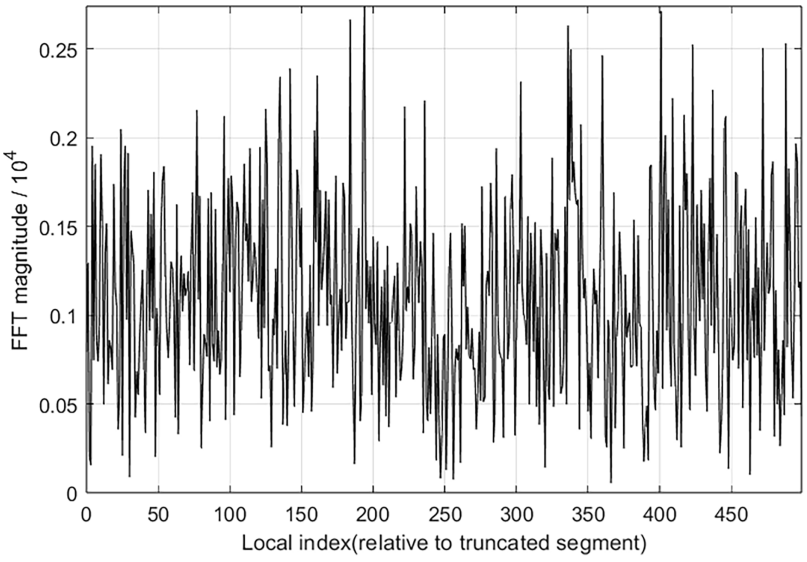


Figure 11: MCOPASM simulation results.

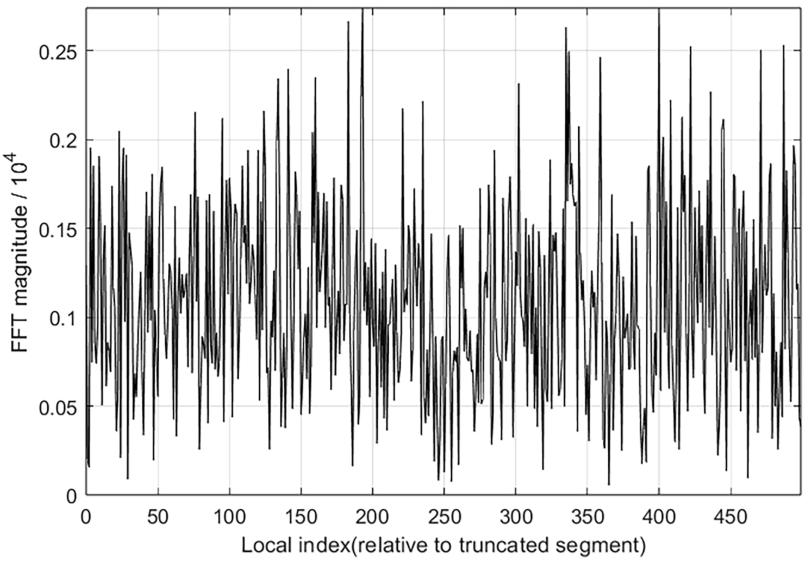


Figure 12: MATLAB simulation results.

Table 3: Correctness testing across output plot variants.

Version	Correctness
8M-point Single-Core Standard C Version (SCSC)	Correct
8M-point Single-Core Vectorized C Version (SCVC)	Correct
8M-point Single-Core Assembly Version (SCASM)	Correct
8M-point Single-Core Optimized Assembly Version (SCOPASM)	Correct
8M-point 4-Core Vectorized C Version (MCVC)	Correct
8M-point 4-Core Optimized Assembly Version (MCOPASM)	Correct

5.2 Performance Evaluation

The performance of the 8M-point FFT was tested on the YHFT-Matrix DSP processor. The results are shown in Table 4 and Fig. 13. It can be seen that the single-core vectorized C version achieves approximately 10× speed up over the standard C version, primarily due to the single-core vectorization strategy. The hand-written single-core assembly further improves performance by approximately 2.8× over the SCVC version, mainly benefiting from fine-grained memory management and instruction streamlining. From the single-core assembly version to the single-core optimized assembly version, performance improved by approximately additional 2.8×, primarily due to loop unrolling and software pipelining. From the single-core optimized assembly version to the multi-core optimized assembly version, performance improved by approximately 2×, mainly attributable to multi-core parallelism. Overall, the multi-core optimized assembly (MCOPASM) version is approximately 15.3× faster than the SCVC version.

Cycle counts are measured in processor clock cycles. The execution time is calculated as: Time (s) = Total Cycles/Clock Frequency (Hz). In the experiments, the operating frequency is 1.6 GHz (i.e., 1 cycle = 0.625 ns).

To quantify the components of the single-core optimized kernel, ablation experiments were performed on SCOPASM by selectively removing the DMA transfer operations and the computation instructions at the 8M-point scale. The cycle count values measured are as follows: for the intact SCOPASM algorithm, the cycle count is about 33.68M; when the DMA transfer operations are removed, the cycle count is about

4.02M; when the vector computation instructions are removed, the cycle count is about 30.87M. These results indicate that SCOPASM is therefore basically DMA-bound because of the deep joint optimization of the DMA transferring and computing. Because the DMA transfers share the DDR/GSM bandwidth among the four cores, MCOPASM achieves only 1.98× speedup over SCOPASM.

Table 4: Performance testing across different versions.

Version	Cycle
8M-point Single-Core Standard C Version (SCSC)	2,518,917,269
8M-point Single-Core Vectorized C Version (SCVC)	260,876,519
8M-point Single-Core Assembly Version (SCASM)	94,614,446
8M-point Single-Core Optimized Assembly Version (SCOPASM)	33,681,439
8M-point 4-Core Vectorized C Version (MCVC)	81,850,613
8M-point 4-Core Optimized Assembly Version (MCOPASM)	16,995,115

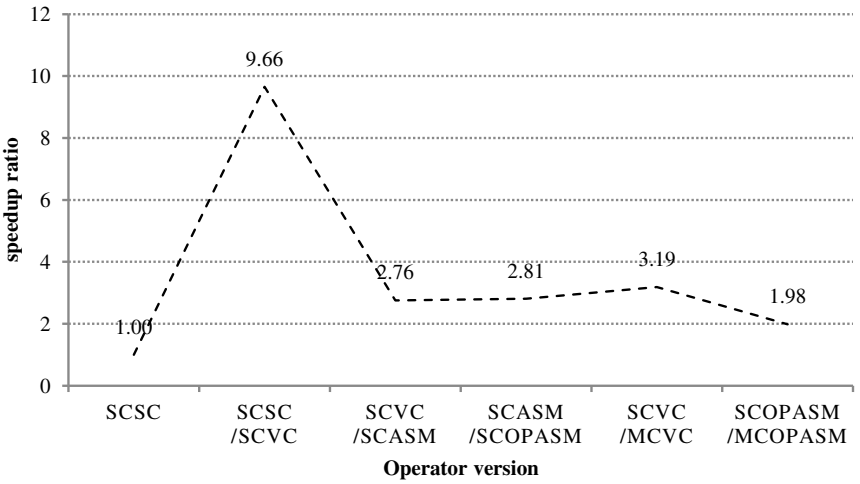


Figure 13: Speedup relative to the previous stage.

5.3 Comparison with FFTW on a General-Purpose CPU

To compare with a widely used external FFT implementation, 2M-point, 8M-point, and 32M-point single-precision complex FFT benchmarks were executed using the FFTW library on a general-purpose CPU (AMD Ryzen 5 7500F) with its single-precision API (fftwf). The DSP execution times are converted from the measured clock cycles at the 1.6 GHz operating frequency. Fig. 14 shows the execution times of the proposed DSP implementations alongside the single-threaded and multi-threaded FFTW plans at the three FFT sizes. All FFTW timings refer to the transform execution only and exclude plan-construction overhead.

It can be seen from Fig. 14 that, for every FFT size, the four-core optimized DSP implementation (MCOPASM) outperforms the multi-threaded FFTW library: by 1.45× at 2M points, 1.97× at 8M points, and 2.11× at 32M points. The single-core optimized DSP implementation (SCOPASM) is 2.81×, 2.91×, and 3.00× faster than the single-threaded FFTW at 2M, 8M, and 32M points, respectively, even though the Ryzen 5 7500F operates at a boost frequency of up to 5.0 GHz with much larger caches. The advantage of the DSP grows slightly with the FFT size, suggesting that its DMA/AM-based data-flow organization efficiently handles the large data volumes. It should be noted that the DSP and the Ryzen 5 7500F differ substantially

in architecture, i.e., the DSP is a specialized vector processor at 1.6 GHz, while the Ryzen 5 7500F is a general-purpose x86 CPU with a higher clock frequency and a different memory hierarchy. Thus, the above comparison just serves as an engineering reference, and the observed trend should not be taken as evidence of a general scalability advantage.

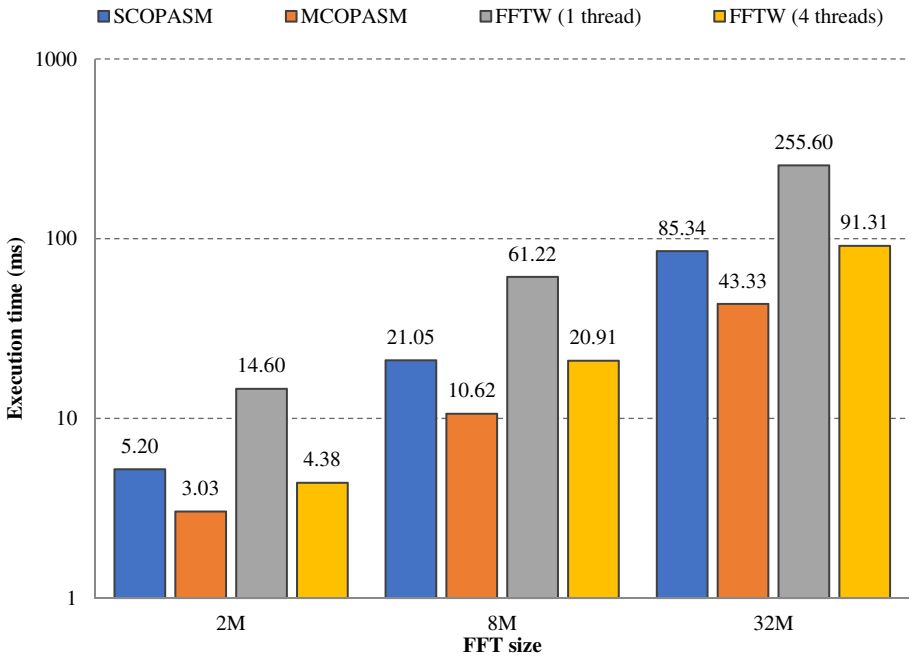


Figure 14: Execution times of the proposed DSP implementation and the FFTW library at different FFT sizes.

5.4 Portability Analysis

The experimental results in this section validate the effectiveness of the proposed method. However, it should be noted that this method is built upon specific architectural features of the target platform. The proposed optimization relies on a small set of architectural features, which defines its portability boundary. Among these features, two are essential. First, the transposition-free bit-reversal mechanism (Section 4.1.4) relies on DMA transfers driven by a pre-computed index table, so the DMA engines must support non-contiguous (indexed or strided) target addressing and multiple channels. Second, the vectorized butterfly computing kernels and the register-level shuffle packing (Section 4.1.3) require independent SIMD units with vector shuffle instructions. Notably, no dedicated bit-reversal or permutation hardware is required. The multi-core model further assumes hardware barriers or comparably low-latency synchronization primitives, and the double-buffering scheme assumes a small on-chip memory (AM) together with a larger on-chip buffer (GSM) accessible via by DMA.

For processors differs from the DSP platform used in our experiments, the proposed method requires appropriate portability adaptations depending on the actual hardware environment. For other processors that provide these features—SIMD vector units with shuffle support, multi-channel indexed DMA, on-chip caches, and hardware barriers—the proposed method can be ported by just retuning a small set of hardware-dependent parameters: the vector width, the capacities of AM and GSM, the number of DMA channels, and the core count. The method itself does not depend on the specific instruction set of the YHFT-Matrix. For platforms without indexed DMA, the bit-reversal will fall back to explicit transposition or in-place permutation at the cost of additional data movement. For platforms without in-vector shuffle instructions,

the packing in the last several final stages can be replaced by memory-based reordering, leading to some loss of efficiency.

6 Conclusion

This paper presents a hybrid optimization method for large-size FFT on the multi-core vector DSP. For cases in which the volume of input data far exceeds the on-chip memory capacity, the method combines mixed-radix (radix-4/radix-2) decomposition with a cross-layer co-design that jointly optimizes task partitioning, SIMD vectorization, data movement, and register-level data shuffling.

Combining the computational characteristics of large-size FFT and hardware architecture features, there are three main mechanisms in this method: a transposition-free DMA bit-reversal scheme that fuses output reordering into the DMA write-back phase via a pre-computed global index table and multi-channel concurrent scattered writes; a register-level shuffle-packing strategy that combines multiple small butterfly groups into fully occupied vector registers and reuses radix-4 shuffle templates for radix-2 butterflies, keeping the SIMD units fully utilized in the final stages; and a static, deterministic multi-core mapping with hardware-barrier-only synchronization that eliminates explicit inter-core communication and runtime scheduling overhead. Together with multi-level (DDR/GSM/AM) double buffering that overlaps computation and DMA transfers, these techniques yield an end-to-end optimized implementation.

The engineering-feasible implementation path for deterministic large-size signal processing of our method is provided for the YHFT-Matrix platform, by which the MCOPASM version can achieve a 15.3× speedup over the single-core vectorized C baseline. Future work will focus on alleviating the data-movement bottleneck in extending the framework to two- and three-dimensional FFTs, non-power-of-two sizes, and emerging multi-core vector DSPs.

Acknowledgement: Not applicable.

Funding Statement: National Science and Technology Major Project (No. 2022ZD0119003) and Hunan Provincial Natural Science Foundation (No. 2023JJ50019).

Author Contributions: Junyang Tang: software, validation, manuscript writing. Yonghua Hu: conceptualization, methodology, supervision, review and editing. Yuxiang Gao: investigation, algorithm design, data curation, manuscript writing. Jie Luo: resources, formal analysis. All authors reviewed and approved the final version of the manuscript.

Availability of Data and Materials: All experimental results presented in this study are included within the article. The source code used for the implementation is available from the corresponding author upon reasonable request.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Adnan A, Liu Y, Chow CW, Yeh CH. Demonstration of non-Hermitian symmetry (NHS) IFFT/FFT size efficient OFDM non-orthogonal multiple access (NOMA) for visible light communication. IEEE Photonics J. 2020;12(3):7201405. doi:10.1109/JPHOT.2020.2984564.
2. Cen S, Kim DO, Lim CG. A fused CNN-LSTM model using FFT with application to real-time power quality disturbances recognition. Energy Sci Eng. 2023;11(7):2267–80. doi:10.1002/ese3.1450.
3. Sahnoun K, Benabadji N. Satellite image compression algorithm based on the FFT. Int J Multimed Its Appl. 2014;6(1):77–83. doi:10.5121/ijma.2014.6106.

4. Bilal A, Imran A, Baig TI, Liu X, Long H, Alzahrani A, et al. DeepSVDNet: a deep learning-based approach for detecting and classifying vision-threatening diabetic retinopathy in retinal fundus images. Comput Syst Sci Eng. 2024;48(2):511–28. doi:10.32604/csse.2023.039672.
5. Kim BS, Jin Y, Lee J, Kim S. High-efficiency super-resolution FMCW radar algorithm based on FFT estimation. Sensors. 2021;21(12):4018. doi:10.3390/s21124018.
6. Jagadesh T, Rani BS. Time delay estimation in radar system using fuzzy based iterative unscented Kalman filter. Comput Syst Sci Eng. 2023;44(3):2569–83. doi:10.32604/csse.2023.027239.
7. Chen Y, Feng S, Zhao C, Su N, Li W, Tao R, et al. High-resolution remote sensing image change detection based on Fourier feature interaction and multiscale perception. IEEE Trans Geosci Remote Sens. 2024;62:5539115. doi:10.1109/TGRS.2024.3500073.
8. Song H. FST-EfficientNetV2: exceptional image classification for remote sensing. Comput Syst Sci Eng. 2023;46(3):3959–78. doi:10.32604/csse.2023.038429.
9. Likhachev SF, Kostenko VI, Girin IA, Andrianov AS, Rudnitskiy AG, Zharov VE. Software correlator for radioastron mission. J Astron Instrum. 2017;6(3):1750004. doi:10.1142/s2251171717500040.
10. Clark MA, Plante PL, Greenhill LJ. Accelerating radio astronomy cross-correlation with graphics processing units. Int J High Perform Comput Appl. 2013;27(2):178–92. doi:10.1177/1094342012444794.
11. Zhang L, Zhang Y, Su Y, Wu Z, Yan F. A new method for improving time and frequency resolution of passive receiving radar system. IEEE Trans Ind Inform. 2024;20(11):13416–28. doi:10.1109/TII.2024.3435563.
12. Salahuddin, Aslam MF, Yasin A, Aslam M, Latif A. Investigating the scalability of FFT algorithms in contemporary parallel computing environments. Spectr Eng Sci. 2025;3(1):183–214.
13. Kieu-do-Nguyen B, Nguyen K-D, The Binh N, Ma KM, Ta TD, Le DH, et al. Hardware software co-design for multi-threaded computation on RISC-V-based multicore system. IEEE Access. 2024;12:177312–26. doi:10.1109/ACCESS.2024.3505940.
14. Zhan Y, Hu X, Guo Y. Two-dimensional FFT parallel implementation and optimization on FT-X DSP platform. Microelectron Comput. 2023;40(2):71–8. doi:10.19304/J.ISSN1000-7180.2022.0360.
15. Guo J, Liu P, Yang X, Zhang L, Wu D. Parallel optimization of large-point FFT on Sunway 26010. J Zhejiang Univ. 2024;58(1):78–86. (In Chinese).
16. Xia J, Liang Z, Cui L. Design of multi core parallel large point FFT and IFFT. Fire Control Radar Technol. 2016;45(1):64–8. (In Chinese). doi:10.19472/j.cnki.1008-8652.2016.01.012.
17. Wang D, Shi S, Wu T, Liu L, Tan H, Hao Z, et al. A high performance accelerator design for ultra-long point floating-point FFT. J Comput Res Dev. 2021;58(6):1192–203. doi:10.7544/issn1000-1239.2021.20210069.
18. Wang Y, Li C, Liu C, Liu S, Lei Y, Zhang J, et al. Advancing DSP into HPC, AI, and beyond: challenges, mechanisms, and future directions. CCF Trans High Perform Comput. 2021;3(1):114–25. doi:10.1007/s42514-020-00057-2.
19. Cooley JW, Tukey JW. An algorithm for the machine calculation of complex Fourier series. Math Comp. 1965;19(90):297–301. doi:10.1090/s0025-5718-1965-0178586-1.
20. Ghose S, Hsieh K, Boroumand A, Ausavarungnirun R, Mutlu O. Enabling the adoption of processing-in-memory: challenges, mechanisms, future research directions. arXiv:1802.00320. 2018.
21. Popovici DT, Franchetti F, Low TM. Mixed data layout kernels for vectorized complex arithmetic. In: Proceedings of the 2017 IEEE High Performance Extreme Computing Conference (HPEC); 2017 Sep 12–14; Waltham, MA, USA. p. 1–7. doi:10.1109/HPEC.2017.8091024.
22. Hennessy JL, Patterson DA. Computer architecture: a quantitative approach. Amsterdam, The Netherlands: Elsevier; 2011.
23. Frigo M, Johnson SG. The design and implementation of FFTW3. Proc IEEE. 2005;93(2):216–31. doi:10.1109/jproc.2004.840301.