This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2020.2977629, IEEE Transactions on Parallel and Distributed Systems

Automatic Generation of High-Performance FFT Kernels on Arm and x86 CPUs

Zhihao Li, Haipeng Jia, Yunquan Zhang, *Member, IEEE,*
Tun Chen, Liang Yuan, and Richard Vuduc, *Member, IEEE,*

Abstract—This paper presents AutoFFT, a template-based code generation framework that can automatically generate high-performance FFT kernels for all natural-number radices. AutoFFT is based on the Cooley-Tukey FFT algorithm, which exploits the symmetric and periodic properties of the DFT matrix, as the outer parallelization framework. Because butterflies are the core operations of the Cooley-Tukey algorithm, we explore additional symmetric and periodic properties of the DFT matrix and formulate multiple optimized calculation templates to further reduce the number of floating-point operations for butterflies of arbitrary natural numbers. To fully exploit hardware resources, we encapsulate a series of optimizations in an assembly template optimizer. Given any DFT problem, AutoFFT automatically generates C FFT kernels using these calculation templates and converts them into efficient assembly kernels using the template optimizer. Through a series of experiments on Arm, Intel, and AMD processors, we show that AutoFFT-generated kernels can outperform those in Fastest Fourier Transform in the West (FFTW), the Arm Performance Libraries (ARMPL), and the Intel Math Kernel Library (MKL).

Index Terms—AutoFFT, FFT, code generation, template, DFT.

1 Introduction

The discrete Fourier transform (DFT) is a basic discrete transform used to perform Fourier analysis. The definition of the DFT is presented in Eq.1:

$$Y_k = \sum_{i=0}^{n-1} x_i W_n^{ik} = \sum_{i=0}^{n-1} x_i \cdot e^{-\frac{2\pi j}{n} ik} \tag{1}$$

where x is the input sequence of n complex numbers, Y is the corresponding output sequence, $k \in [0, n-1]$, $j = \sqrt{-1}$, and $W_n = e^{-\frac{2\pi j}{n}}$, which is called the twiddle factor (twiddle for short). The DFT matrix $(W_n^{ik})_{n\times n}$ is an expression of the DFT as a transformation matrix, as presented in Eq.2. The DFT can be represented by applying the DFT matrix to the input vector through matrix multiplication. From this point of view, the DFT is essentially a matrix-vector operation, and the computational complexity of the naïve matrix-vector implementation is $O(n^2)$.

$$(W_n^{ik})_{n\times n} = \begin{pmatrix} W_n^0 & W_n^0 & \dots & W_n^0 \\ W_n^0 & W_n^1 & \dots & W_n^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ W_n^0 & W_n^{n-1} & \dots & W_n^{(n-1)(n-1)} \end{pmatrix} \tag{2}$$

- *Zhihao Li is with the SKL of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences; University of Chinese Academy of Sciences; and Georgia Institute of Technology. E-mail: lizhihao@ict.ac.cn*
- *Haipeng Jia (Corresponding author), Yunquan Zhang, Tun Chen, and Liang Yuan are with the SKL of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences. E-mails: {jiahaipeng, zyq, chentun, yuanliang}@ict.ac.cn*
- *Richard Vuduc is with the School of Computational Science and Engineering, Georgia Institute of Technology. E-mails: richie@cc.gatech.edu*

Manuscript received April 19, 2005; revised August 26, 2015.

Many previous studies [1], [2], [3], [4], [5], [6] reduce the computational complexity of the DFT from $O(n^2)$ to $O(n \log n)$ by exploiting the symmetric and periodic properties of the DFT matrix. The Cooley-Tukey algorithm [1] is the most widely used fast Fourier transform (FFT) algorithm in many practical applications [7], [8], [9], [10], [11], [12], [13]. It adopts a divide-and-conquer approach to recursively break down a large DFT into smaller DFTs, achieving an $O(n \log n)$ complexity. For $N = r^v$ with integers N, r, and v, r is called the radix. We present the derivation of the radix-2 FFT in Eq.3.

$$\begin{aligned} Y_k &= \sum_{i=0}^{n-1} x_i W_n^{ik} \\ &= \sum_{i=0}^{n/2-1} x_{2i} W_n^{2ik} + \sum_{i=0}^{n/2-1} x_{2i+1} W_n^{(2i+1)k} \\ &= \sum_{i=0}^{n/2-1} f_i \cdot W_{n/2}^{ik} + W_n^k \sum_{i=0}^{n/2-1} g_i \cdot W_{n/2}^{ik} \end{aligned} \tag{3}$$

The radix-2 FFT divides a large DFT of size n into two smaller DFTs of size $n/2$ in each recursive stage until it obtains indivisible DFTs of size 2 (hence the name "radix-2 FFT"). A DFT of size 2 is also called a radix-2 butterfly. Because the butterflies are critical to the FFT computation, in this paper, we define the butterfly kernel to calculate one butterfly, which is the core computing unit of the FFT computation, and define the FFT kernel to calculate multiple butterflies (described in subsection 5.2).

The performance of the FFT implementation is restricted by its computational complexity and is dependent on the effectiveness of finer optimizations. Therefore, this paper proposes a template-based code generation framework, named AutoFFT, that autogenerates optimized FFT kernels

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2020.2977629, IEEE Transactions on Parallel and Distributed Systems

in assembly by capitalizing on the symmetric and periodic properties of the DFT matrix.

First, we summarize and extract two novel and highly optimized patterns (the pair and quad patterns) with a reduced number of floating-point operations for butterflies in the following steps: 1) For the radix-r butterfly, we investigate the symmetric properties in the horizontal and vertical directions for radices of all natural numbers, which are divided into three cases ($r = 2m + 1$, $r = 4m + 2$, and $r = 4m + 4$). By observing the calculation of each output Y_k, Y_k's twiddles are symmetric in the horizontal direction, and we reduce the multiplications for non-power-of-two and power-of-two radices by factors of 2 and 4, respectively. In the vertical direction, some outputs share a similar symmetric feature, so we can further reduce the arithmetic complexity for non-power-of-two and power-of-two radices by factors of 2 and 4, respectively. 2) We study a periodic property inside one Y_k for these three cases. By exploiting this property, the arithmetic complexity can be further reduced by a factor of $gcd(k, r)$, where gcd is the greatest common divisor.

Second, AutoFFT generates high-performance FFT kernels based on the pair and quad patterns. In the beginning, the computational template designer takes these two highly optimized patterns as inputs and formalize them into computational templates. Then these computational templates are used by a C FFT kernel generator to generate C FFT kernels according to the derived equations of these two patterns. Subsequently, an assembly template optimizer, which is composed of the optimization templates and the register allocation strategy, is introduced to transform C kernels into high-performance assembly FFT kernels for varying architectures. Considering AutoFFT is built on templates, which are the abstractions of typical calculation patterns in FFT calculations, we refer to it as the template-based code generation framework, which is the first framework that can directly generate high-performance assembly FFT kernels by taking advantage of the optimized pair and quad patterns.

At runtime, to obtain the best FFT plan of a given DFT problem, AutoFFT adopts a pruning-based dynamic programming approach to empirically search for the optimal plan. The optimal plan contains the necessary parameters that are used to construct the corresponding butterfly network of the best plan and invokes the required FFT kernels to perform the FFT computation. The experiments show that AutoFFT outperforms the current state-of-the-art, such as the Fastest Fourier Transform in the West (FFTW) [14], [15], [16], ARM Performance Library (ARMPL) [17], and Intel Math Kernel Library (MKL) [18], [19].

The key contributions and innovations of this paper are summarized as follows:

- This paper is the first work that systematically summarizes and extracts the integral and general mathematical expressions for the symmetric and periodic properties of the DFT matrix. These mathematical expressions minimize the number of floating-point operations for all natural-number radices: the pair pattern for odd numbers and the quad pattern for even numbers. Other libraries can also accelerate their butterflies based on these two optimized patterns.
- We propose a template-based code generation framework, which is the first framework to be able to automatically generate high-performance assembly FFT kernels based on the highly optimized pair and quad patterns for varying CPU architectures.
- Based on our code generation framework, we realize a high-performance FFT library for ARMv8, Intel Haswell, and AMD Zen CPUs. This library is on average 2.14, 2.15, and 1.7 times faster than FFTW, ARMPL, and Intel MKL, respectively.

This paper extends a conference version [20]. In particular, it adds: 1) compared to the conference version, which can generate FFT kernels of prime-number and power-of-two radices, this paper extends the generality of the framework and is able to generate kernels for all natural numbers; 2) this paper ports AutoFFT to AMD Zen CPUs and supplements the experimental results; and 3) this paper illustrates the implementations and optimizations adopted in AutoFFT's butterfly network.

The remainder of this paper is organized as follows. Section 2 presents related studies, and section 3 provides an overview of the AutoFFT framework. Section 4 deduces the pair and quad patterns, and section 5 illustrates how AutoFFT autogenerates high-performance assembly FFT kernels based on these two patterns. Section 6 gives the implementations and optimizations of our butterfly network, and section 7 presents the experimental results. Finally, section 8 concludes the paper.

2 Related Work

Many FFT algorithms [1], [2], [3], [4], [5], [6], [21], [22] have been proposed to compute the DFT. As the most popular FFT algorithm, the Cooley-Tukey algorithm [1] is supported by most mainstream FFT libraries. These libraries are optimized by vendors or researchers to achieve high performance on specific hardware architectures.

These highly efficient vendor-supplied FFT libraries, which include ARMPL [17], Intel MKL [18], [19], AMD Optimizing CPU Libraries (AOCL) [23], IBM Engineering and Scientific Subroutine Library (ESSL) [24], and Apple vDSP [25], are appropriately optimized using single instruction multiple data (SIMD) techniques for specific architectures. These implementations have undergone extensive architecture-dependent tuning on specific microarchitecture features to pursue peak system performance.

Because these vendor-specific libraries cannot be ported to processors from other vendors, many excellent auto-tuning systems have been developed. FFTW [14], [15], [16] and UHFFT [26], [27] conduct tuning in two stages. In the install-time stage, the special-purpose compiler generates several highly optimized instruction set architecture (ISA)-specific SIMD codelets to boost performance. In the runtime stage, its adaptive framework creates and empirically chooses the optimal plan according to the input parameters and hardware features. The optimal plan organizes and assembles required pregenerated codelets to perform FFTs that involve large DFT problems. SPIRAL [28], [29]

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2020.2977629, IEEE Transactions on Parallel and Distributed Systems

is a three-stage framework. In the first stage, SPIRAL expresses the mathematical formulas in the Signal Processing Language (SPL) [22] for a given transform. The second stage converts the SPL formulas into Σ-SPL [30], and then performs various computational optimizations such as code reordering in Σ-SPL. Finally, the optimized Σ-SPL is translated into high-level code (SIMD intrinsics) suitable for the given architecture [29]. Other open-source FFT implementations include Ne10 [31], which contains FFT kernels that have been heavily optimized for ARM-based CPUs equipped with NEON SIMD capabilities, the Fastest Fourier Transform in the East (FFTE) [32], which is a Fortran FFT package that supports problem sizes of the form $2^a3^b5^c$ for CPUs and GPUs, and the Fastest Fourier Transform in the South (FFTS) [33], which is a hand-optimized library for Intel and ARM CPUs.

Significant progress has been made over the past decade in the fields of code generation and automatic performance tuning on GPUs. In addition to high-performance vendor-tuned FFT libraries, such as cuFFT for NVIDIA GPUs [34], clFFT for AMD GPUs [35], and genFFT for Intel processor graphics [36], researchers have developed several FFT implementations [37], [38], [39], [40], [41]. Although the factors that affect the FFT performance on GPUs can be quite different than those on CPUs, most GPU frameworks are similar to FFTW: 1) autogenerating or manually writing optimized FFT kernels that exploit the underlying hardware features, such as GPU memory hierarchy and warp/wavefront scheduling, and 2) searching for the optimal plan to exploit the available computational resources for a given transform.

According to the definition of the twiddle factor, the symmetric and periodic properties of the DFT matrix are inherited from trigonometric functions, such as the sine and cosine functions. Many FFT libraries, such as FFTW, Intel MKL, ARMPL, SPIRAL, Ne10, and so on, utilize these properties more or less to optimize their FFT kernels, especially for power-of-two radices. However, according to the public materials of these libraries, none of them has provided integral and general mathematical expressions to combine the symmetry and periodicity for all radices of natural numbers. Besides, different from other FFT libraries, AutoFFT is built on the templates, which are the abstractions of typical calculation patterns in FFT calculations. These templates are defined based on the optimized pair and quad patterns and used to auto-generate low-level assembly FFT kernels. FFTW's codelets are some fundamental kernels used to calculate small DFTs. So FFTW's codelets are conceptually equivalent to AutoFFT's FFT kernels, which means that codelets and templates belong to different granularities (templates are in finer granularity than codelets). Compared with the prior work, this paper contains the following three contributions. First, this paper is the first work that systematically summarizes and extracts the integral mathematical expressions of the symmetric and periodic properties (the pair and quad patterns) to minimize the arithmetic complexity for all radices. Second, we formalize these two patterns into templates so that these two patterns can be applied in the code generation. Third, we build a novel framework AutoFFT, which is based on the template methodology, to enable the assembly code generation. Benefiting from the template-based code generation mechanism, AutoFFT is able to directly generate efficient assembly FFT kernels for varying architectures without compiler intervention.

This paper compares AutoFFT with FFTW, ARMPL, and Intel MKL on ARM, Intel, and AMD CPU platforms. ARMPL and Intel MKL are the official math libraries for ARM and Intel processors, respectively. These two vendor-tuned libraries are well-optimized to achieve high performance. The performance of FFTW is typically superior to that of other publicly available FFT libraries and is even competitive with vendor-tuned libraries. In contrast to vendor-tuned libraries, FFTW's performance is portable, which means that FFTW performs well on most architectures. According to AMD official description in [23], its official math library AOCL regards FFTW as its FFT module.

3 The AutoFFT Framework

This section introduces the two stages of AutoFFT, as shown in Fig.1. The install-time stage generates efficient FFT kernels, and the runtime stage empirically searches for the optimal plan to construct an efficient butterfly network for a given FFT size and the underlying hardware features.

3.1 The Install-Time Stage

The install-time stage is responsible for generating high-performance FFT kernels that are defined to calculate multiple butterflies. To reduce the number of floating-point operations of different butterflies as many as possible, the pair and quad patterns are derived from the symmetric and periodic properties of the DFT matrix $(W_r^{ik})_{r\times r}$. These two patterns are the foundation of the code generation framework. The details of how to deduce these two patterns will be given in section 4. The install-time stage adopts three components to capitalize on these two patterns.

First, the computational template designer. To make full use of the pair and quad patterns, a computational template designer is introduced. This designer takes these two highly optimized patterns as inputs and then obtains high-level computational templates as outputs. The computational templates are composed of meta templates and hybrid templates. Meta templates are predefined C preprocessor macros that represent some basic complex number operations. Unlike meta templates, hybrid templates are strongly related to radix r. For different radix r, the number and type of meta templates used by hybrid templates are different. Hence, hybrid templates are defined and extracted as C functions according to the derived equations of the pair and quad patterns.

Second, the C FFT kernel generator. This generator takes the computational templates as inputs and then enables the automatic generation of high-level FFT kernels with the lowest arithmetic complexity for all radices according to Alg.1, which is the specific implementation of the pair and quad patterns.

Third, the assembly optimizer. To further exploit hardware resources, The optimizer defines optimization templates that match the underlying hardware capabilities, such as SIMD vectorization, register resources, and the instruction set. In addition, it also categorizes small/medium/large radices (described in subsubsection 5.3.3). After that, the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2020.2977629, IEEE Transactions on Parallel and Distributed Systems

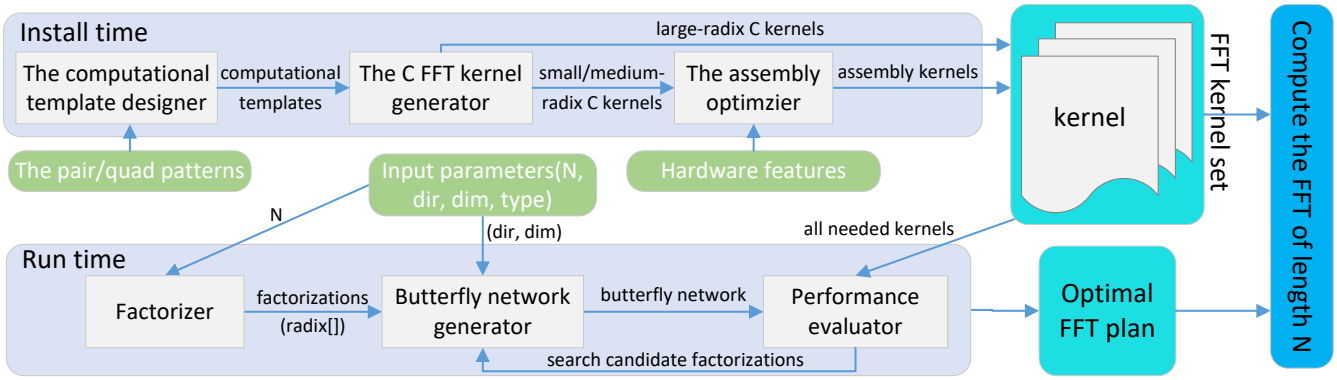


Fig. 1: Overview of AutoFFT.

optimizer takes the C FFT kernels and optimization templates as inputs and transforms the C FFT kernels into high-performance assembly FFT kernels according to the register allocation strategy.

Due to the limited register resources in modern CPUs, it is difficult to transform the C FFT kernels of large radices into efficient assembly code. Hence, AutoFFT can directly generate assembly FFT kernels for small and medium radices without compiler intervention and generate C FFT kernels for large radices. These kernels make up our FFT kernel set. As the computational core of FFT computation, these kernels have a crucial impact on the overall performance and will be invoked as needed at runtime.

3.2 The Runtime Stage

According to the Cooley-Tukey algorithm [1], a given transform size is recursively factorized into smaller sizes until they can no longer be decomposed. When the given size is large, the size decomposition tree can be large too. Because each branch of this tree represents a factorization for the given FFT size, we call this decomposition tree as the factorization tree, and each branch of the factorization tree is also known as an FFT plan. The runtime stage is responsible for reducing the search space of many potential FFT plans and seeking the optimal plan for a given DFT problem. The runtime stage consists of three components.

First, the number factorizer. The factorizer takes the FFT size as input, and then recursively factorizes the given FFT size and builds the factorization tree. As a result, it outputs many potential FFT plans. Each FFT plan represents one factorization that holds the radix for each stage of the butterfly network in $radix[]$.

Second, the butterfly network generator. AutoFFT adopts the Stockham auto-sort FFT [42], [43] network, which is SIMD-friendly and requires no explicit bit-reversal permutation. The generator takes the necessary input parameters ($radix[]$, the transform direction, and dimension) to construct the butterfly network.

Third, the performance evaluator. Because the factorization tree can be large, the evaluator adopts a depth-first search to identify the shortest path of the tree and prune unneeded branches. Because the Cooley-Tukey algorithm is a memory-intensive algorithm, a branch with a shorter path indicates that its network contains fewer stages, which may lead to more efficient memory accesses. Subsequently, a bottom-up dynamic programming method is adopted on the pruned tree by making full use of the recursive structure of the FFT. We build a performance table to record the minimum execution time of the subsequences with various input and output strides, so we do not need to reevaluate the same subsequences with the same strides. Finally, the evaluator evaluates all candidate factorizations to determine the best factorization.

After obtaining the optimal plan, we construct the Stockham butterfly network according to the best factorization and call the needed FFT kernels to efficiently solve the given DFT problem. The auto-tuning framework of AutoFFT refers to the current state-of-the-art [16], [27], [39], and this paper does not discuss the runtime stage in detail but rather focuses on the FFT kernel generation.

4 Optimized Calculation Patterns

This section focuses on the algorithmic optimizations on butterflies. Performing algorithmic optimizations on butterflies and extracting general calculation patterns can bring two great benefits. First, because butterflies are the core operations of the Cooley-Tukey algorithm, reducing the floating-point operations of the butterfly kernels can significantly enhance the overall performance. Second, these calculation patterns regularize the calculation of the butterfly kernels, facilitating and enabling the code generation process. In this section, we systematically summarize the integral and general mathematical expressions for the symmetric and periodic properties and extract relatively optimized calculation patterns for radices of all natural numbers.

In our previous work [20], we empirically divide radix r into two cases from the perspective of engineering requirements: 1) r is a prime number and 2) r is a power of two. However, this division manner of r limits the generality of the code generation framework, which means the framework does not support generating butterfly kernels for radices beyond these two cases, such as radices 6, 9, 10, 12, and so on. However, a native radix may be faster than mixed radices. For example, radix 6 can be faster than the combination of radix 2 and radix 3 when processing FFTs of power-of-six sizes.

In this paper, we improve and supplement the generality of AutoFFT based on the previous work [20] so that it can

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2020.2977629, IEEE Transactions on Parallel and Distributed Systems

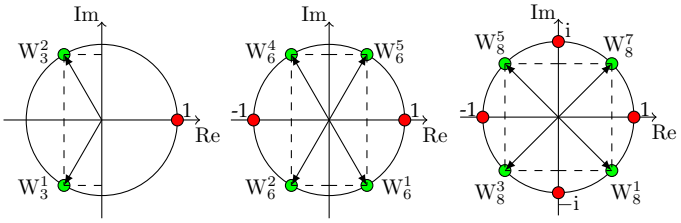
generate butterfly kernels for all natural numbers. Here, the natural numbers are divided into three cases: $r = 2m + 1$, $r = 4m + 2$, and $r = 4m + 4$. To simplify the representation of the formulas in this section, we introduce the following symbols.

$$\begin{aligned} c_i =& x_i + x_{r-i}, \text{ and } d_i = x_i - x_{r-i} \\ u_i =& x_{r/2-i} + x_{r/2+i}, \text{ and } v_i = x_{r/2-i} - x_{r/2+i} \end{aligned}$$

4.1 The Symmetry Property

4.1.1 The $r = 2m + 1$ Radices

Let $r = 2m + 1$. In this case, twiddles are symmetric about the x-axis. Fig.2(a) presents twiddles' characteristics of radix 3, which complies with $r = 2m + 1$.



(a) Radix 3 (b) Radix 6 (c) Radix 8

Fig. 2: These three subfigures present the symmetries of twiddles of (a) radix 3, (b) radix 6, and (c) radix 8, respectively. The red points should be processed separately and the green points contain good symmetry.

According to Eq.1, when calculating Y_0 for the radix-r butterfly, because $k = 0$, the following equality is always true: $W_r^{ik} = W_r^0 = 1$. Thus, Y_0 is calculated separately using Eq.4.

$$Y_0 = x_0 + \sum_{i=1}^{m} c_i \tag{4}$$

For the remaining outputs Y_k with $k \in [1, r-1]$, we minimize the number of floating-point operations by taking advantage of the horizontal and vertical symmetries of the DFT matrix.

Regarding the horizontal symmetry, W_r^{ik} and $W_r^{(r-i)k}$ are symmetric about the x-axis, and $W_r^{(r-i)k} = W_r^{-ik}$ which means that $Re(W_r^{ik}) = Re(W_r^{-ik})$ and $Im(W_r^{ik}) = -Im(W_r^{-ik})$. We refer to this as the horizontal symmetry since it exploits the reuse along each row in Eq.2. For each Y_k, we define $W_r^{ik} = a_{ik} + b_{ik} \cdot j$ and $W_r^{(r-i)k} = W_r^{-ik} = a_{ik} - b_{ik} \cdot j$. Based on this symmetry, the calculation of Y_k can be reduced by uniting the like terms $x_i W_r^{ik}$ and $x_{r-i} W_r^{(r-i)k}$, as presented in Eq.5.

$$\begin{aligned} Y_k =& \sum_{i=0}^{r-1} x_i W_r^{ik} = x_0 + \sum_{i=1}^{r-1} x_i W_r^{ik} \\ =& x_0 + \sum_{i=1}^{m} c_i \cdot a_{ik} + \sum_{i=1}^{m} d_i \cdot b_{ik} \cdot j \end{aligned} \tag{5}$$

Regarding the vertical symmetry, by observing Eq.5 and Eq.6, we find that Y_k and Y_{r-k} are like terms. We refer to this as the vertical symmetry since it exploits the reuse between rows in Eq.2. Similar to Eq.5, Y_{r-k} can be calculated using Eq.6. Accordingly, we further reduce the number of floating-point operations by uniting the like terms and calculating Y_k and Y_{r-k} together.

$$\begin{aligned} Y_{r-k} =& \sum_{i=0}^{r-1} x_i W_r^{i(r-k)} \\ =& x_0 + \sum_{i=1}^{r-1} x_i W_r^{ir-ik} \\ =& x_0 + \sum_{i=1}^{r-1} x_i W_r^{-ik} \\ =& x_0 + \sum_{i=1}^{m} c_i \cdot a_{ik} - \sum_{i=1}^{m} d_i \cdot b_{ik} \cdot j \end{aligned} \tag{6}$$

4.1.2 The $r = 4m + 2$ Radices

Let $r = 4m + 2$. In this case, twiddles are symmetric about the x-axis and the y-axis. Fig.2(b) presents the symmetric characteristics of radix-6 twiddles, which complies with $r = 4m + 2$.

For Y_0 and $Y_{r/2}$, all required twiddles can be simplified to $W_r^0 = 1$ and $W_r^{r/2} = -1$; therefore, Y_0 and $Y_{r/2}$ are calculated separately using Eq.7.

$$\begin{aligned} Y_0 =& \sum_{i=0}^{r-1} x_i W_r^{i \cdot 0} = \sum_{i=0}^{r/2-1} (x_i + x_{r-i-1}) \\ Y_{r/2} =& \sum_{i=0}^{r-1} x_i W_r^{i \cdot r/2} = \sum_{i=0}^{r/2-1} (x_{2i} - x_{2i+1}) \end{aligned} \tag{7}$$

Compared with the $r = 2m + 1$ case, the $r = 4m + 2$ case has better symmetry in both horizontal and vertical directions.

Regarding the horizontal symmetry, when $r = 4m + 2$, W_r^{ik}, $W_r^{(r-i)k}$, $W_r^{(r/2-i)k}$, and $W_r^{(r/2+i)k}$ are symmetric to some extent. Because $W_r^{(r/2-i)k} = (-1)^k W_r^{-ik}$ and $W_r^{(r/2+i)k} = (-1)^k W_r^{ik}$, we have $W_r^{ik} = a_{ik} + b_{ik} \cdot j$, $W_r^{(r-i)k} = a_{ik} - b_{ik} \cdot j$, $W_r^{(r/2-i)k} = (-1)^k (a_{ik} - b_{ik} \cdot j)$ and $W_r^{(r/2+i)k} = (-1)^k (a_{ik} + b_{ik} \cdot j)$. Y_k's calculation can be reduced by uniting these like terms $x_i W_r^{ik}$, $x_{r/2-i} W_r^{(r/2-i)k}$, $x_{r/2+i} W_r^{(r/2+i)k}$ and $x_{r-i} W_r^{(r-i)k}$, as presented in Eq.8 where $k \in [1, m]$.

$$\begin{aligned} Y_k =& x_0 + (-1)^k x_{r/2} \\ &+ \sum_{i=1}^{m} (c_i + (-1)^k u_i) a_{ik} \\ &+ \sum_{i=1}^{m} (d_i - (-1)^k v_i) b_{ik} \cdot j \end{aligned} \tag{8}$$

Regarding the vertical symmetry, Y_{r-k}, $Y_{r/2-k}$, and $Y_{r/2+k}$ can be calculated using Eq.9. Because the four outputs Y_k, Y_{r-k}, $Y_{r/2-k}$, and $Y_{r/2+k}$ are like terms, we optimize the calculation by uniting the like terms and calculating these four results together.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2020.2977629, IEEE Transactions on Parallel and Distributed Systems

$$
\begin{aligned}
Y_{r-k} =& x_0 + (-1)^k x_{r/2} \\
&+ \sum_{i=1}^{m}(c_i + (-1)^k u_i)a_{ik} - \sum_{i=1}^{m}(d_i - (-1)^k v_i)b_{ik} \cdot j \\
Y_{r/2-k} =& x_0 + (-1)^{k+1} x_{r/2} + \sum_{i=1}^{m}(-1)^i(c_i - (-1)^k u_i)a_{ik} \\
&- \sum_{i=1}^{m}(-1)^i(d_i + (-1)^k v_i)b_{ik} \cdot j \\
Y_{r/2+k} =& x_0 + (-1)^{k+1} x_{r/2} + \sum_{i=1}^{m}(-1)^i(c_i - (-1)^k u_i)a_{ik} \\
&+ \sum_{i=1}^{m}(-1)^i(d_i + (-1)^k v_i)b_{ik} \cdot j
\end{aligned} \tag{9}
$$

4.1.3 The $r = 4m + 4$ Radices

Let $r = 4m + 4$. The calculation pattern for the $r = 4m + 4$ radices is very similar to the $r = 4m + 2$ case, where the twiddles are symmetric about the x-axis and the y-axis. Their main difference is that the former contains extra two twiddles that are always symmetric about the x or y axis, such as $W_r^{r/4 \cdot k}$ and $W_r^{3r/4 \cdot k}$ in Y_k's calculation. Hence, Y_0, $Y_{r/2}$, $Y_{r/4}$, and $Y_{3r/4}$ can be calculated separately. The calculations of Y_0 and $Y_{r/2}$ remain the same as the $r = 4m + 2$ case using Eq.7. Regarding $Y_{r/4}$, and $Y_{3r/4}$, they are calculated using Eq.10. Fig.2(c) presents twiddles' characteristics of radix 8, which complies with $r = 4m + 4$.

$$
\begin{aligned}
Y_{r/4} =& \sum_{i=0}^{r-1} x_i W_r^{i \cdot r/4} \\
=& \sum_{i=0}^{r/4-1} ((x_{4i} - x_{4i+2}) - (x_{4i+1} - x_{4i+3}) \cdot j) \\
Y_{3r/4} =& \sum_{i=0}^{r-1} x_i W_r^{i \cdot 3r/4} \\
=& \sum_{i=0}^{r/4-1} ((x_{4i} - x_{4i+2}) + (x_{4i+1} - x_{4i+3}) \cdot j)
\end{aligned} \tag{10}
$$

The horizontal and vertical symmetries in the $r = 4m+4$ radices are also similar to those in the $r = 4m + 2$ case. At the same time, it should be noted that there are subtle differences between these two cases within the equations when calculating $Y_{r/2-k}$ and $Y_{r/2+k}$.

Regarding the horizontal symmetry, because $W_r^{(r/2-i)k} = (-1)^k W_r^{-ik}$ and $W_r^{(r/2+i)k} = (-1)^k W_r^{ik}$, we can unite like terms $x_i W_r^{ik}$, $x_{r/2-i} W_r^{(r/2-i)k}$, $x_{r/2+i} W_r^{(r/2+i)k}$ and $x_{r-i} W_r^{(r-i)k}$ for Y_k, as presented in Eq.11 where $k \in [1, m]$.

$$
\begin{aligned}
Y_k =& x_0 + (-1)^k x_{r/2} + x_{r/4} W_r^{r/4 \cdot k} + x_{3r/4} W_r^{3r/4 \cdot k} \\
&+ \sum_{i=1}^{m}(c_i + (-1)^k u_i)a_{ik} + \sum_{i=1}^{m}(d_i - (-1)^k v_i)b_{ik} \cdot j
\end{aligned} \tag{11}
$$

Regarding the vertical symmetry, we use Eq.12 to calculate Y_{r-k}, $Y_{r/2-k}$, and $Y_{r/2+k}$. Because the four outputs Y_k, Y_{r-k}, $Y_{r/2-k}$, and $Y_{r/2+k}$ are like terms, we optimize the calculations by uniting the like terms and calculating these four results together.

$$
\begin{aligned}
Y_{r-k} =& x_0 + (-1)^k x_{r/2} \\
&+ \sum_{i=1}^{m}(c_i + (-1)^k u_i)a_{ik} - \sum_{i=1}^{m}(d_i - (-1)^k v_i)b_{ik} \cdot j \\
&+ x_{r/4} W_r^{r/4 \cdot (r-k)} + x_{3r/4} W_r^{3r/4 \cdot (r-k)} \\
Y_{r/2-k} =& x_0 + (-1)^{k+1} x_{r/2} + \sum_{i=1}^{m}(-1)^i(c_i + (-1)^k u_i)a_{ik} \\
&- \sum_{i=1}^{m}(-1)^i(d_i - (-1)^k v_i)b_{ik} \cdot j \\
&+ x_{r/4} W_r^{r/4 \cdot (r/2-k)} + x_{3r/4} W_r^{3r/4 \cdot (r/2-k)} \\
Y_{r/2+k} =& x_0 + (-1)^{k+1} x_{r/2} + \sum_{i=1}^{m}(-1)^i(c_i + (-1)^k u_i)a_{ik} \\
&+ \sum_{i=1}^{m}(-1)^i(d_i - (-1)^k v_i)b_{ik} \cdot j \\
&+ x_{r/4} W_r^{r/4 \cdot (r/2+k)} + x_{3r/4} W_r^{3r/4 \cdot (r/2+k)}
\end{aligned} \tag{12}
$$

4.2 The Periodic Property

The number of floating-point operations can be further reduced by capitalizing on the twiddle periodicity. Let $p = gcd(r, k)$ and $q = r/p$. When $p = 1$, Y_k's twiddles contain only one period. When $p \neq 1$, Y_k's twiddles periodically repeat at intervals of length q. We refer to this as the periodic property. Consequently, inputs with a distance of q multiply the same twiddle; therefore, we can obtain Eq.13.

$$
\begin{aligned}
Y_k &= \sum_{i=0}^{r-1} x_i W_r^{ik} \\
&= \sum_{i=0}^{r-1} x_i W_r^{(ik \bmod r)} \\
&= \sum_{s=0}^{p-1} \sum_{t=0}^{q-1} x_{t+s \cdot q} W_r^{tk} \\
&= \sum_{t=0}^{q-1} (\sum_{s=0}^{p-1} x_{t+s \cdot q}) \cdot W_r^{tk}
\end{aligned} \tag{13}
$$

In this way, we divide Y_k's r twiddles into p groups, and each group contains the same q twiddles. Benefiting from this periodic property, like terms occur when calculating Y_k; thus, we can reduce the number of floating-point operations by uniting these like terms, as presented in Eq.13.

We define a new complex sequence $\hat{x}$ to represent the inner summation of Eq.13. Because each Y_k contains its own $\hat{x}$, when k is given, p and q are settled. Hence, each element of $\hat{x}$ is defined in Eq.14 with $t \in [0, q-1]$; thus, the Y_k in Eq.13 can be re-expressed as the Y_k in Eq.14.

$$
\begin{aligned}
\hat{x}_{(t,k)} &= \sum_{s=0}^{p-1} x_{t+s \cdot q} \\
Y_k &= \sum_{t=0}^{q-1} \hat{x}_{(t,k)} \cdot W_r^{tk}
\end{aligned} \tag{14}
$$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2020.2977629, IEEE Transactions on Parallel and Distributed Systems

Next, we extract the periodic properties of the three cases ($r = 2m + 1$, $r = 4m + 2$, and $r = 4m + 4$) to further reduce the number of floating-point operations. At a higher level of abstraction, the periodic properties can be seen as the outer framework of the symmetric properties.

4.2.1 The $r = 2m + 1$ Radices

In our previous work [20], because we have not considered the periodicity of the $r = 2m + 1$ case, it only support the generation of prime radices, such as 3 and 5. Here, we further generalize the $r = 2m + 1$ radices by formulating and combining the horizontal and vertical symmetries with the periodicity, so that we can generate radices of all odd numbers, such as radix 9. We split and parameterize the calculation of Eq.5 and Eq.6 into parts that are denoted as A, C, and D, as shown in Eq.15. Based on these parameters, the Y_k in Eq.5 and the Y_{r-k} in Eq.6 are re-expressed in Eq.16. Since this calculation pattern can calculate two results at a time, we call it **the pair pattern**.

$$\begin{aligned} A &= \hat{x}_{(0,k)} \\ C &= (\hat{x}_{(i,k)} + \hat{x}_{(q-i,k)})a_{ik} \\ D &= (\hat{x}_{(i,k)} - \hat{x}_{(q-i,k)})b_{ik} \cdot j \end{aligned} \tag{15}$$

$$\begin{aligned} Y_k &= A + \sum_{i=1}^{m} C + \sum_{i=1}^{m} D \\ Y_{r-k} &= A + \sum_{i=1}^{m} C - \sum_{i=1}^{m} D \end{aligned} \tag{16}$$

4.2.2 The $r = 4m + 2$ and $r = 4m + 4$ Radices

Considering that there are few differences in symmetry and periodicity between the $r = 4m + 2$ case and the $r = 4m + 4$ case, we use a uniform way to represent them. We define ($o = r/2 \bmod 2$) to distinguish the two cases. When $r = 4m + 2$, $o = 1$; otherwise, $o = 0$. Unlike our previous work [20], which only supports the $r = 4m+4$ case for power-of-two radices, this paper can generate radices of all even number including power-of-two radices.

To combine the horizontal and vertical symmetries with the periodicity for the $r = 4m+2$ and $r = 4m+4$ radices, we parameterize their calculation patterns of Y_k, Y_{r-k}, $Y_{r/2-k}$, and $Y_{r/2+k}$ using A, B, C, and D, as shown in Eq.17. Based on these parameters, these outputs can be re-expressed as Eq.18. Since this calculation pattern can calculate four results at a time, we call it **the quad pattern**. Note that when $r = 4m + 4$, $Y_{r/4}$ and $Y_{3r/4}$ are calculated separately here based on A and B. When $k = r/4$, $o = 0$ and $p = q = 4$; therefore, we obtain $Y_{r/4} = A + B$, and $Y_{3r/4} = A - B$.

$$\begin{aligned} A =& \hat{x}_{(0,k)} - \hat{x}_{(q/2,k)} \\ B =& (1-o)((\hat{x}_{(q/4,k)} - \hat{x}_{(3q/4,k)}) \cdot Im(W_{4p}^{k}) \cdot j) \\ C =& ((\hat{x}_{(i,k)} + \hat{x}_{(q-i,k)}) \\ & + (-1)^{k+o}(\hat{x}_{(q/2-i,k)} + \hat{x}_{(q/2+i,k)}))a_{ik} \\ D =& ((\hat{x}_{(i,k)} - \hat{x}_{(q-i,k)}) \\ & - (-1)^{k+o}(\hat{x}_{(q/2-i,k)} - \hat{x}_{(q/2+i,k)}))b_{ik} \cdot j \end{aligned} \tag{17}$$

CPX_ADD(out,in1,in2): out.r=in1.r+in2.r out.i=in1.i+in2.i	**CPX_SUB**(out,in1,in2): out.r=in1.r-in2.r out.i=in1.i-in2.i
CPX_MLA(out,in1,in2,s): out.r=in1.r+s*in2.r out.i=in1.i+s*in2.i	**CPX_MUL_S**(out,in,s): out.r=s*in.r out.i=s*in.i
CPX_MUL(out,in1,in2): rr=in1.r*in2.r ii=in1.i*in2.i ri=in1.r*in2.i ir=in1.i*in2.r out.r=rr-ii out.i=ri+ir	**CPX_OUT**(head,tail,A,B): head.r=A.r-B.i head.i=A.i+B.r tail.r=A.r+B.i tail.i=A.i-B.r

Fig. 3: Meta templates supported in AutoFFT.

$$\begin{aligned} Y_k =& A + B + \sum_{i=1}^{q/4-1} C + \sum_{i=1}^{q/4-1} D \\ Y_{r-k} =& A - B + \sum_{i=1}^{q/4-1} C - \sum_{i=1}^{q/4-1} D \\ Y_{r/2-k} =& A - B + \sum_{i=1}^{q/4-1} (-1)^i \cdot C - \sum_{i=1}^{q/4-1} (-1)^i \cdot D \\ Y_{r/2+k} =& A + B + \sum_{i=1}^{q/4-1} (-1)^i \cdot C + \sum_{i=1}^{q/4-1} (-1)^i \cdot D \end{aligned} \tag{18}$$

5 FFT Kernel Generation

5.1 The Computational Template Designer

The template-based code generation system is built on the pair and quad patterns. To make full use of these patterns and enable the code generation, we introduce a computational template designer. The template designer takes these two patterns as inputs and then obtains high-level computational templates as outputs. The computational templates are the basis of the code generation system. They consist of meta templates and hybrid templates. Meta templates are predefined hardcoded C preprocessor macros representing some basic arithmetic operations on complex numbers. Hybrid templates are defined as high-level C functions extracted by the designer according to the pair and quad patterns (Eq.16 and Eq.18). This is because the value of radix r determines the number and type of meta templates used in hybrid templates. Note that the input sequence x is equal to the original inputs multiplied by twiddles, and Y is the transformed result.

5.1.1 Meta Templates

As presented in section 4, butterfly calculations consist of some arithmetic operations on complex numbers; therefore, we define these operation units as meta templates in Fig.3.

- ***CPX_ADD***(), ***CPX_SUB***(), *and* ***CPX_MUL***() represent complex addition, subtraction, and multiplication, respectively.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2020.2977629, IEEE Transactions on Parallel and Distributed Systems


```
CALC_LIKE_TERMS(add[],sub[],in[],r):
  m=(r-1)/2, t=r-1;
  if(r%2==0)
    CPX_ADD(add[0],in[0],in[r/2]);
    CPX_SUB(sub[0],in[0],in[r/2]);
  else
    add[0]=in[0];sub[0]=-in[0];
  endif
  for(h=1; i<=m; h++,t--)
    CPX_ADD(add[h],in[h],in[t]);
    CPX_SUB(sub[h],in[h],in[t]);
  end for
```

```
CALC_OUT_SPECIAL(out[],add[],sub[],r):
  m=r/2,out[0]=out[m]=0
  for(i=0;i<=m;i++)
    CPX_ADD(out[0],out[0],add[i]);
  end for
  if(r%2==0)
    for(i=0;i<m;i+=2)
      CPX_ADD(out[m],out[m],add[i]);
      CPX_SUB(out[m],out[m],add[i+1]);
    end for
  end if
```

```
CALC_OUT_PAIR(out[],in[],k,r):
  A=C=D=0; p=gcd(r,k); q=r/p;
  _in = period_in(in, p, q);
  A=pair_A(_in,0,k);
  CPX_ADD(C,C,A);
  for(i=1; i<q/2; ++i)
    _C=pair_C(_in,i,q-i,k)
    _D=pair_D(_in,i,q-i,k)
    CPX_ADD(C,C,_C);
    CPX_ADD(D,D,_D);
  end for
  CPX_OUT(out[k],out[r-k],C,D);
```

```
CALC_OUT_QUAD(out[],in[],k,r,o):
  A = 0;
  B = 0;
  C1 = C2 = 0;
  D1 = D2 = 0;
  p=gcd(r,k); q=r/p;
  _in = period_in(in, p, q);

  for(i=1; i<q/4; ++i)
    C=quad_C(_in,i,q-i,q/2-i,q/2+i,k,o)
    D=quad_D(_in,i,q-i,q/2-i,q/2+i,k,o)
    CPX_ADD(C1,C1,C);
    CPX_ADD(D1,D1,D);
    if(i%2==1)
      CPX_SUB(C2,C2,C);
      CPX_SUB(D2,D2,D);
    else
      CPX_ADD(C2,C2,C);
      CPX_ADD(D2,D2,D);
    end if
  end for

  A=quad_A(_in,0,q/2,k);
  B=quad_B(_in,q/4,3q/4,k,p,o);

  if((k==1) && (o==0))
    out[r/4]=A+B;
    out[3*r/4]=A-B;
  end if

  out[k]=quad_Yk(A,B,C1,D1);
  out[r-k]=quad_Yrsk(A,B,C1,D1);
  out[r/2-k]=quad_Yr2sk(A,B,C2,D2);
  out[r/2+k]=quad_Yr2pk(A,B,C2,D2);
```

Fig. 4: Hybrid templates supported in AutoFFT.

- ***CPX_MLA***$(out, in1, in2, s)$ represents the fused multiply-add (FMA) operation on complex numbers. It is used for the accumulation operations in Eq.16 and Eq.18. Likewise, ***CPX_MUL_S***(out, in, s) is used for these two equations to complete the multiplication between a complex number in and a real number s.
- ***CPX_OUT***$(head, tail, A, B)$ is used to calculate a pair of results (such as Y_k and Y_{r-k}, and $Y_{r/4-k}$ and $Y_{3r/4-k}$), as presented in Eq.16 and Eq.18.

5.1.2 *Hybrid Templates*

Hybrid templates are extracted and formalized by the computational template designer to implement the pair pattern (Eq.16) for $r = 2m + 1$ radices, and the quad pattern (Eq.18) for $r = 4m + 2$ and $r = 4m + 4$ radices, as shown in Fig.4.

- ***CALC_LIKE_TERMS***(add, sub, in, r) unites like terms. When $r = 2m + 1$, according to Eq.15, add stores (x_0), $(x_1 + x_{r-1})$, ..., $(x_m + x_{r-m})$, and sub stores $(-x_0)$, $(x_1 - x_{r-1})$, ..., $(x_m - x_{r-m})$. When r is an even number, according to Eq.17, add stores $(x_0 + x_m)$, $(x_1 + x_{r-1})$, ..., $(x_m + x_{r-m})$, and sub stores $(x_0 - x_m)$, $(x_1 - x_{r-1})$, ..., $(x_m - x_{r-m})$; in contains $x_0 + \ldots + x_{r-1}$.
- ***CALC_OUT_SPECIAL***(out, add, sub, r) adopts Eq.4 to separately calculate Y_0 when r is an odd number; when r is an even number, it separately calculates Y_0 and $Y_{r/2}$ using Eq.7.
- ***CALC_OUT_PAIR***(out, add, sub, k, r) implements the pair pattern (Y_k and Y_{r-k}) according to Eq.16. Because this process only involves basic complex number operations, we simplify the expressions for calculating Eq.15's $\hat{x}/A/C/D$ for readability.
- ***CALC_OUT_QUAD***(out, add, sub, k, r) implements the quad pattern (Y_k, $Y_{r/2-k}$, $Y_{r/2+k}$, and Y_{r-k}) according to Eq.18. In addition, $Y_{r/4}$ and $Y_{3r/4}$ are calculated separately. We simplify the expressions for calculating Eq.17's $\hat{x}/A/B/C/D$ for readability.

Algorithm 1 butterfly_kernel(out, in, tw, r, i, isFirst)

Input: in[]: the input data of current stage; tw[]: twiddles; r: radix; i: the i^{th} butterfly; isFirst: whether it is the first stage.
Output: out[]: output data.

```
1: Load inputs into tmpIn[] from in[] according to i
2: if isFirst=0 then
3:    Load twiddles into tmpTW[] from tw[]
4:    for i←1 to r do
5:       CPX_MUL(tmpIn[i], tmpIn[i], tmpTW[i-1])
6:    end for
7: end if
8: o←(r/2) mod 2
9: CALC_LIKE_TERMS(add, sub, tmpIn, r)
10: CALC_OUT_SPECIAL(tmpOut, add, sub, r)
11: if r is an odd number then
12:    for i←1 to r/2 do
13:       CALC_OUT_PAIR(tmpOut, add, sub, i, r)
14:    end for
15: else if r is an even number then
16:    for i←1 to r/4 do
17:       CALC_OUT_QUAD(tmpOut, tmpIn, i, r, o)
18:    end for
19: end if
20: Store outputs from tmpOut[] to out[]
```

Algorithm 2 FFT_kernel(out, in, tw, r, butterfly_num, isFirst)

Input: in[]: inputs; tw[]: twiddles; r: radix; butterfly_num: the number of butterflies; isFirst: whether it is the first stage.
Output: out[]: outputs.

```
1: for i←0 to butterfly_num do
2:    butterfly_kernel(out, in, tw, r, i, isFirst)
3: end for
```

5.2 The C FFT Kernel Generator

After we obtain the computational templates, a C FFT kernel generator is designed to generate integral FFT kernels that can independently solve given DFT problems. According to Alg.1, this generator takes a given radix r (an arbitrary natural number) as input and calls the needed computational templates, then an efficient C FFT kernel for the given radix r can be automatically generated.

The butterfly kernel is the core computing module for calculating one butterfly, so its implementation and optimization are critical to the overall performance. In Alg.1, lines 1∼7 load and multiply the inputs and twiddles and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2020.2977629, IEEE Transactions on Parallel and Distributed Systems

then store the results in the temporary complex array $tmpIn$. In the first stage, all twiddles equal 1; thus, lines 3∼6 are skipped. Line 9 calls *CALC_LIKE_TERMS*() to calculate the like terms that can be reused by other hybrid templates. When r is an odd number, line 10 calls *CALC_OUT_SPECIAL*() to calculate Y_0, and lines 12∼14 then perform the pair pattern. When r is an even number, line 10 calculates Y_0 and $Y_{r/2}$, and lines 16∼18 perform the quad pattern. Finally, line 20 stores the results in out.

The FFT kernel is adopted to process multiple butterflies in a for-loop, as shown in Alg.2. According to the Stockham FFT network, adjacent butterflies can be calculated together, so we can SIMDize the for-loop to improve the performance of the FFT kernel.

5.3 The Assembly Template Optimizer

The C FFT kernel generator is mostly concerned with reducing floating-point operations. However, the underlying hardware designs for factors such as registers and pipeline structure also have a high impact on the performance. To further exploit the hardware resources, we need to accelerate and transform the C FFT kernels into assembly kernels. To automate the transformation process for varying architectures, the assembly template optimizer is introduced. The optimizer takes optimization templates as inputs, and then automatically transform the C FFT kernels into assembly according to the register allocation strategy, which is defined based on the underlying architectures. The optimization templates are hardcoded architecture-specific assembly templates that are transformed from the high-level meta templates.

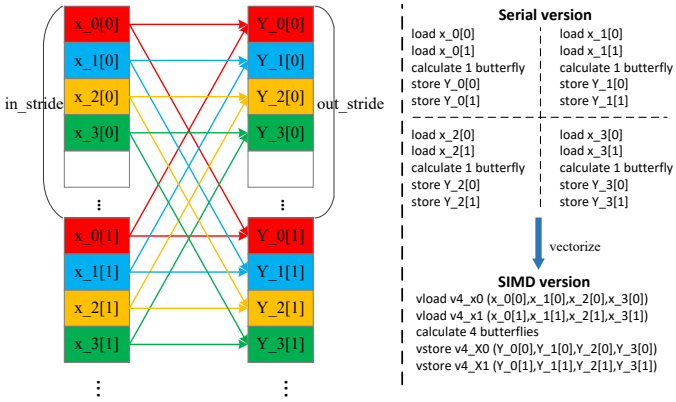


Fig. 5: SIMDize four adjacent radix-2 butterflies. Note that in the i^{th} stage, $in\_stride = N/2$ and $out\_stride = 2^i$.

5.3.1 *Butterfly Vectorization*

Most modern CPUs provide SIMD techniques to boost performance; therefore, AutoFFT adopts the SIMD-friendly Stockham FFT network. In this network, the inputs and outputs of adjacent butterflies are contiguous in memory; therefore, Alg.2 can be SIMDized to calculate multiple butterflies simultaneously. Fig.5 presents how to SIMDize multiple radix-2 butterflies for a sequence of length N. Four adjacent radix-2 butterflies are represented by four colors. One radix-2 butterfly requires two inputs (such as x_0[0] and x_0[1]) and yields two outputs (Y_0[0] and Y_0[1]).

Meta Templates	ARMv8 Optimization Templates	x86-64 Optimization Templates
CPX_ADD(out,in1,in2): out.r=in1.r+in2.r out.i=in1.i+in2.i	**CPX_ADD**(out_r,out_i,in1_r,in1_i,in2_r,in2_i): fadd out_r.4s, in1_r.4s, in2_r.4s fadd out_i.4s, in1_i.4s, in2_i.4s	**CPX_ADD**(out,in1,in2): vaddps %in1, %in2, %out
CPX_SUB(out,in1,in2): out.r=in1.r-in2.r out.i=in1.i-in2.i	CPX_SUB(out_r,out_i,in1_r,in1_i,in2_r,in2_i): fsub out_r.4s, in1_r.4s, in2_r.4s fsub out_i.4s, in1_i.4s, in2_i.4s	**CPX_SUB**(out,in1,in2): vsubps %in1, %in2, %out
CPX_MLA(out,in1,in2,s): out.r=in1.r+s*in2.r out.i=in1.i+s*in2.i	**CPX_MLA**(out_r,out_i,in1_r,in1_i,in2_r,in2_i,s): fmla out_r.4s, in2_r.4s, s.s[0] fmla out_i.4s, in2_i.4s, s.s[0]	**CPX_MLA**(out,in1,in2,s): vfmadd231ps %in2, %s, %out
CPX_MUL_S(out,in,s): out.r=s*in.r out.i=s*in.i	**CPX_MUL_S**(out_r,out_i,in_r,in_i,s): fmul out_r.4s, in_r.4s, s.s[0] fmul out_i.4s, in_i.4s, s.s[0]	**CPX_MUL_S**(out,in,s): vmulps %in, %s, %out
CPX_MUL(out,in1,in2): rr=in1.r*in2.r ii=in1.i*in2.i ri=in1.r*in2.i ir=in1.i*in2.r out.r=rr-ii out.i=ri+ir	**CPX_MUL**(out_r,out_i,in1_r,in1_i,in2_r,in2_i): fmul rr.4s, in1_r.4s, in2_r.4s fmul ii.4s, in1_i.4s, in2_i.4s fmul ri.4s, in1_r.4s, in2_i.4s fmul ir.4s, in1_i.4s, in2_r.4s fsub out_r.4s, rr.4s, ii.4s fadd out_i.4s, ri.4s, ir.4s	**CPX_MUL**(out,in1,in2): vmovsldup %in2, %tmp0 vmovshdup %in2, %tmp1 vmulps %in1, %tmp0, %tmp0 vpshufd $ 0xb1, %tmp1, %tmp1 vfmaddsub213ps %tmp1, %tmp0, %out
CPX_OUT(head,tail,A,B): head.r=A.r-B.i head.i=A.i+B.r tail.r=A.r+B.i tail.i=A.i-B.r	**CPX_OUT**(h_r,h_i,t_r,t_i,A_r,A_i,B_r,B_i): fsub h_r.4s, A_r.4s, B_i.4s fadd h_i.4s, A_i.4s, B_r.4s fadd t_r.4s, A_r.4s, B_i.4s fsub t_i.4s, A_i.4s, B_r.4s	**CPX_OUT**(head,tail,A,B): vshufps 0xb1, %B, %B, %tail vaddsubps %tail, %A, %head vfmsubadd231ps ONE(%rip), %A, %tail

Fig. 6: Instruction mapping rules between meta templates and optimization templates.

5.3.2 *Instruction Mapping*

We define instruction mapping rules to translate the high-level computational templates into architecture-specific hardcoded optimization templates by selecting and scheduling efficient assembly instructions. AutoFFT currently focuses on the ARMv8 ISA and the x86-64 ISA. Their instruction mapping rules are listed in Fig.6. When SIMDizing butterflies on the ARMv8 architecture, we use two 128-bit registers to separately hold four complex numbers' real and imaginary parts. On the Intel Haswell and AMD Zen architectures, we use one 256-bit register to hold four complex numbers by interleaving their real and imaginary parts. We adopt this approach for the following reasons: 1) AVX2 does not support efficient load/store instructions such as ld2/st2 instructions in ARM NEON; and 2) AVX2 provides the $vaddsubps$ instruction to efficiently perform multiplication on complex numbers for the interleaved pattern.

To keep every execution unit of the processor busy with instructions, AutoFFT reorders the instruction streams using the following two methods. 1) Detach instructions with dependencies, especially for memory instructions and corresponding arithmetic instructions. Because the latencies of memory instructions are high, we insert independent instructions between the memory instructions and the corresponding dependent arithmetic instructions. 2) Rearrange independent instructions based on the functionalities of the issue ports. For example, the issue ports 0/1/5 of the Haswell architecture perform arithmetic operations, and independent instructions such as floating-point multiplication, FMA, and shuffle can be dispatched to these three ports in parallel [44].

Benefiting from the optimization templates, when new architectures emerge, we only need to implement the corresponding optimization templates. As shown in Fig.6, the optimization templates use register aliases instead of physi-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2020.2977629, IEEE Transactions on Parallel and Distributed Systems

cal vector registers. In the following, we define the register allocation strategy for different radices to transform the C FFT kernels into corresponding assembly FFT kernels.

5.3.3 Register Allocation Strategy

Vector registers are scarce resources in modern CPUs. We design a strict vector register allocation strategy to make full use of register resources and enable the automatic generation of assembly FFT kernels. The register allocation strategy contains two steps: the register alias usage step, which is used to reduce the difficulty of the generation of assembly FFT kernels, and the vector register mapping step, which is used to complete the mappings between register aliases and vector registers.

Register Alias Usage. The number of vector registers in modern CPUs is limited. ARMv8 processors contain 32 128-bit vector registers, while Intel Haswell and AMD Zen processors contain 16 256-bit vector registers. Because AMD Zen and Intel Haswell have the same amount of registers, and their register alias usages are the same, so we omit the repeated description of AMD Zen's register alias usage for simplicity. During the process of assembly FFT kernel generation, vector registers become increasingly scarce as the radix increases, so we introduce register aliases to reduce the difficulty of assembly code generation. By using register aliases, we do not need to worry about whether the number of physical vector registers meets the needs of assembly code generation. In this step, only register aliases are used in the generated assembly FFT kernels.

The main idea of register alias usage is to group register aliases according to their functionalities in Alg.1 and to strictly define the usage rules and the required amount of register aliases of each group. According to Alg.1, AutoFFT divides register aliases into four groups: the input group, the twiddle group, the temporary group, and the output group. We now analyze the required register aliases of the four groups for the radix-r kernel.

On the ARMv8 architecture, the input group requires $2r$ registers for $tmpIn$, the twiddle group requires $2r-2$ registers for $tmpTW$, and the temporary group requires $2r$ registers for add/sub. The other required registers in the temporary group and the output group are as follows: 1) When r is an odd number, the temporary variables (rr/ii/ri/ir used in $CPX\_MUL()$ and C/D used in $CALC\_OUT\_PAIR()$) occupy 8 registers. In addition, 4 registers are needed to hold a pair of outputs. Thus, the sum of the required registers is $6r+10$. 2) When r is an even number, $CALC\_OUT\_QUAD()$'s $A/B/C1/C2/D1/D2$ occupy 12 temporary registers, and 8 registers for four outputs are required. Thus, the sum of the required registers is $6r+18$. On the Haswell architecture, because we use one 256-bit register to process 4 complex numbers, when r is an odd number, the sum of the required registers is $3r+5$; when r is an even number, the sum of the required registers is $3r+9$.

Vector Register Mapping. So far, the assembly FFT kernels generated from the register alias usage step only use register aliases, however, to complete the code generation and obtain runnable assembly FFT kernels, we still need to replace register aliases with physical vector registers. Hence, the main idea of the vector register mapping step is to construct the mappings between register aliases and physical vector registers. Because AMD Zen and Intel Haswell have the same register resources, so we omit the repeated description of AMD Zen's register mapping rule for simplicity. The register mapping rule varies for different radices and can be divided into the following three cases based on Alg.1:

(1) Small Radices. According to the analysis in the register alias usage step, 1) when r is an odd number, on the ARMv8 architecture, where the required vector registers are $6r+10 \leq 32$, we have $r \leq 11/3$. On the Haswell architecture, where $3r+5 \leq 16$, we have $r \leq 11/3$; therefore, radix 3 is a small radix. 2) When r is an even number, on the ARMv8 architecture, where $6r+18 \leq 32$, we have $r \leq 7/3$; on the Haswell architecture, where $3r+9 \leq 16$, we have $r \leq 7/3$. Therefore, radix 2 is a small radix. Hence, there are sufficient registers for each group to independently perform their tasks for radices 2 and 3.

(2) Medium Radices. As the radix grows, the four groups require more vector registers, and the register resources become insufficient to independently perform the tasks for the four groups. Thus, we reuse registers according to the following four rules: 1) Reuse $tmpTW$. the operations between twiddles and inputs are independent; therefore, it is unnecessary to load all twiddles at once, and we suggest loading 4 twiddles each time. After the complex number multiplications are completed, the registers can be reused for the next 4 twiddles. In addition, after completing lines 1∼7 of Alg.1, the registers used for $tmpTW$ can be freed. 2) Reuse $tmpIn$. For $CALC\_LIKE\_TERMS()$ in line 9, we introduce one temporary complex number to stagger the registers used for add/sub and $tmpIn$; then, add/sub can reuse $tmpIN$'s registers. 3) Reuse the temporary registers. Temporary registers can also be used for the rr/ii/ri/ir of $CPX\_MUL()$. 4) Reuse $tmpOut$. In lines 13 and 17, after obtaining an output, we immediately store it in memory, which means the output group only needs to maintain one complex number.

As analyzed above, on the ARMv8 architecture, lines 1∼7 in Alg.1 require $2r+8$ registers for $tmpIn$ and $tmpTW$. Subsequently, we have the following: 1) When r is an odd number, lines 9∼20 require only $2r+6$ registers: the temporary group requires $2r$ registers for add/sub, 4 registers are used for the temporary variables, and $tmpOut$ requires 2 registers. By adopting the four reuse rules, $2r+8$ registers are sufficient because they can be reused in lines 9∼20; if $2r+8 \leq 32$, we have $r \leq 12$. Similarly, the Haswell architecture requires $r+4$ registers; if $r+4 \leq 16$, we have $r \leq 12$; therefore, 5, 7, 9, and 11 are medium radices. 2) When r is an even number, on the ARMv8 architecture, lines 9∼20 require $2r+12$ registers, which is larger than $2r+8$: add/sub require $2r$ registers, the temporary variables require 12 registers, and $tmpOut$ requires 2 registers. When $2r+14 \leq 32$, we have $r \leq 9$. Similarly, the Haswell architecture requires $r+7$ registers. When $r+7 \leq 16$, we have $r \leq 9$; therefore, 4, 6, and 8 are medium radices.

(3) Large Radices. When $r > 11$, vector register resources are insufficient: we have to use stack or memory instructions to temporarily hold relevant data, which degrades the performance. Large radices require more vector registers, which are limited in modern CPUs. AutoFFT

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2020.2977629, IEEE Transactions on Parallel and Distributed Systems

provides C FFT kernels to perform the FFT computation for large radices. Specialized FFT algorithms, such as Rader's algorithm [2], can achieve better performance than the Cooley-Tukey algorithm for large radices.

To replace register aliases with specific physical vector registers for assembly kernels of small and medium radices, the template optimizer maintains a register resource pool and a lookup register usage table based on the register mapping rule to guarantee the consistency of vector register usage across the instruction streams. As analyzed above, because there are sufficient vector registers for small radices, the optimizer can directly take out the required registers in the resource pool and map specific vector registers to corresponding register aliases in the register usage table. For medium radices, because we have strictly defined the register mapping rule for them above, the optimizer determines the exact reuse time of different registers based on the four register reuse rules and fills out the register usage table. When a vector register is ready for reuse, it will be labeled the "usable" tag and returned to the register resource pool.

The register allocation strategy essentially is determined by the number and the width of SIMD registers, so it can be easily extended to the future ARM SVE and AVX-512 by replacing the number and the width of corresponding SIMD registers. Because the FFT kernels of radices 2/3/4/5/6/7/8/9/11 (small and medium radices) are well-optimized assembly code, AutoFFT is best at computing FFTs of sizes of the form $2^a3^b5^c7^d11^e$ (the exponents are arbitrary). FFTs of other sizes are slower because they need to call the C FFT kernels of large radices. Regardless, sizes of the form $2^a3^b5^c7^d11^e$ meet the needs of most applications.

6 Butterfly Network Implementations and Optimizations

The butterfly network determines data access patterns and the order in which the needed FFT kernels are executed for a given DFT problem. Even for the same butterfly network, different implementations and optimizations can lead to different performances. Fig.7 presents the differences between the traditional decimation-in-time (DIT) network and the Stockham network when processing the FFT of size eight. This section describes the specific implementations and optimizations adopted in AutoFFT's network.

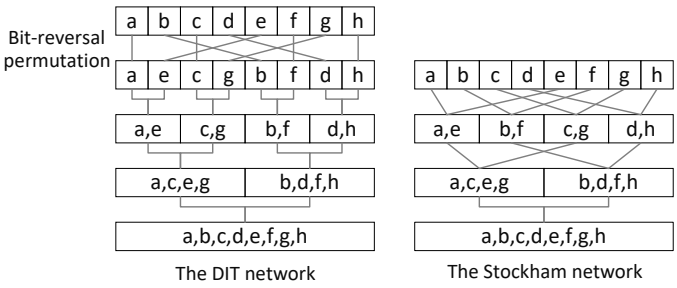


Fig. 7: Dataflows of the DIT network and the Stockham network when processing the FFT of size eight.

6.1 Specific Implementations of the Stockham Network

The Stockham network is organized as a three-level network in AutoFFT. These three levels are embodied as three for-loops in the concrete implementation, as shown in Alg.2 and Alg.3. Here, we define the outermost for-loop (Alg.3's line 3) as the stage loop, the middle for-loop (Alg.3's line 7) as the section loop, and the innermost loop as the butterfly loop (Alg.2's line 1). In this stage-section-butterfly network, the number of butterflies in each stage is always equal to the given FFT size N over the adopted radix r (N/r). In addition, one stage contains at least one section, which contains at least one butterfly. Fig.8 shows the stage-section-butterfly network of an eight-point FFT based on radix 2. In this example, the first stage contains four sections, and each section contains one radix-2 butterfly; the second stage contains two sections, and each section contains two radix-2 butterflies; and the third stage contains one section, which contains four radix-2 butterflies.

Alg.3 presents the pseudocode of the three-level network. Compared with traditional Cooley-Tukey networks, such as the DIT network and decimation-in-frequency (DIF) [45] network, the three-level network contains three advantages: 1) Requiring no explicit bit-reversal permutation. In traditional butterfly networks, such as the DIT network shown in Fig.7, the explicit bit-reversal permutation is required to reorder data from the natural order to the bit-reversed order. The bit-reversal permutation introduces extra and incoherent memory accesses, which can be expensive. Moreover, incoherent memory accesses can cause difficulties in unifying memory access patterns of inputs and outputs. Because both inputs and outputs of the three-level network are in the natural order, no explicit bit-reversal permutation is required. 2) SIMD-friendly. Considering that memory layouts of inputs and outputs of adjacent butterflies in each section are contiguous, we can easily SIMDize multiple adjacent butterflies in each section. 3) Supporting mixed radices. Since inputs and outputs are in the natural order, our network supports the lengths of arbitrary combinations of varying radices.

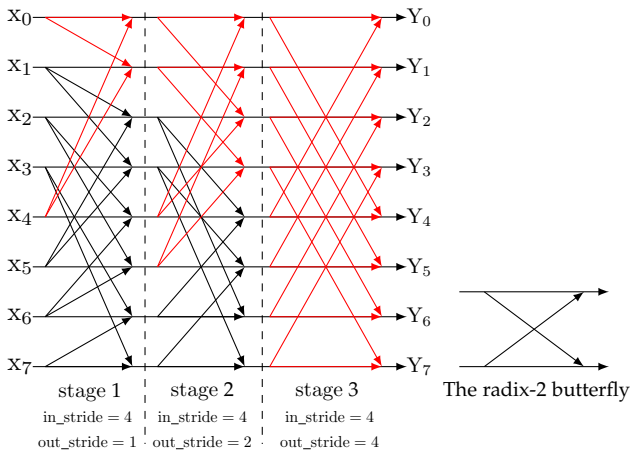


Fig. 8: The stage-section-butterfly network diagram for the FFT of size eight. Red arrows represent one section in each stage.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2020.2977629, IEEE Transactions on Parallel and Distributed Systems

Algorithm 3 The overview of the butterfly network

Input: in[]: inputs; radix[]: required radices in each stage; tw[]: twiddles; stage_num: stages in the network; section_num: sections in each stage.
Output: out[]: outputs.

```
1: r←radix[0], butterfly_num=1
2: FFT_kernel(out, in, NULL, r, section_num, 1)
3: for i←1 to stage_num do
4:    butterfly_num=butterfly_num*r
5:    r←radix[i] // radix in stage i
6:    section_num=section_num/r
7:    for j←0 to section_num do
8:       FFT_kernel(out, in, tw, r, butterfly_num, 0)
9:    end for
10: end for
```

6.2 Optimizations in the First Stage

As shown in Fig.8 and Alg.3, the first stage of the network is processed and optimized independently to further exploit the parallelism: 1) Unlike the other stages, twiddles in the first stage are always equal to the constant 1 according to the definition of twiddles, so memory read operations of twiddles and complex number multiplications between inputs and twiddles can be completely eliminated. 2) Since each section of the first stage only contains one butterfly, instead of SIMDizing butterflies within one section in the section level similar to other stages, we need to directly SIMDize butterflies across multiple sections in the first stage. 3) To avoid explicit bit-reversal permutation and unify memory access patterns for other stages, we rearrange the memory layout of computational results of the first stage, so the output stride of memory store operations in the first stage is different from those in other stages. The output stride can be calculated using Eq.19, where $radix[]$ holds the radix of each stage. Regarding the input stride in_stride, it is equal to $N/radix$ for all stages.

$$out\_stride = \begin{cases} 1, & \text{the } 1^{st} \text{ stage } (i = 0) \\ \prod_{j=0}^{i-1} radix[j], & \text{the } i^{th} \text{ stage } (i > 0) \end{cases} \quad (19)$$

6.3 Cache-Friendly Butterfly Execution Order

The CPU cache system has an important impact on the performance of the FFT. Usually, the execution order of butterflies is performed stage-by-stage (breadth-first order) in accordance with the butterfly network. For example, butterflies in the second stage need to wait for all butterflies in the first stage to complete before they can be executed. In this way, when the FFT size is large, it is very likely to cause L1 cache misses: the preceding butterfly calculation results in the current stage will be replaced out of L1 cache by the results of subsequent butterflies, resulting in cache misses when butterflies in the next stage read their inputs.

To make better use of the cache system and reduce L1 cache misses, we adopt a cache-friendly manner to determine the execution order of butterflies in the butterfly network: 1) Butterflies of the first two stages are executed in the depth-first order; and 2) butterflies of the remaining stages are executed in the breadth-first order. Fig.9 presents the depth-first execution order of the first two stages. In this example, we assume the SIMD width is 2 for simplicity. To ensure that the inputs (green points) for the blue butterflies in the second stage are not replaced out of L1 cache when reading them, we will calculate the blue butterflies as soon as the calculations of the red butterflies are complete.

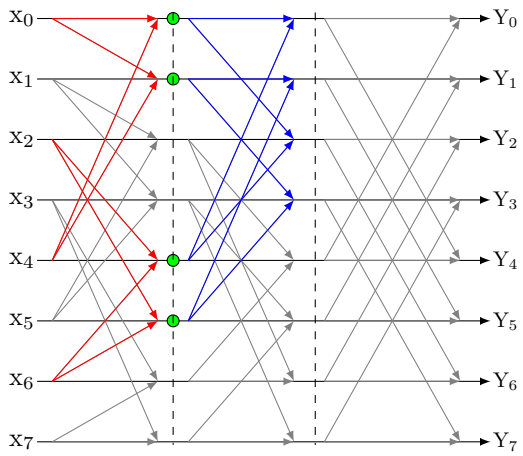


Fig. 9: The depth-first butterfly execution order in the first two stages. The order in which the butterflies of the first stage (red butterflies) is calculated depends on what inputs are required for the butterflies of the second stage (blue butterflies). For simplicity, we assume the SIMD width is 2 (SIMDize two blue butterflies) in this example.

TABLE 1: Experimental Environment

CPU	FT-2000+	Xeon E7-4850 v3	Hygon C86 7185
Arch.	AArch64	Haswell	Zen
Frequency	2.2 GHz	2.2 GHz	2.0 GHz
SIMD	128	256	256
L1 cache	32 KB	32 KB	32 KB
GCC	4.9.3	5.5.0	4.9.4
FFTW	3.3.8	3.3.8	3.3.8
ARMPL	19.2.0	-	-
Intel MKL	-	2019 Update 4	-

7 Performance Evaluation

This section evaluates the performance of AutoFFT on server-grade ARMv8 and x86-64 (Intel Haswell and AMD Zen) CPUs. AutoFFT supports complex/real and out-of-place/in-place FFT computations. Because FFTW, ARMPL, and Intel MKL are the most widely used and mature FFT libraries, we compare the performance of AutoFFT with them. For a one-dimensional (1D) FFT of length N with an execution time of t seconds, we report its performance in GFlops according to Eq.20 [46], which is adopted in the well-known benchmark benchFFT [47]. Note that the x-axis of figures in this section represents the transform size N.

$$GFlops = \frac{5N \cdot \log_2 N \cdot 10^{-9}}{t} \quad (20)$$

The experimental conditions are listed in Table 1. Our experiments take the C version of AutoFFT as the baseline to determine the performance boost achieved by AutoFFT's assembly kernels. In our experiments, we use the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2020.2977629, IEEE Transactions on Parallel and Distributed Systems

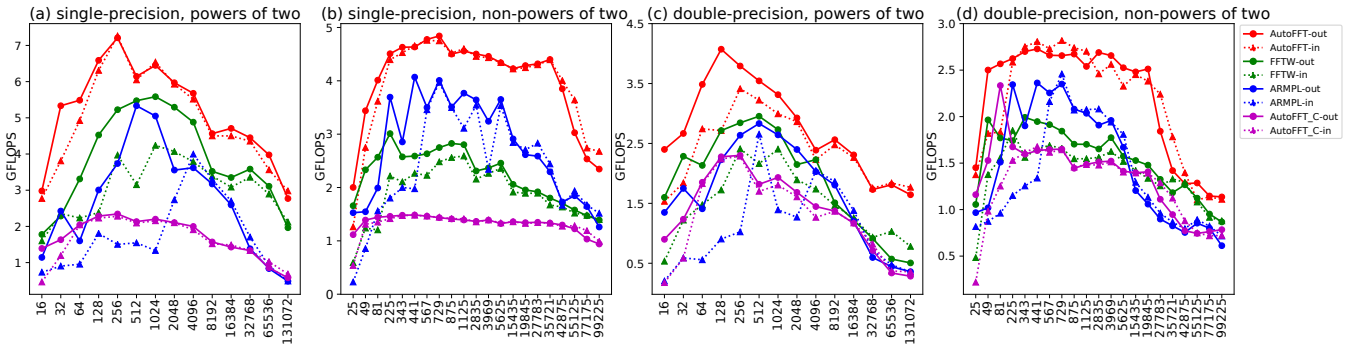


Fig. 10: The 1D C2C FFT performances on ARMv8 CPUs.

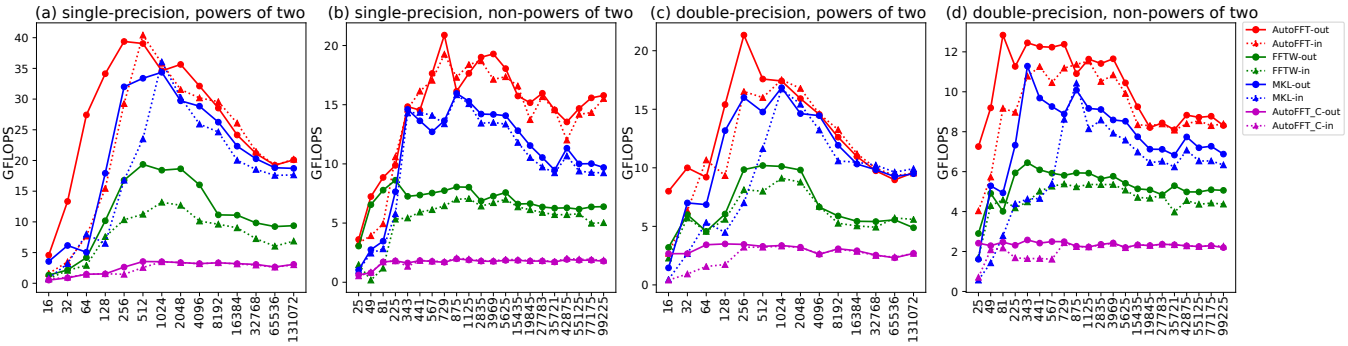


Fig. 11: The 1D C2C FFT performances on Intel Haswell CPUs.

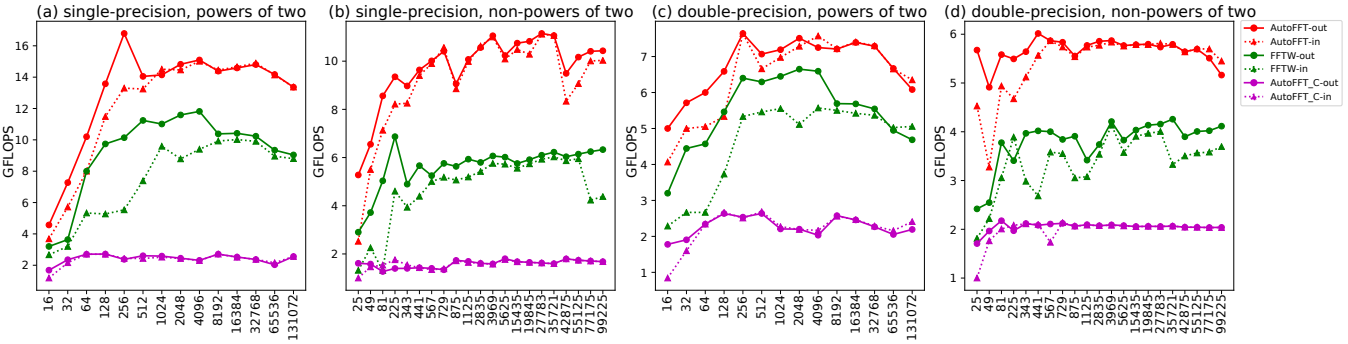


Fig. 12: The 1D C2C FFT performances on AMD Zen CPUs.

FFTW_MEASURE flag for all FFT plans. Considering Intel MKL's FFTW interfaces and its API share the same source code [48], we use the FFTW interfaces for Intel MKL in our experiments. In addition, because AMD takes FFTW as their official FFT library of AOCL [23], our experiments use the pre-built FFTW-3.3.8 library, which is compiled and provided by AOCL. For simplicity, we define a new naming scheme (a two-part string) that reflects the name of the FFT library (AutoFFT/FFTW/ARMPL/MKL) and out-of-place/in-place (out/in) FFT computations for the experimental figures. For example, AutoFFT-out denotes an out-of-place transform of AutoFFT. Because the 1D complex-to-complex (C2C) FFT is the core operation of other transforms, we conduct an in-depth analysis of the performance of the 1D C2C FFT.

Fig.10 shows the performances of the 1D C2C FFTs of AutoFFT, FFTW, ARMPL, and the baseline on the ARMv8 architecture. AutoFFT is faster than FFTW and ARMPL for both single-precision (SP) and double-precision (DP) sequences. From the C2C FFTs' performance curves on the ARMv8 architecture in Fig.10, we can conclude the following. 1) These three libraries achieve similar performance trends. When the FFT size is small and the transformed data can reside in the cache system, the performance increases as the FFT size increases; when the FFT size is large and the cache system cannot hold all needed data, the cache miss rate is high; so the performance decreases as the FFT

TABLE 2: The C2C FFT average speedups obtained by AutoFFT on ARMv8 CPUs.

Data Size	Single Precision			Double Precision		
	FFTW	ARMPL	Baseline	FFTW	ARMPL	Baseline
2^n	1.53	3.01	3.24	1.72	2.56	2.63
Non-2^n	1.98	1.77	2.99	1.49	1.63	1.76

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2020.2977629, IEEE Transactions on Parallel and Distributed Systems

size increases. Hence, the performances of these libraries first increase and then decrease as the FFT size increases. 2) Compared with other libraries, the performance of AutoFFT stands out when processing FFTs of non-power-of-two sizes. In addition, the performance gaps between AutoFFT and the other libraries on single-precision sequences are larger than those on double-precision sequences. 3) Compared with AutoFFT and FFTW, ARMPL's performance gaps between the out-of-place transform and the corresponding in-place transform are larger. We believe that the performance of ARMPL's in-place transforms can be further improved. Table 2 lists the C2C FFT average speedups of AutoFFT compared with FFTW, ARMPL, and the baseline on ARMv8 CPUs.

TABLE 3: The C2C FFT average speedups obtained by AutoFFT on Intel Haswell CPUs.

Data Size	Single Precision			Double Precision		
	FFTW	MKL	Baseline	FFTW	MKL	Baseline
2^n	2.82	1.46	10.1	1.93	1.59	4.7
Non-2^n	2.84	1.52	8.32	1.97	1.72	4.44

Fig.11 shows the performances of the 1D C2C FFTs for AutoFFT, FFTW, Intel MKL, and the baseline on the Intel Haswell architecture. Based on the performance curves of C2C FFTs on the Intel Haswell architecture in Fig.11, we can conclude the following. 1) The performance trends among AutoFFT, FFTW, and Intel MKL are similar, but FFTW is generally slower than AutoFFT and Intel MKL. 2) Compared with AutoFFT's C kernels, its assembly kernels achieve higher speedup on Intel Haswell than on ARMv8. 3) Intel MKL performs very well when the FFT size is a power of two. For FFT sizes below 2048, AutoFFT is faster than Intel MKL. However, when the FFT size exceeds 2048, Intel MKL is close to that of AutoFFT, especially for DP floating-point data. There are two possible reasons for this difference in performance between AutoFFT and Intel MKL. First, in addition to the Cooley-Tukey algorithm, Intel MKL may adopt other FFT algorithms, such as the four-step FFT algorithm (or six-step, depending on the number of transpositions) [49], the split-radix [5], and the Rader-Brenner [6] algorithms, to obtain higher performance at large scales. Second, because our FFT kernels are autogenerated, they can be further improved by adopting instruction reordering and data prefetching. Table 3 lists the C2C FFT average speedups of AutoFFT compared with FFTW, ARMPL, and the baseline on Intel Haswell CPUs.

TABLE 4: The C2C FFT average speedups obtained by AutoFFT on AMD Zen CPUs.

Data Size	Single Precision		Double Precision	
	FFTW	Baseline	FFTW	Baseline
2^n	1.59	5.33	1.35	3
Non-2^n	1.92	5.96	1.61	2.77

Fig.12 shows the performances of the 1D C2C FFTs for AutoFFT, FFTW, and the baseline on the AMD Zen architecture. As shown in Fig.12, the performance of AutoFFT is relatively stable at different sizes, and even when the data size is large, it still achieves good performance. Besides, the performances of AutoFFT's in-place and out-of-place transforms are close and stable, however, most of FFTW's in-place transforms are slower than its corresponding out-of-place transforms. Table 4 lists the C2C FFT average speedups of AutoFFT compared with FFTW, ARMPL, and the baseline on AMD Zen CPUs. Compared with FFTW, AutoFFT has significant performance advantages on the AMD Zen architecture.

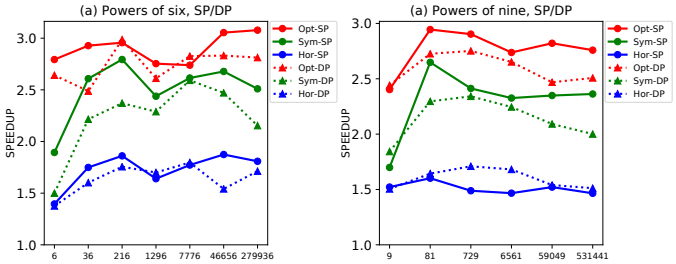


Fig. 13: These two subfigures present the performance speedups of *Opt*, *Sym*, and *Hor* over the DFT implementation of power-of-six and power-of-nine sizes respecively on the ARMv8 architecture. From these two subfigures, we can clearly see the empirical performance effect of the horizontal, vertical, and periodic optimizations.

As illustrated in section 4, the symmetric (the horizontal and vertical symmetries, described in subsection 4.1) and periodic (described in subsection 4.2) properties of the DFT matrix are adopted to reduce the number of floating-point operations of the butterflies. We call these three algorithmic optimizations as the horizontal optimization, the vertical optimization, and the periodic optimization, respectively. In this part, we evaluate the performance effect of these three optimizations by comparing them with the DFT implementation (the baseline for these three algorithmic optimizations). For simplicity, we define *Opt* to represent the fully optimized kernels (optimized by the horizontal, vertical, and periodic optimizations), *Sym* to represent the kernels optimized by the horizontal and vertical optimizations, and *Hor* to represent the kernels only optimized by the horizontal optimization. Fig.13 presents the performance speedups of *Opt*, *Sym*, and *Hor* by comparing them with the baseline. From this figure, we can see the performance improvements brought by these three optimizations step by step. Theoretically, for non-power-of-two radices, the horizontal and vertical symmetries reduce arithmetic operations by a factor of 4 (the horizontal and vertical optimizations can halve the amount of the arithmetic operations by combining like terms, respectively). However, *Sym* and *Hor* are only up to 2.79 times and 1.87 times faster than

TABLE 5: The average speedups of the symmetric (the horizontal and vertical symmetries) and periodic properties over the DFT implementation on ARMv8 CPUs.

Data Size	Single Precision			Double Precision		
	Opt	*Sym*	*Hor*	*Opt*	*Sym*	*Hor*
6^n	2.9	2.51	1.73	2.74	2.23	1.64
9^n	2.76	2.3	1.51	2.59	2.14	1.6

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2020.2977629, IEEE Transactions on Parallel and Distributed Systems

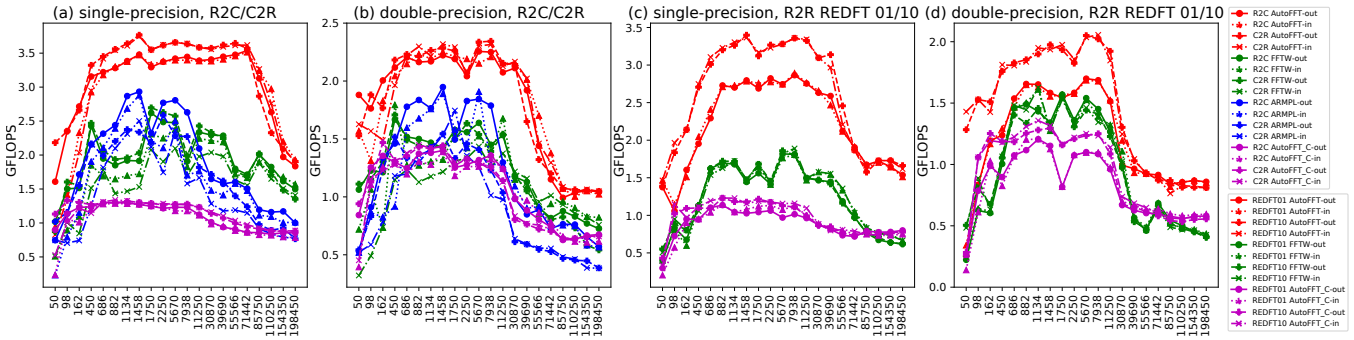


Fig. 14: The 1D real FFT performances on ARMv8 CPUs.

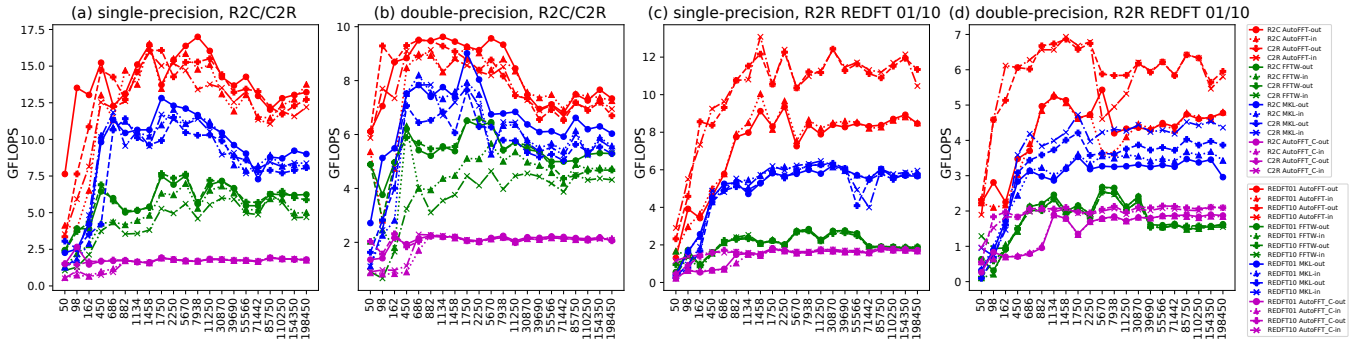


Fig. 15: The 1D real FFT performances on Intel Haswell CPUs.

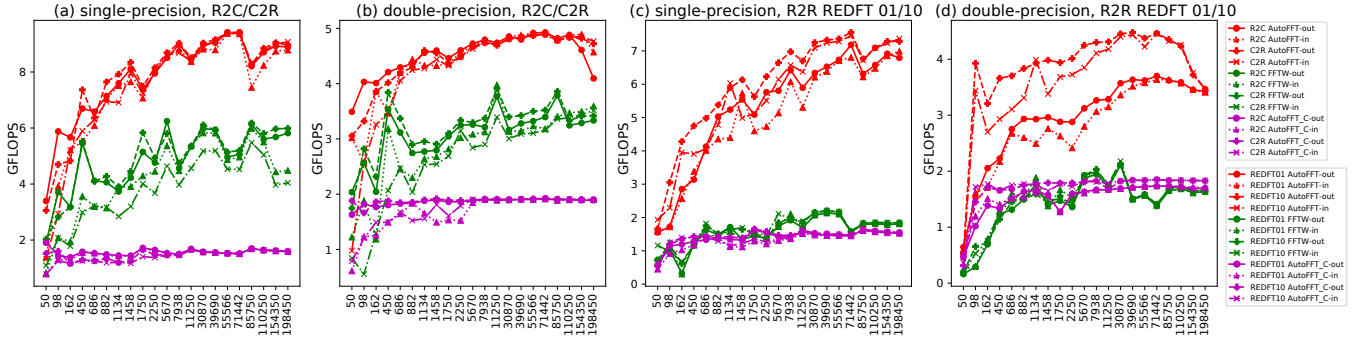


Fig. 16: The 1D real FFT performances on AMD Zen CPUs.

the baseline, respectively. The main reason why *Sym* can not achieve the theoretical 4x performance boost (and why *Hor* can not achieve the theoretical 2x performance boost) is that although we reduced the arithmetic operations of the Cooley-Tukey FFT kernels by a factor of 4, we did not reduce any memory operations. As we all know, memory operations are much more costly than arithmetic operations in modern CPUs. Because the periodic property can further reduce the arithmetic floating-point operations by a factor of $gcd(k, r)$ for each output Y_k of radix r, *Opt* achieves better performance than *Sym*. The performance curves in Fig.13 confirm this point: the red curves of these two subfigures are above the corresponding green curves. Compared with the baseline, the average speedups of *Opt*, *Sym*, and *Hor* over the baseline are listed in Table 5.

In many applications, the inputs or outputs are real numbers. AutoFFT currently supports the following transforms [50]: real-to-complex (R2C); complex-to-real (C2R); three types of real-to-real (R2R) transforms (real-to-"half-complex"/"half-complex"-to-real (R2HC/HC2R); the discrete Hartley transform (DHT); four kinds of discrete cosine transforms (DCT I∼IV); and four kinds of discrete sine transforms (DST I∼IV)). To reduce unnecessary computations and memory access, we adopt two reduction methods to summarize and extract unified optimization patterns of real FFTs. First, the complex reduction. For FFTs with N real number inputs, the complex reduction is defined to take N real numbers as $N/2$ complex numbers and perform the $N/2$-point complex FFT, after that, split operations are applied on the transformed results. Each real FFT conducts a split operation according to its definitions. Second, the real reduction. Because DCT/DST contains special symmetric properties in its inputs, the real reduction is defined to reduce a N-point real FFT to a $N/2$-point real FFT. Due to the space limitations of this paper, we present the performances of the R2C/C2R and R2R REDFT01/REDFT10 real

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2020.2977629, IEEE Transactions on Parallel and Distributed Systems

TABLE 6: The real FFT average speedups obtained by AutoFFT on ARMv8, Intel Haswell, and AMD Zen architectures.

Architecture	FFT Type	Single Precision				Double Precision			
		FFTW	ARMPL	MKL	Baseline	FFTW	ARMPL	MKL	Baseline
ARMv8	R2C/C2R	1.71	2.01	-	2.82	1.61	1.91	-	1.77
	R2R	2.04	-	-	2.66	1.51	-	-	1.58
Intel Haswell	R2C/C2R	2.54	-	1.64	7.98	1.9	-	1.47	4.14
	R2R	4.42	-	2.1	5.99	3.57	-	2.06	3.07
AMD Zen	R2C/C2R	1.76	-	-	5.09	1.56	-	-	2.48
	R2R	3.54	-	-	3.94	2.47	-	-	2

FFTs below.

In Table 6, we provide the real FFT average speedups on ARMv8, Intel Haswell, and AMD Zen architectures. 1) On the ARMv8 architecture, because ARMPL does not support real FFTs other than R2C/C2R FFTs, we provide the performances of only ARMPL's R2C/C2R FFTs here. Fig.14 shows that the performances of AutoFFT's real FFTs outperform those of FFTW, ARMPL, and the baseline, especially when the FFT size is large. In general, most of the characteristics of the real FFTs' performance curves are similar to the C2C FFTs' performance curves on the ARMv8 architecture. However, compared with other libraries, ARMPL's performance curves of R2C/C2R FFTs decrease faster as the FFT size increases. Its performance curves are very close to or even lower than those of the baseline at large scales, especially for the in-place transforms. 2) Fig.15 shows the performances of real FFTs of AutoFFT, FFTW, Intel MKL, and the baseline on the Intel Haswell architecture. For R2C/C2R FFTs, FFTW is generally slower than AutoFFT and Intel MKL, and AutoFFT is faster than MKL FFT in most cases, as shown in Fig.15(a) and Fig.15(b). For R2R REDFT01/REDFT10, AutoFFT outperforms the other two libraries, and FFTW is much slower than AutoFFT and Intel MKL, and the R2R FFT performances of FFTW and the baseline are very close, as shown in Fig.15(c) and Fig.15(d). 3) Fig.16 presents the performances of real FFTs of AutoFFT, FFTW and the baseline on the AMD Zen architecture. The performances in Fig.16 show that AutoFFT outperforms FFTW, and we believe that AutoFFT is the best choice for the AMD Zen CPUs. Different from R2C/C2R FFTs in Fig.16(a) and Fig.16(b), we can see that the R2R FFT performances of FFTW and the baseline are very close, as shown in Fig.16(c) and Fig.16(d). This phenomenon also exists in Intel Haswell architecture, and we believe that FFTW can further optimize real FFTs and achieve better performances on these two architectures.

TABLE 7: The average and maximum speedups of AutoFFT.

Architecture	Software	Average	Max
ARMv8	FFTW	1.7	2.04
	ARMPL	2.15	3.01
	Baseline	2.43	3.24
Intel Haswell	FFTW	2.75	4.42
	MKL	1.7	2.1
	Baseline	6.09	10.1
AMD Zen	FFTW	1.98	3.54
	Baseline	3.82	5.96

At the end of this section, we summarize the average and maximum performance improvements of AutoFFT by comparing it with FFTW, ARMPL, Intel MKL, and the baseline on ARMv8, Intel Haswell, and AMD Zen architectures in Table 7.

8 Conclusion

This paper proposes a template-based framework named AutoFFT that makes full use of the experience of the domain and optimization experts to automatically generate extremely high-performance FFT code of radices of all natural numbers for ARM, Intel, and AMD platforms. AutoFFT thus substantially reduces the laborious work of developing assembly kernels manually. The experiments show that AutoFFT performs generally better than FFTW, ARMPL, and Intel MKL. Our future work will concentrate on extending the template-based methodology to other numerical algorithms.

Acknowledgments

This work is supported by the National Key Research and Development Program of China under Grant Nos. 2107YFB0202105, 2016YFB0200803, and 2017YFB0202302; the National Natural Science Foundation of China under Grant Nos. 61602443, 61432018, 61521092, and 61502450.

References

[1] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[2] C. M. Rader, "Discrete fourier transforms when the number of data samples is prime," *Proceedings of the IEEE*, vol. 56, no. 6, pp. 1107–1108, 1968.

[3] L. Bluestein, "A linear filtering approach to the computation of discrete fourier transform," *IEEE Transactions on Audio and Electroacoustics*, vol. 18, no. 4, pp. 451–455, 1970.

[4] D. Kolba and T. Parks, "A prime factor fft algorithm using high-speed convolution," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 25, no. 4, pp. 281–294, 1977.

[5] P. Duhamel and H. Hollmann, "Split radix'fft algorithm," *Electronics letters*, vol. 20, no. 1, pp. 14–16, 1984.

[6] C. Rader and N. Brenner, "A new principle for fast fourier transformation," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 24, no. 3, pp. 264–266, 1976.

[7] J. W. Cooley, P. A. Lewis, and P. D. Welch, "The fast fourier transform and its applications," *IEEE Transactions on Education*, vol. 12, no. 1, pp. 27–34, 1969.

[8] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from berkeley," Technical Report UCB/EECS-2006-183, EECS Department, University of . . . , Tech. Rep., 2006.

[9] P. Costa, "A fft-based finite-difference solver for massively-parallel direct numerical simulations of turbulent flows," *Computers & Mathematics with Applications*, vol. 76, no. 8, pp. 1853–1862, 2018.

[10] Z. Li, H. Jia, Y. Zhang, S. Liu, S. Li, X. Wang, and H. Zhang, "Efficient parallel optimizations of a high-performance sift on gpus," *Journal of Parallel and Distributed Computing*, vol. 124, pp. 78–91, 2019.

[11] D. Zhang, Z. Chen, C. Xiao, M. Qin, and H. Wu, "Accurate simulation of turbulent phase screen using optimization method," *Optik*, vol. 178, pp. 1023–1028, 2019.

[12] C. Gong, W. Bao, and G. Tang, "A parallel algorithm for the riesz fractional reaction-diffusion equation with explicit finite difference method," *Fractional Calculus and Applied Analysis*, vol. 16, no. 3, pp. 654–669, 2013.

[13] C. Gong, W. Bao, G. Tang, B. Yang, and J. Liu, "An efficient parallel solution for caputo fractional reaction–diffusion equation," *The Journal of Supercomputing*, vol. 68, no. 3, pp. 1521–1537, 2014.

[14] M. Frigo and S. G. Johnson, "The fastest Fourier transform in the west," Massachusetts Institute of Technology, Tech. Rep. MIT-LCS-TR-728, September 1997.

[15] M. Frigo and S. G. Johnson, "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, May 1998, pp. 1381–1384 vol.3.

[16] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[17] ARM, "Arm performance libraries (armpl) 19.2.0," https://static.docs.arm.com/101004/1920/arm_performance_libraries_reference_101004_1920_00_en.pdf, 2019.

[18] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, "Intel math kernel library," in *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 2014, pp. 167–188.

[19] Intel, "Intel math kernel library (intel mkl) 2019 update 4," https://software.intel.com/en-us/mkl, 2019.

[20] Z. Li, H. Jia, Y. Zhang, T. Chen, L. Yuan, L. Cao, and X. Wang, "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*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356138

[21] G. Bruun, "z-transform dft filters and fft's," *IEEE Transactions on Acoustics Speech and Signal Processing*, vol. 26, no. 1, pp. 56–63, 1978.

[22] J. Xiong, J. Johnson, R. Johnson, and D. Padua, "Spl: A language and compiler for dsp algorithms," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, ser. PLDI '01. New York, NY, USA: ACM, 2001, pp. 298–308. [Online]. Available: http://doi.acm.org/10.1145/378795.378860

[23] AMD, "Aocl: Amd optimizing cpu libraries," https://developer.amd.com/amd-aocl/, 2020.

[24] IBM, "Essl: Ibm engineering and scientific subroutine library," https://www.ibm.com/support/knowledgecenter/en/SSFHY8_6.1/navigation/welcome.html, 2020.

[25] Apple, "The apple accelerate libraries - vdsp," https://developer.apple.com/documentation/accelerate/vdsp/fast_fourier_transforms, 2020.

[26] D. Mirković, R. Mahasoom, and L. Johnsson, "An adaptive software library for fast fourier transforms," in *Proceedings of the 14th International Conference on Supercomputing*, ser. ICS '00. New York, NY, USA: ACM, 2000, pp. 215–224. [Online]. Available: http://doi.acm.org/10.1145/335231.335252

[27] A. Ali and L. Johnsson, "Uhfft: A high performance dft framework," 2006.

[28] M. Puschel, J. M. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko *et al.*, "Spiral: Code generation for dsp transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

[29] F. Franchetti, T. M. Low, D. T. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, and J. M. Moura, "Spiral: Extreme performance portability," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1935–1968, 2018.

[30] F. Franchetti, Y. Voronenko, and M. Püschel, "Formal loop merging for signal transforms," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 315–326, 2005.

[31] ARM, "Arm ne10 project." https://github.com/projectNe10/Ne10, 2020.

[32] D. Takahashi, "Ffte: A fast fourier transform package," http://www.ffte.jp/, 2014.

[33] A. Blake and M. Hunter, "Dynamically generating fft code," *Journal of Signal Processing Systems*, vol. 76, no. 3, pp. 275–281, 2014.

[34] Nvidia, "Cufft library," https://docs.nvidia.com/pdf/CUFFT_Library.pdf, 2020.

[35] AMD, "A software library containing fft functions written in opencl," https://github.com/clMathLibraries/clFFT, 2020.

[36] D. Petre, A. T. Lake, and A. Hux, "Opencl™ fft optimizations for intel® processor graphics," in *Proceedings of the 4th International Workshop on OpenCL*, ser. IWOCL '16. New York, NY, USA: ACM, 2016, pp. 12:1–12:4. [Online]. Available: http://doi.acm.org/10.1145/2909437.2909451

[37] A. Nukada, Y. Maruyama, and S. Matsuoka, "High performance 3-d fft using multiple cuda gpus," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, ser. GPGPU-5. New York, NY, USA: ACM, 2012, pp. 57–63. [Online]. Available: http://doi.acm.org/10.1145/2159430.2159437

[38] Y. Dotsenko, S. S. Baghsorkhi, B. Lloyd, and N. K. Govindaraju, "Auto-tuning of fast fourier transform on graphics processors," *SIGPLAN Not.*, vol. 46, no. 8, pp. 257–266, Feb. 2011. [Online]. Available: http://doi.acm.org/10.1145/2038037.1941589

[39] Y. Li, Y.-Q. Zhang, Y.-Q. Liu, G.-P. Long, and H.-P. Jia, "Mpfft: An autotuning fft library for opencl gpus," *Journal of Computer Science and Technology*, vol. 28, no. 1, pp. 90–105, 2013.

[40] A. Gholami, J. Hill, D. Malhotra, and G. Biros, "Accfft: A library for distributed-memory FFT on CPU and GPU architectures," *CoRR*, vol. abs/1506.07933, 2015. [Online]. Available: http://arxiv.org/abs/1506.07933

[41] C. Cecka, "Low communication fmm-accelerated fft on gpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 54:1–54:11. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126919

[42] T. G. Stockham, Jr., "High-speed convolution and correlation," in *Proceedings of the April 26-28, 1966, Spring Joint Computer Conference*, ser. AFIPS '66 (Spring). New York, NY, USA: ACM, 1966, pp. 229–233. [Online]. Available: http://doi.acm.org/10.1145/1464182.1464209

[43] P. N. Swarztrauber, "Vectorizing the ffts," in *Parallel Computations*, G. RODRIGUE, Ed. Academic Press, 1982, pp. 51 – 83. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780125921015500075

[44] Intel, "Intel 64 and ia-32 architectures optimization reference manual (chapter 2.1)," https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, 2016.

[45] A. V. Oppenheim, *Discrete-time signal processing*. Pearson Education India, 1999.

[46] M. Frigo and S. G. Johnson, "The benchmarking methodology of benchfft," http://www.fftw.org/speed/, 2020.

[47] M. Frigo and S. Johnson, "benchfft," http://www.fftw.org/benchfft, 2020.

[48] Intel, "Intel math kernel library developer reference's appendix c: Fftw interface to intel math kernel library," https://software.intel.com/sites/default/files/mkl-2019-developer-reference-c_2.pdf, 2020.

[49] D. H. Bailey, "Ffts in external or hierarchical memory," *The journal of Supercomputing*, vol. 4, no. 1, pp. 23–35, 1990.

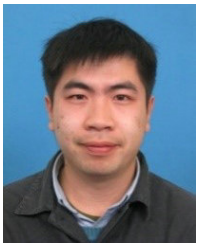
[50] X. Wang, H. Jia, Z. Li, and Y. Zhang, "Implementation and optimization of multi-dimensional real fft on armv8 platform," in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2018, pp. 338–353.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2020.2977629, IEEE Transactions on Parallel and Distributed Systems

Zhihao Li is currently working toward the Ph.D. degree at the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences. In addition, he is a joint Ph.D. student at the School of Computational Science and Engineering, Georgia Institute of Technology. His research interests include parallel computing, high-performance FFT library, and many-core/large-scale parallel programming method.

Haipeng Jia received the Ph.D. degree from the Ocean University of China in 2012. He was a visiting Ph.D. student with the Institute of Software, Chinese Academy of Sciences from 2010 to 2012. He is currently an Assistant Professor with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences. His research interests include heterogeneous computing and many-core parallel programming method.

Yunquan Zhang received the Ph.D. degree in computer software and theory from the Chinese Academy of Sciences in 2000. He is a Full Professor of computer science with the Institute of Computing Technology, Chinese Academy of Sciences. His research interests are in the areas of high performance computing, with particular emphasis on large scale parallel computation and programming models, and high-performance parallel numerical algorithms.



Tun Chen received the master degree in computer science from Hunan Normal University in 2018. He is currently working toward the Ph.D. degree at the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences. His research interests include high performance computing, heterogeneous computing, in-core parallelism, and optimized FFT library.

Liang Yuan received the Ph.D. degree from the Institute of Software, Chinese Academy of Sciences in 2013. He was a visiting Ph.D. student with the University of Rochester in 2011. He is currently an Assistant Professor with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences. His research interests include large-scale parallel computing and heterogeneous computing.

Richard Vuduc is a professor in the School of Computational Science and Engineering at the Georgia Institute of Technology. He received his Ph.D. in computer science from the University of California, Berkeley, and his B.S. in computer science from Cornell University. His research lab, the HPC Garage, is interested in high-performance computing, with an emphasis on algorithms, performance analysis, and performance engineering.