AutoFFT: A Template-Based FFT Codes Auto-Generation Framework for ARM and X86 CPUs

Zhihao Li
SKL of Computer Architecture,
Institute of Computing Technology,
Chinese Academy of Sciences
University of Chinese Academy of Sciences
lizhihao@ict.ac.cn

Haipeng Jia*
SKL of Computer Architecture,
Institute of Computing Technology,
Chinese Academy of Sciences
jiahaipeng@ict.ac.cn

Yunquan Zhang
SKL of Computer Architecture,
Institute of Computing Technology,
Chinese Academy of Sciences
zyq@ict.ac.cn

Tun Chen
SKL of Computer Architecture,
Institute of Computing Technology,
Chinese Academy of Sciences
University of Chinese Academy of Sciences
chentun@ict.ac.cn

Liang Yuan
Luning Cao
SKL of Computer Architecture,
Institute of Computing Technology,
Chinese Academy of Sciences
{yuanliang,caoluning}@ict.ac.cn

Xiao Wang
SKL of Computer Architecture,
Institute of Computing Technology,
Chinese Academy of Sciences
University of Chinese Academy of Sciences
wangxiao@ict.ac.cn

ABSTRACT

The discrete Fourier transform (DFT) is widely used in scientific and engineering computation. This paper proposes a template-based code generation framework named AutoFFT that can automatically generate high-performance fast Fourier transform (FFT) codes. AutoFFT employs the Cooley-Tukey FFT algorithm, which exploits the symmetric and periodic properties of the DFT matrix as the outer parallelization framework. To further reduce the number of floating-point operations of butterflies, we explore more symmetric and periodic properties of the DFT matrix and formulate two optimized calculation templates for prime and power-of-two radices. 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 two templates and transfers them to efficient assembly codes using the template optimizer. Experiments show that AutoFFT outperforms FFTW, ARMPL, and Intel MKL on average across all FFT types on ARMv8 and Intel x86-64 processors.

CCS CONCEPTS

• **Software and its engineering** → *Source code generation*; *Assembly languages*; • **Computing methodologies** → *Vector / streaming algorithms*.

*Corresponding author

SC '19, November 17–22, 2019, Denver, CO, USA

ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356138

KEYWORDS

AutoFFT, FFT, code generation, template, DFT

ACM Reference Format:
Zhihao Li, Haipeng Jia, Yunquan Zhang, Tun Chen, Liang Yuan, Luning Cao, and Xiao Wang. 2019. AutoFFT: A Template-Based FFT Codes Auto-Generation Framework for ARM and X86 CPUs. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3295500.3356138

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, $k \in [0, n-1]$, and $j = \sqrt{-1}$. $W_n = e^{-\frac{2\pi j}{n}}$ is called the twiddle factor (twiddle for short), and $(W_n^{ik})_{n\times n}$ is called the DFT matrix, as presented in Eq.2. The DFT is essentially a matrix-vector operation, and the computational complexity of the naïve matrix-vector implementation is $O(n^2)$.

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

Many previous studies [9, 13, 16, 31, 40, 41] 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 [13] is the most widely used

fast Fourier transform (FFT) algorithm in many practical applications [7, 12, 14, 26, 27, 33, 47]. It adopts a divide-and-conquer approach to recursively break down a large DFT into smaller DFTs, achieving an $O(n \log n)$ complexity. 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"). 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 codes in assembly by capitalizing on the symmetric and periodic properties of the DFT matrix.

First, AutoFFT defines highly optimized patterns (the pair and quad patterns) with a reduced number of floating-point operations for butterflies. 1) We explore more symmetric properties of the DFT matrix for the radix-r butterfly. In particular, we investigate the symmetric properties in the horizontal and vertical directions for two types of radices (the prime and power-of-two radices). By observing the calculation of each Y_k, Y_k's twiddles are symmetric in the horizontal direction, and we reduce the multiplications for prime and power-of-two radices by factors of 2 and 4, respectively. In the vertical direction, some outputs share a similar symmetric feature, and we can further reduce the arithmetic complexity for prime and power-of-two radices by factors of 2 and 4, respectively. 2) We study a periodic property inside one Y_k for a power-of-two radix r when k is an even number. By exploiting this property, the operations can be further reduced by a factor of $gcd(k, r)$, where gcd is the greatest common divisor.

Furthermore, AutoFFT generates high-performance FFT kernels based on the pair and quad patterns. By formalizing the optimization experience of domain experts, computational templates are defined based on the pair and quad patterns. These computational templates are used by a C FFT kernel generator to generate C FFT kernels. Subsequently, an assembly template optimizer composed of deeply optimized optimization templates is designed to translate C kernels into high-performance assembly FFT kernels for varying architectures.

At runtime, to obtain the best factorization 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 factorization and invokes the required FFT kernels to perform the FFT computation.

The key contributions of this paper are summarized as follows:

- We propose a template-based code generation framework that summarizes the experience of domain and optimization experts to automatically generate high-performance assembly FFT kernels for varying CPU architectures.
- We exploit the symmetric and periodic properties of the DFT matrix and derive two butterfly calculation patterns to minimize the number of floating-point operations: the pair pattern for prime radices and the quad pattern for power-of-two radices. Other libraries can also accelerate their butterflies based on these two optimized patterns.
- We implement a high-performance FFT library based on our framework for ARMv8 and Intel x86-64 CPUs.

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 presents the experimental results. Finally, Section 7 concludes the paper.

2 RELATED WORK

Many FFT algorithms [9, 10, 13, 16, 31, 40, 41, 46] have been proposed to compute the DFT. As the most popular FFT algorithm, the Cooley-Tukey algorithm [13] is supported by most mainstream FFT libraries. These libraries are implemented and optimized by vendors or researchers to achieve high performance on specific hardware architectures.

These highly efficient vendor-supplied FFT libraries, which include ARM Performance Library (ARMPL) [6], Intel Math Kernel Library (MKL) [45], IBM Engineering and Scientific Subroutine Library (ESSL) [28], and Apple vDSP [4], are appropriately accelerated using single instruction multiple data (SIMD) techniques for specific architectures. These implementations have undergone extensive architecture-dependent tuning and optimizations on specific microarchitecture innovations and 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. The Fastest Fourier Transform in the West (FFTW) [22–24] and UHFFT [1, 34] 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 [17, 39] is a three-stage framework. In the first stage, SPIRAL expresses the mathematical formulas in the Signal Processing Language (SPL) [46] for a given transform. The second stage converts the SPL formulas into Σ-SPL [19], and then performs various computational optimizations such as code reordering in Σ-SPL. Finally, the optimized

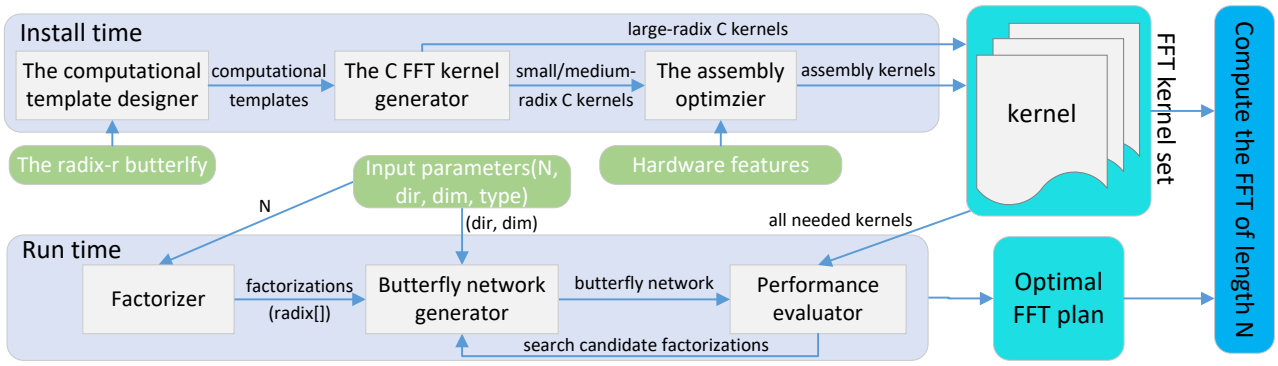

Figure 1: Overview of AutoFFT.

Σ-SPL is translated into source code suitable for the given architecture. This code may contain features such as SIMD instructions and multithreading [18, 38]. Other open-source FFT implementations include Ne10 [5], 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) [44], which is a Fortran FFT package that supports problem sizes of the form $2^a 3^b 5^c$ for CPUs and GPUs, and the Fastest Fourier Transform in the South (FFTS) [8], which is a hand-optimized library for Intel and ARM CPUs.

In addition, 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 [36], clFFT for AMD GPUs [3], and genFFT for Intel processor graphics [37], researchers have developed several FFT implementations [11, 15, 25, 32, 35]. 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 butterfly 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.

This paper uses ARMPL, Intel MKL, and FFTW to compare AutoFFT on the ARMv8 and Intel x86-64 processors. 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. It is known that AMD adopts FFTW as the FFT module of their official math libraries—AMD Optimizing CPU Libraries (AOCL) [2]. In contrast to vendor-tuned libraries, FFTW's performance is portable, which means that FFTW performs well on most architectures. Compared with these state-of-the-art libraries, our template-based code autogeneration approach contains the following advantages. First, our templates are extracted and optimized based on two butterfly calculation patterns with the lowest arithmetic complexity. Second, the efficiency and portability of AutoFFT are good. Benefiting from the code generation mechanism, AutoFFT can rapidly generate high-performance assembly FFT kernels when a new architecture emerges.

3 THE AUTOFFT FRAMEWORK

This section introduces the two stages of AutoFFT's framework, 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.

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 much as possible, the pair and quad patterns are derived based on the symmetric and periodic properties of the DFT matrix $(W_r^{ik})_{r\times r}$. The install-time stage adopts three components to capitalize on these two patterns:

(1) The computational template designer. A computational template designer is introduced according to these two optimized patterns to extract high-level computational templates.
(2) The C FFT kernel generator. This generator takes computational templates as inputs and generates efficient FFT kernels for all types of radices.
(3) The assembly optimizer. The optimizer defines optimization templates and small/medium/large radices (described in subsubsection 5.3) that match the underlying hardware capabilities, such as register resources and the instruction set. To further exploit the hardware resources, the optimizer translates the C FFT kernels for small and medium radices into FFT kernels using optimization templates.

Due to the limited register resources in modern CPUs, the C FFT kernels for large radices are difficult to autotranslate into efficient assembly codes. Hence, we eventually obtain an FFT kernel set composed of assembly FFT kernels for small/medium radices and C FFT kernels for large radices. 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

The runtime stage is responsible for seeking the optimal plan for a given DFT problem that can reduce searching for the best factorization of the given problem size. Because the search space of

factorizations can be large, it is necessary to seek the optimal plan in a reasonable time. The runtime stage consists of three components:

(1) The number factorizer. The factorizer generates the factorization tree for a given FFT size. Each branch of the tree represents one factorization that holds the radix for each stage of the butterfly network in *radix*[].
(2) 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 constructs the butterfly network based on the necessary input parameters (*radix*[], the transform direction, and dimension).
(3) 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 (factorization) 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 factorization 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 auto-sort butterfly network according to the best factorization and call the needed FFT kernels to efficiently perform the FFT computation for a given DFT problem. The auto-tuning framework of AutoFFT refers to the current state-of-the-art [1, 24], 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 their optimized 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 autogeneration. In this section, we systematically summarize the symmetry and periodicity of butterflies (radices) and extract relatively optimized calculation patterns.

We divide radix r into two types: prime radices and power-of-two radices (other radices can be represented by combining prime and power-of-two radices). Based on these two types of radices, AutoFFT can perform FFTs of arbitrary sizes. By capitalizing on twiddles' symmetric and periodic properties, we formulate two highly optimized butterfly calculation patterns: the pair pattern for prime radices and the quad pattern for power-of-two radices. According to Eq.1, radix r consists of $4r^2$ multiplications and $4r^2-2r$ additions of real numbers.

4.1 The Pair Pattern for Prime Radices

Let r be a prime number, and denote $r = 2m + 1$. According to Eq.1, when calculating Y_0 for the radix-r butterfly, $W_r^{ik} = W_r^0 = 1$; therefore, Y_0 is calculated separately using Eq.4.

$$Y_0 = x_0 + \sum_{i=1}^{m}(x_i + x_{r-i}) \tag{4}$$

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.

4.1.1 ***The Horizontal Symmetry.*** W_r^{ik} and W_r^{-ik} are symmetric about the x-axis, 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.

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

4.1.2 ***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 the pair results Y_k and Y_{r-k} together. This optimized calculation pattern for prime radices is called **the pair pattern**.

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

To formalize the pair pattern, we define P_k and Q_k to represent the common calculation terms of Y_k and Y_{r-k}. After obtaining P_k and Q_k, Y_k and Y_{r-k} can be calculated together. The general definition of the pair pattern is presented in Eq.7.

$$\begin{cases} P_k = \sum_{i=1}^{m}(x_i + x_{r-i})a_{ik} \\ Q_k = \sum_{i=1}^{m}(x_i - x_{r-i})b_{ik} \cdot j \\ Y_k = x_0 + P_k + Q_k \\ Y_{r-k} = x_0 + P_k - Q_k \end{cases} \quad k \in [1, m] \tag{7}$$

The pair pattern minimizes the number of floating-point operations of prime radices, as presented in Table.1: 1) As shown in Eq.7, for each Y_k, P_k and Q_k require $2m$ real multiplications. Because $k \in [1, m]$, $r^2 - 2r + 1$ real multiplications are required. 2) For each

Y_k, the pair pattern requires $8m-4$ and 6 real additions in Eq.7. In addition, because the sum term in Eq.4 is obtained when calculating P_k, Eq.4 only requires 2 real additions for Y_0; therefore, a total of $2r^2-3r+3$ real additions are required.

4.2 The Quad Pattern for Power-of-two Radices

Let r be a power of two. Because Y_0, $Y_{r/2}$, $Y_{r/4}$, and $Y_{3r/4}$ can be calculated separately, we denote that $r=4m+4$. Regarding Y_0 and $Y_{r/2}$, because twiddles contain symmetric and periodic properties, 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.8. ($Y_{r/4}$ and $Y_{3r/4}$ are calculated in subsubsection 4.2.3).

$$\begin{aligned} Y_0 &= \sum_{i=0}^{r-1} x_i W_r^{i\cdot 0} = \sum_{i=0}^{r/2-1} (x_i + x_{r-i-1}) \\ Y_{r/2} &= \sum_{i=0}^{r-1} x_i W_r^{i\cdot r/2} = \sum_{i=0}^{r-1} (-1)^i x_i = \sum_{i=0}^{r/2-1} (x_{2i} - x_{2i+1}) \end{aligned} \tag{8}$$

Compared with prime radices, power-of-two radices have better symmetry and better periodicity under certain conditions. When r is a power of two, 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. Assuming $W_r^{ik} = a_{ik} + b_{ik}\cdot j$ and $W_r^{(r-i)k} = W_r^{-ik} = a_{ik} - b_{ik}\cdot j$, we further define $W_r^{(r/2-i)k} = (-1)^k(a_{ik} - b_{ik}\cdot j)$ and $W_r^{(r/2+i)k} = (-1)^k(a_{ik} + b_{ik}\cdot j)$. For the periodic property, when we calculate Y_k with $gcd(r,k)\neq 1$, Y_k's twiddles contain better periodicity. Therefore, we can optimize the calculation of power-of-two radices by capitalizing on the horizontal symmetry, the vertical symmetry, and the periodic property.

4.2.1 ***The Horizontal Symmetry.*** As $W_r^{(r/2-i)k} = (-1)^k W_r^{-ik}$ and $W_r^{(r/2+i)k} = (-1)^k W_r^{ik}$, Y_k's calculation can be reduced by uniting the 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.9 where $k\in[1,m]$.

$$\begin{aligned} Y_k =& x_0 + (-1)^k x_{r/2} + x_{r/4}W_r^{r/4\cdot k} + x_{3r/4}W_r^{3r/4\cdot k} \\ &+ \sum_{i=1}^{m}((x_i + x_{r-i}) + (-1)^k(x_{r/2-i} + x_{r/2+i}))a_{ik} \\ &+ \sum_{i=1}^{m}((x_i - x_{r-i}) - (-1)^k(x_{r/2-i} - x_{r/2+i}))b_{ik}\cdot j \end{aligned} \tag{9}$$

4.2.2 ***The Vertical Symmetry.*** Similar to Eq.9, Y_{r-k}, $Y_{r/2-k}$, and $Y_{r/2+k}$ can be calculated using Eq.10. 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. Like the prime radices, $m(4m+8) = r^2/4 - 4$ real multiplications and $r^2-5r+16$ real additions are required.

$$\begin{aligned} Y_{r-k} =& x_0 + (-1)^k x_{r/2} + x_{r/4}W_r^{r/4\cdot(r-k)} + x_{3r/4}W_r^{3r/4\cdot(r-k)} \\ &+ \sum_{i=1}^{m}((x_i + x_{r-i}) + (-1)^k(x_{r/2-i} + x_{r/2+i}))a_{ik} \\ &- \sum_{i=1}^{m}((x_i - x_{r-i}) - (-1)^k(x_{r/2-i} - x_{r/2+i}))b_{ik}\cdot j \\ Y_{r/2-k} =& x_0 + (-1)^k x_{r/2} + x_{r/4}W_r^{r/4\cdot(r/2-k)} + x_{3r/4}W_r^{3r/4\cdot(r/2-k)} \\ &+ \sum_{i=1}^{m}(-1)^i((x_i + x_{r-i}) + (-1)^k(x_{r/2-i} + x_{r/2+i}))a_{ik} \\ &- \sum_{i=1}^{m}(-1)^i((x_i - x_{r-i}) - (-1)^k(x_{r/2-i} - x_{r/2+i}))b_{ik}\cdot j \\ Y_{r/2+k} =& x_0 + (-1)^k x_{r/2} + x_{r/4}W_r^{r/4\cdot(r/2+k)} + x_{3r/4}W_r^{3r/4\cdot(r/2+k)} \\ &+ \sum_{i=1}^{m}(-1)^i((x_i + x_{r-i}) + (-1)^k(x_{r/2-i} + x_{r/2+i}))a_{ik} \\ &+ \sum_{i=1}^{m}(-1)^i((x_i - x_{r-i}) - (-1)^k(x_{r/2-i} - x_{r/2+i}))b_{ik}\cdot j \end{aligned} \tag{10}$$

4.2.3 ***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.11 by taking advantage of the periodic property.

$$\begin{aligned} Y_k &= \sum_{i=0}^{r-1} x_i W_r^{ik} = \sum_{i=0}^{r-1} x_i W_r^{(ik \bmod r)} \\ &= \sum_{s=0}^{p-1}\sum_{t=0}^{q-1} x_{t+s\cdot q}W_r^{tk} = \sum_{t=0}^{q-1}(\sum_{s=0}^{p-1} x_{t+s\cdot q})\cdot W_r^{tk} \end{aligned} \tag{11}$$

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.11.

We define a new complex sequence $\hat{x}$ to represent the inner summation of Eq.11. 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.12 with $t\in[0,q-1]$; thus, the Y_k in Eq.11 can be re-expressed as the Y_k in Eq.12.

$$\begin{aligned} \hat{x}_{(t,k)} &= \sum_{s=0}^{p-1} x_{t+s\cdot q} \\ Y_k &= \sum_{t=0}^{q-1} \hat{x}_{(t,k)}\cdot W_r^{tk} \end{aligned} \tag{12}$$

To formulate and capitalize on the horizontal symmetry of Eq.9 in Eq.12, we need to parameterize the calculation of Eq.9 using A,

Table 1: The number of real multiplications and additions among the DFT, the pair pattern, and the quad pattern.

Method	# Multiplication	# Addition
DFT	$4r^2$	$4r^2 - 2r$
The pair pattern	$r^2 - 2r + 1$	$2r^2 - 3r + 3$
The quad pattern	$\sum_{k=1}^{r/4-1}(r+4)/p$	$r^2 - 5r + 16$

B, C, and D, as shown in Eq.13. Based on these parameters, the Y_k in Eq.12 is re-expressed as Eq.14.

$$\begin{aligned} A &= \hat{x}_{(0,k)} - \hat{x}_{(q/2,k)} \\ B &= (\hat{x}_{(q/4,k)} - \hat{x}_{(3q/4,k)}) \cdot Im(W_{4p}^{k}) \cdot j \\ C &= ((\hat{x}_{(i,k)} + \hat{x}_{(q-i,k)}) + (-1)^k(\hat{x}_{(q/2-i,k)} + \hat{x}_{(q/2+i,k)}))a_{ik} \\ D &= ((\hat{x}_{(i,k)} - \hat{x}_{(q-i,k)}) - (-1)^k(\hat{x}_{(q/2-i,k)} - \hat{x}_{(q/2+i,k)}))b_{ik} \cdot j \end{aligned} \tag{13}$$

$$Y_k = A + B + \sum_{i=1}^{m} C + \sum_{i=1}^{m} D \tag{14}$$

In addition, Y_{r-k}, $Y_{r/2-k}$, and $Y_{r/2+k}$ can be optimized by the periodic property. Based on the vertical symmetry, these calculations are re-expressed in Eq.15. This optimized calculation pattern for power-of-two radices is called **the quad pattern**. Note that $Y_{r/4}$ and $Y_{3r/4}$ are calculated separately. When $k = r/4$, $p = q = 4$; therefore, we obtain $Y_{r/4} = A + B$, and $Y_{3r/4} = A - B$.

$$\begin{aligned} Y_{r-k} &= A - B + \sum_{i=1}^{m} C - \sum_{i=1}^{m} D \\ Y_{r/2-k} &= A - B + \sum_{i=1}^{m} (-1)^i \cdot C - \sum_{i=1}^{m} (-1)^i \cdot D \\ Y_{r/2+k} &= A + B + \sum_{i=1}^{m} (-1)^i \cdot C + \sum_{i=1}^{m} (-1)^i \cdot D \end{aligned} \tag{15}$$

The periodicity further reduces the number of real multiplications by a factor of p when $p > 1$. Hence, the quad pattern requires $\sum_{k=1}^{m}(4m+8)/p$ real multiplications that are fewer than $r^2/4 - 4$.

5 FFT KERNEL GENERATION

5.1 The Computational Template Designer

To adopt the pair pattern and the quad pattern in the process of code generation, we introduce a computational template designer by extracting the formalized expressions of these two patterns. The designer constructs computational templates that consist of meta templates and hybrid templates. Note that the input sequence x is equal to the original input data multiplied by twiddles, and Y is the transformed result.

5.1.1 ***Meta Templates.*** As presented in Section 4, butterfly calculations consist of basic complex number operations; therefore, we define these basic operation units as meta templates in Fig.2.

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

```
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
```

Figure 2: Meta templates supported in AutoFFT.

```
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[],add[],sub[],k,r):
  CPX_MLA(P,add[0],add[1],W_r^k r);
  CPX_MUL_S(Q,sub[1],W_r^k i);
  for(i=2;i<=r/2;++i)
    CPX_MLA(P,P,add[i], W_r^ik r);
    CPX_MLA(Q,Q,sub[i],W_r^ik i);
  end for
  CPX_OUT(out[k],out[r-k],P,Q);
```

```
CALC_OUT_QUAD(out[],in[],k,r):
  initial A,B,C1,D1,C2,D2 as 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)
    D=quad_D(_in,i,q-i,q/2-i,q/2+i,k)
    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);
  if(k==1)
    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);
```

Figure 3: Hybrid templates supported in AutoFFT.

- ***CPX_MLA***$(out, in1, in2, s)$ represents the fused multiply-add (FMA) operation.
- ***CPX_MUL_S***(out, in, s) is used to multiply a complex number in by a real number s.
- ***CPX_OUT***$(head, tail, A, B)$ is used to calculate a pair of results (Y_k and Y_{r-k}), as presented in Eq.7.

5.1.2 ***Hybrid Templates.*** Hybrid templates are introduced to implement the pair pattern (Eq.7) and the quad pattern (Eq.14 and Eq.15), as shown in Fig.3.

- ***CALC_LIKE_TERMS***(add, sub, in, r) unites like terms. When r is a prime number, according to Eq.7, add stores (x_0), $(x_1 +$

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: CALC_LIKE_TERMS(add, sub, tmpIn, r)
9: CALC_OUT_SPECIAL(tmpOut, add, sub, r)
10: if r is a prime number then
11:    for i←1 to r/2 do
12:       CALC_OUT_PAIR(tmpOut, add, sub, i, r)
13:    end for
14: else if r is a power of two then
15:    for i←1 to r/4 do
16:       CALC_OUT_QUAD(tmpOut, tmpIn, i, r)
17:    end for
18: end if
19: 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
```

$x_{r-1}), \ldots, (x_m + x_{r-m})$, and *sub* stores $(-x_0), (x_1 - x_{r-1}), \ldots, (x_m - x_{r-m})$. When r is a power of two, according to Eq.13, *add* stores $(x_0 + x_m), (x_1 + x_{r-1}), \ldots, (x_m + x_{r-m})$, and *sub* stores $(x_0 - x_m), (x_1 - x_{r-1}), \ldots, (x_m - x_{r-m})$; *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 a prime number; when r is a power of two, it separately calculates Y_0 and $Y_{r/2}$ using Eq.8.
- ***CALC_OUT_PAIR***(out, add, sub, k, r) implements the pair pattern (Y_k and Y_{r-k}) according to Eq.7.
- ***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.14 and Eq.15. In addition, $Y_{r/4}$ and $Y_{3r/4}$ are calculated separately. Because this process only involves basic complex number operations, we simplify the expressions for calculating Eq.13's $\hat{x}/A/B/C/D$ for readability.

5.2 The C FFT Kernel Generator

Based on the computational templates, a C FFT kernel generator is designed to autogenerate efficient FFT kernels for all radices.

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 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 8 calls *CALC_LIKE_TERMS*() to calculate the like terms that can be reused by other hybrid templates. When r is a prime number, line 9 calls *CALC_OUT_SPECIAL*() to calculate Y_0, and lines 11~13 then perform the pair pattern. When r is a power of two, line 9 calculates Y_0 and $Y_{r/2}$, and lines 15~17 then perform the quad pattern. Finally, line 19 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, the assembly template optimizer is designed to translate C FFT kernels into high-performance assembly FFT kernels for varying architectures.

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.

5.3.2 ***Instruction Mapping.*** We define instruction mapping rules to translate computational templates into hardware-specific 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.4. When SIMDizing butterflies on the ARMv8 architecture, we use 2 128-bit registers to separately hold 4 complex numbers' real and imaginary parts. On the Haswell architecture, we use one 256-bit register to hold 4 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 complete complex multiplication 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 should insert independent instructions between the memory instructions and the corresponding 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 them in parallel [29].

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

Figure 4: Instruction mapping rules between meta templates and optimization templates.

Benefiting from the optimization templates, when new architectures emerge, we only need to implement the corresponding optimization templates. As shown in Fig.4, the optimization templates use register aliases instead of physical registers. To translate the C kernels into assembly kernels, we develop register allocation strategies for different radices to implement the mapping between the register aliases and physical registers.

5.3.3 ***Register Allocation Strategies***. Because the number of vector registers in modern CPUs is limited, registers become increasingly scarce as the radix increases. Hence, we need to define register allocation strategies for different radices. We take the ARMv8 and Haswell architectures as examples. Note that ARMv8 processors contain 32 128-bit vector registers, while Intel Haswell processors contain 16 256-bit vector registers.

The main idea of register allocation optimization is to group vector registers according to their functions and to strictly define the usage rules of each group. AutoFFT divides vector registers into four groups: the input group, the twiddle group, the temporary group, and the output group. We now analyze the required registers of the four groups for the radix-r kernel according to Alg.1.

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 a prime number, the temporary variables (rr/ii/ri/ir used in $CPX\_MUL()$ and P/Q 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 a power of two, $CALC\_OUT\_QUAD()$'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 a prime number, the sum of the required registers is $3r+5$; when r is a power of two, the sum of the required registers is $3r+9$.

The register allocation strategies vary for different radices and can be divided into three cases based on Alg.1:

Small Radices. As analyzed above, 1) when r is a prime number, on the ARMv8 architecture, where the required 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 a power of two, 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 complete their tasks for radices 2 and 3.

Medium Radices. As the radix grows, the four groups require additional 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 8, 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 12 and 16, 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 a prime number, lines 8~19 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 8~19; 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, and 11 are medium radices. 2) When r is a power of two, on the ARMv8 architecture, lines 8~19 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 and 8 are medium radices.

Large Radices. When $r \geq 13$, the register resources are insufficient: we have to use the stack or memory instructions to temporarily hold relevant data, which degrades the performance. Large radices require more registers, which are limited in modern CPUs. AutoFFT provides C FFT kernels to perform the FFT computation for large radices. Specialized algorithms like Rader's algorithm [41] are better than the Cooley-Tukey algorithm for large radices.

Table 2: Experimental Environment

CPU	FT-2000+	Xeon E5-2670 v3	Xeon Silver 4110
Arch.	AArch64	Haswell	Skylake
Freq.	2.2 GHz	2.3 GHz	2.1 GHz
SIMD	128 bits	256 bits	512 bits
L1 cache	32KB	32 KB	32 KB
Compiler	gcc-4.9.3	gcc-8.1.0	gcc-5.4.0
FFTW	3.3.8	3.3.8	3.3.8
ARMPL	19.2.0	-	-
Intel MKL	-	2019 Update 1	2019 Update 1

To map specific physical registers to these register aliases, we maintain a lookup register usage table based on the three register allocation strategies to guarantee the consistency of register usage across the instruction streams. These generated FFT kernels will be called according to the butterfly network at runtime to perform FFTs of arbitrary sizes.

As analyzed above, our method essentially is not restricted to the specific width of SIMD registers but the number of SIMD registers, so it can be easily extended to the future ARM SVE and AVX-512 by replacing the number of corresponding SIMD registers. Because the FFT kernels of radices 2/3/4/5/7/8/11 (small/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 purely C FFT kernels of large radices. Regardless, sizes of the form $2^a3^b5^c7^d11^e$ meet the needs of most applications.

6 PERFORMANCE EVALUATION

This section evaluates the performance of AutoFFT on ARMv8 and Intel x86-64 architectures. 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 these libraries. 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.16 [21], which is adopted in the well-known benchmark benchFFT [20]. Note that the x-axis of figures in this section represents the transform size N.

$$GFlops = \frac{5N \cdot \log_2 N \cdot 10^{-9}}{t} \tag{16}$$

The experimental conditions are listed in Table 2. Our experiments take the C version of AutoFFT as the baseline to determine the performance boost achieved by AutoFFT's assembly kernels. Considering Intel MKL's FFTW interfaces and its own API share the same source code [30], we use the FFTW interfaces for Intel MKL in our experiments. In addition, the FFTW_MEASURE flag is used for both FFTW and Intel MKL. 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.5 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- and double-precision sequences. For single-precision floating-point sequences, when the FFT size is a power of two, as shown in Fig.5(a), AutoFFT is on average 1.53, 3.01, and 3.24 times faster than FFTW, ARMPL, and the baseline, respectively. When the FFT size is not a power of two, as shown in Fig.5(b), AutoFFT is on average 1.98, 1.77, and 2.99 times faster than FFTW, ARMPL, and the baseline, respectively. For double-precision floating-point sequences, when the FFT size is a power of two, as shown in Fig.5(c), AutoFFT is on average 1.72, 2.56, and 2.63 times faster than FFTW, ARMPL, and the baseline, respectively. When the FFT size is not a power of two, as shown in Fig.5(d), AutoFFT is on average 1.49, 1.63, and 1.76 times faster than FFTW, ARMPL, and the baseline, respectively.

From the C2C FFTs' performance curves on the ARMv8 architecture in Fig.5, 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 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.

Fig.6 shows the performances of the 1D C2C FFTs for AutoFFT, FFTW, Intel MKL, and the baseline on the Haswell architecture. For single-precision sequences, when the FFT size is a power of two, AutoFFT is on average 2.29, 1.26, and 8.6 times faster than FFTW, Intel MKL, and the baseline, respectively. When the FFT size is not a power of two, AutoFFT is on average 2.46, 1.41, and 7.72 times faster than FFTW, Intel MKL, and the baseline, respectively. For double-precision sequences, when the FFT size is a power of two, AutoFFT is on average 1.98, 1.05, and 4.23 times faster than FFTW, Intel MKL, and the baseline, respectively, and when the FFT size is not a power of two, AutoFFT is on average 1.78, 1.31, and 4.77 times faster than FFTW, Intel MKL, and the baseline, respectively.

Based on the performance curves of C2C FFTs on the Haswell architecture in Fig.6, 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 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 close to or even faster than Intel MKL. However, when the FFT size exceeds 2048, the performance curves of AutoFFT decrease faster than those of Intel MKL. There are two possible reasons for this difference in performance between AutoFFT and Intel

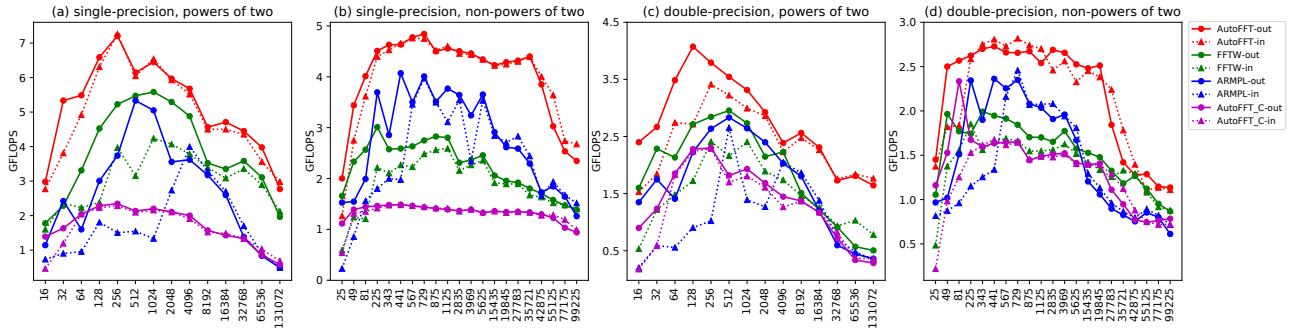

Figure 5: The 1D C2C FFT performances on ARMv8 CPUs.

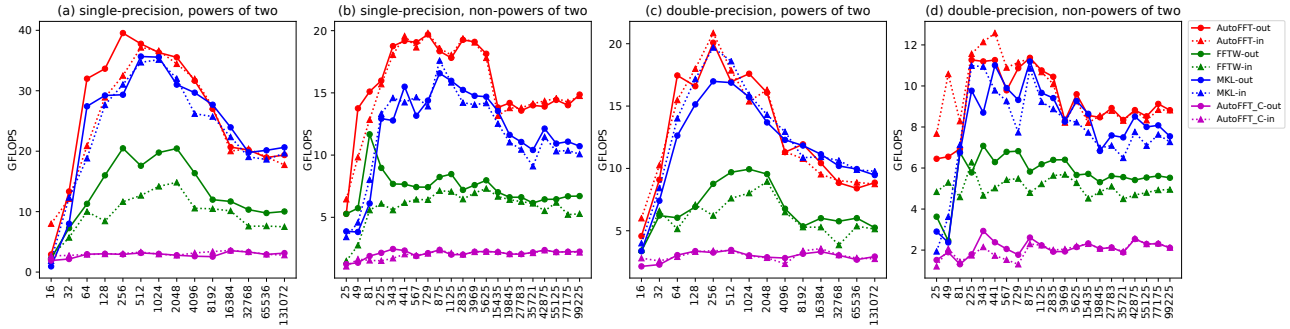

Figure 6: The 1D C2C FFT performances on Intel Haswell CPUs.

MKL. First, in addition to the Cooley-Tukey algorithm, Intel MKL may adopt other FFT algorithms, such as the split-radix [16] and the Rader-Brenner [40] 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.

To obtain a better competitive analysis, we extend AutoFFT to AVX-512 on Intel Skylake processors. Fig.7 shows the performances of the 1D C2C single-precision floating-point FFTs for AutoFFT and Intel MKL on Haswell and Skylake, and AutoFFT also outperforms Intel MKL on Skylake: when the FFT size is a power of two, AutoFFT

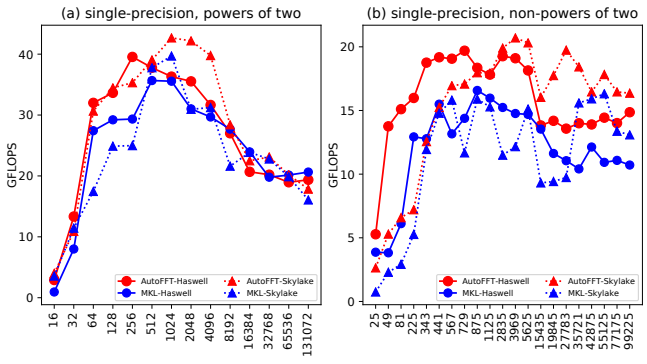

Figure 7: The 1D C2C FFT performances on Intel Haswell and Skylake CPUs.

is on average 1.2 times faster than Intel MKL; when the FFT size is not a power of two, AutoFFT is on average 1.55 times faster than Intel MKL. Intel MKL contains similar performance trends on both platforms when the FFT size is a power of two; however, when the FFT size is not a power of two, the performance trend fluctuates greater on Skylake than that on Haswell.

In many applications, the inputs or outputs are real numbers. AutoFFT currently supports the following transforms: 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 reductions (complex reduction and real reduction) to summarize and extract unified optimization patterns of real FFTs. For FFTs with real number inputs, we use complex reduction to reduce an N-point real FFT to an $N/2$-point complex FFT and then apply split operations on the transformed results. Each real FFT conducts a split operation according to its own definitions. In addition, DCT/DST contains special symmetric properties in its inputs; thus, we use real reduction to reduce an N-point real FFT to an $N/2$-point real FFT. Due to the space limitations of this paper, we present the performances of the R2C/C2R and R2R REDFT01/REDFT10 real FFTs below.

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.8 shows that the performances of AutoFFT's real

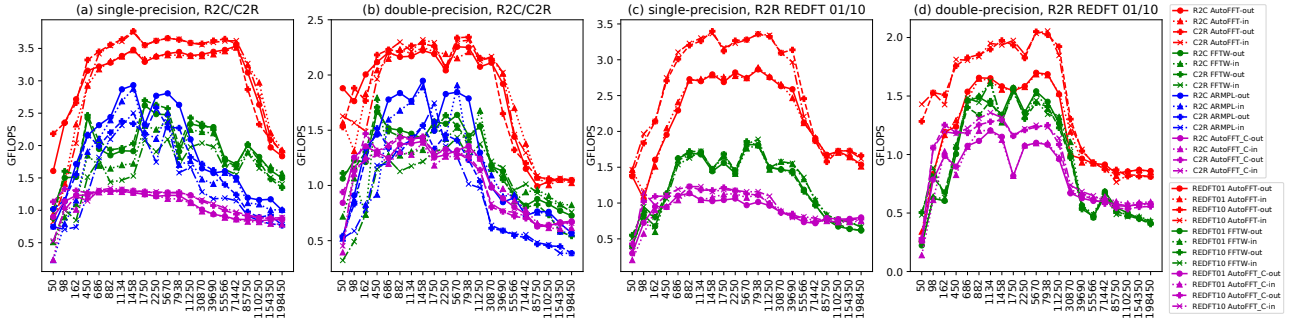

Figure 8: The 1D real FFT performances on ARMv8 CPUs.

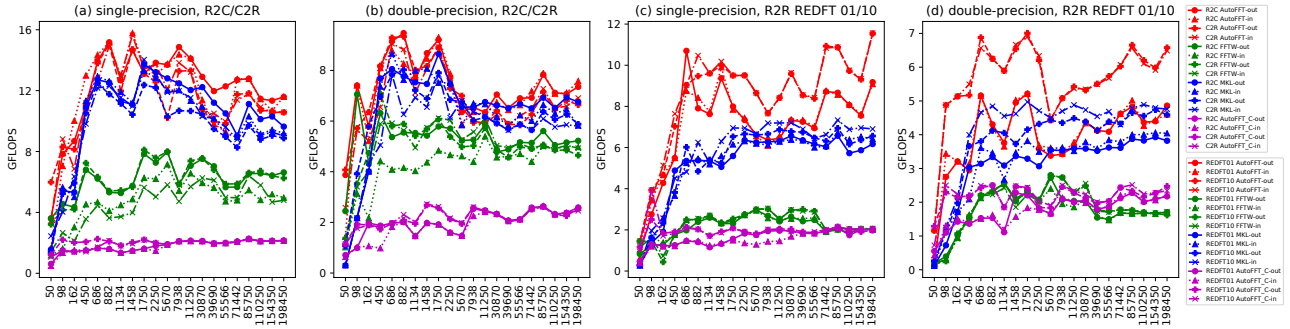

Figure 9: The 1D real FFT performances on Intel Haswell CPUs.

FFTs outperform those of FFTW, ARMPL, and the baseline on the ARMv8 architecture, especially when the FFT size is large. For R2C/C2R FFTs, when dealing with single-precision floating-point sequences, AutoFFT is on average 1.71, 2.01, and 2.82 times faster than FFTW, ARMPL, and the baseline, respectively, as shown in Fig.8(a), and when dealing with double-precision floating-point sequences, AutoFFT is on average 1.61, 1.91, and 1.77 times faster than FFTW, ARMPL, and the baseline, respectively, as shown in Fig.8(b). For R2R REDFT01/REDFT10, when dealing with single-precision floating-point sequences, AutoFFT is on average 2.04 and 2.66 times faster than FFTW and the baseline, respectively, as shown in Fig.8(c), and when dealing with double-precision floating-point sequences, AutoFFT is on average 1.51 and 1.58 times faster than FFTW and the baseline, respectively, as shown in Fig.8(d). 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 real FFTs decrease faster as the FFT size increases. Its performance is very close to or even lower than the baseline at large scales, especially for the in-place transforms.

Fig.9 shows the performances of real FFTs of AutoFFT, FFTW, Intel MKL, and the baseline on the Haswell architecture. For R2C/C2R FFTs, when dealing with single-precision floating-point sequences, AutoFFT is on average 2.24, 1.24, and 6.29 times faster than FFTW, Intel MKL, and the baseline, respectively, as shown in Fig.9(a); when dealing with double-precision floating-point sequences, AutoFFT is on average 1.47, 1.49, and 3.56 times faster than FFTW, Intel MKL, and the baseline, respectively, as shown in Fig.9(b). Therefore, FFTW is generally slower than AutoFFT and Intel MKL, and AutoFFT is faster than MKL FFT in most cases, as shown in Fig.9(a) and Fig.9(b). For R2R REDFT01/REDFT10, when dealing with single-precision floating-point sequences, AutoFFT is on average 3.59, 1.6, and 4.4 times faster than FFTW, Intel MKL, and the baseline, respectively, as shown in Fig.9(c); when dealing with double-precision floating-point sequences, AutoFFT is on average 3.26, 1.54, and 2.4 times faster than FFTW, Intel MKL, and the baseline, respectively, as shown in Fig.9(d). Therefore, AutoFFT outperforms the other two libraries, and FFTW is much slower than AutoFFT and Intel MKL, as shown in Fig.9(c) and Fig.9(d).

Table 3: The average and maximum speedups of AutoFFT.

	ARMv8			Haswell		
Speedup	FFTW	ARMPL	Baseline	FFTW	MKL	Baseline
Average	1.7	2.15	2.43	2.38	1.36	5.25
Max	2.04	3.01	3.24	3.59	1.6	8.6

Table 3 summarizes the average and maximum improvements of AutoFFT by comparing it with FFTW, ARMPL, Intel MKL, and the baseline on the ARMv8 and Intel Haswell platforms. On the ARMv8 architecture, AutoFFT is on average 1.7, 2.15, and 2.43 times faster than FFTW, ARMPL, and the baseline, respectively, across all types of FFTs. The maximum performance of AutoFFT is 2.04 times faster than that of FFTW, 3.01 times faster than that of ARMPL, and 3.24 times faster than that of the baseline. On the Haswell architecture,

AutoFFT is on average 2.38, 1.36, and 5.25 times faster than FFTW, Intel MKL, and the baseline, respectively, across all types of FFTs. The maximum performance of AutoFFT is 3.59 times faster than that of FFTW, 1.6 times faster than that of Intel MKL, and 8.6 times faster than that of the baseline.

7 CONCLUSION

This paper proposes a template-based framework named AutoFFT that makes full use of the experience of domain and optimization experts to automatically generate extremely high-performance assembly FFT codes for ARMv8 and Intel Haswell processors. AutoFFT thus substantially reduces the laborious work of developing assembly codes 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 approach to other numerical algorithms.

ACKNOWLEDGMENTS

We would like to express our gratitude to all reviewer's constructive comments for helping us polish this article. 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] Ayaz Ali and Lennart Johnsson. 2006. UHFFT: A high performance DFT framework. (2006).
[2] AMD. 2019. AOCL: AMD Optimizing CPU Libraries. https://developer.amd.com/wp-content/resources/AMDCPULibrariesUserGuide_1.0.pdf.
[3] AMD. 2019. A software library containing FFT functions written in OpenCL. https://github.com/clMathLibraries/clFFT.
[4] Apple. 2019. The Apple Accelerate libraries - vDSP. https://developer.apple.com/documentation/accelerate/vdsp/fast_fourier_transforms.
[5] ARM. 2019. ARM Ne10 project. https://github.com/projectNe10/Ne10.
[6] ARM. 2019. Arm Performance Libraries (ARMPL) 19.2.0. https://static.docs.arm.com/101004/1920/arm_performance_libraries_reference_101004_1920_00_en.pdf.
[7] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. 2006. *The landscape of parallel computing research: A view from berkeley.* Technical Report. Technical Report UCB/EECS-2006-183, EECS Department, University of ....
[8] Anthony Blake and Matt Hunter. 2014. Dynamically generating FFT code. *Journal of Signal Processing Systems* 76, 3 (2014), 275–281.
[9] Leo Bluestein. 1970. A linear filtering approach to the computation of discrete Fourier transform. *IEEE Transactions on Audio and Electroacoustics* 18, 4 (1970), 451–455.
[10] Georg Bruun. 1978. z-transform DFT filters and FFT's. *IEEE Transactions on Acoustics Speech and Signal Processing* 26, 1 (1978), 56–63.
[11] Cris Cecka. 2017. Low Communication FMM-accelerated FFT on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 54, 11 pages. https://doi.org/10.1145/3126908.3126919
[12] James W Cooley, Peter AW Lewis, and Peter D Welch. 1969. The fast Fourier transform and its applications. *IEEE Transactions on Education* 12, 1 (1969), 27–34.
[13] James W Cooley and John W Tukey. 1965. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation* 19, 90 (1965), 297–301.
[14] Pedro Costa. 2018. A FFT-based finite-difference solver for massively-parallel direct numerical simulations of turbulent flows. *Computers & Mathematics with Applications* 76, 8 (2018), 1853–1862.
[15] Yuri Dotsenko, Sara S. Baghsorkhi, Brandon Lloyd, and Naga K. Govindaraju. 2011. Auto-tuning of Fast Fourier Transform on Graphics Processors. *SIGPLAN Not.* 46, 8 (Feb. 2011), 257–266. https://doi.org/10.1145/2038037.1941589
[16] Pierre Duhamel and Henk Hollmann. 1984. Split radix'FFT algorithm. *Electronics letters* 20, 1 (1984), 14–16.
[17] Franz Franchetti, Tze Meng Low, Doru Thom Popovici, Richard M Veras, Daniele G Spampinato, Jeremy R Johnson, Markus Püschel, James C Hoe, and Jose MF Moura. 2018. SPIRAL: Extreme Performance Portability. *Proc. IEEE* 106, 11 (2018), 1935–1968.
[18] F. Franchetti, M. Puschel, Y. Voronenko, S. Chellappa, and J. M. F. Moura. 2009. Discrete fourier transform on multicore. *IEEE Signal Processing Magazine* 26, 6 (November 2009), 90–102. https://doi.org/10.1109/MSP.2009.934155
[19] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. 2005. Formal loop merging for signal transforms. *ACM SIGPLAN Notices* 40, 6 (2005), 315–326.
[20] M Frigo and SG Johnson. 2019. benchFFT. http://www.fftw.org/benchfft.
[21] M Frigo and SG Johnson. 2019. The benchmarking methodology of benchFFT. http://www.fftw.org/speed/.
[22] Matteo Frigo and Steven G. Johnson. 1997. *The Fastest Fourier Transform in the West.* Technical Report MIT-LCS-TR-728. Massachusetts Institute of Technology.
[23] M. Frigo and S. G. Johnson. 1998. FFTW: an adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*, Vol. 3. 1381–1384 vol.3. https://doi.org/10.1109/ICASSP.1998.681704
[24] Matteo Frigo and Steven G. Johnson. 2005. The Design and Implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".
[25] Amir Gholami, Judith Hill, Dhairya Malhotra, and George Biros. 2015. AccFFT: A library for distributed-memory FFT on CPU and GPU architectures. *CoRR* abs/1506.07933 (2015). arXiv:1506.07933 http://arxiv.org/abs/1506.07933
[26] Chunye Gong, Weimin Bao, and Guojian Tang. 2013. A parallel algorithm for the Riesz fractional reaction-diffusion equation with explicit finite difference method. *Fractional Calculus and Applied Analysis* 16, 3 (2013), 654–669.
[27] Chunye Gong, Weimin Bao, Guojian Tang, Bo Yang, and Jie Liu. 2014. An efficient parallel solution for Caputo fractional reaction–diffusion equation. *The Journal of Supercomputing* 68, 3 (2014), 1521–1537.
[28] IBM. 2019. ESSL: IBM Engineering and Scientific Subroutine Library. https://www.ibm.com/support/knowledgecenter/en/SSFHY8_6.1/navigation/welcome.html.
[29] Intel. 2016. 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.
[30] Intel. 2019. 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.
[31] D Kolba and TW Parks. 1977. A prime factor FFT algorithm using high-speed convolution. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 25, 4 (1977), 281–294.
[32] Yan Li, Yun-Quan Zhang, Yi-Qun Liu, Guo-Ping Long, and Hai-Peng Jia. 2013. MPFFT: An autotuning FFT library for OpenCL GPUs. *Journal of Computer Science and Technology* 28, 1 (2013), 90–105.
[33] Zhihao Li, Haipeng Jia, Yunquan Zhang, Shice Liu, Shigang Li, Xiao Wang, and Hao Zhang. 2019. Efficient parallel optimizations of a high-performance SIFT on GPUs. *J. Parallel and Distrib. Comput.* 124 (2019), 78–91.
[34] Dragan Mirković, Rishad Mahasoom, and Lennart Johnsson. 2000. An Adaptive Software Library for Fast Fourier Transforms. In *Proceedings of the 14th International Conference on Supercomputing (ICS '00)*. ACM, New York, NY, USA, 215–224. https://doi.org/10.1145/335231.335252
[35] Akira Nukada, Yutaka Maruyama, and Satoshi Matsuoka. 2012. High Performance 3-D FFT Using Multiple CUDA GPUs. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units (GPGPU-5)*. ACM, New York, NY, USA, 57–63. https://doi.org/10.1145/2159430.2159437
[36] Nvidia. 2019. CUFFT library. https://docs.nvidia.com/pdf/CUFFT_Library.pdf.
[37] Dan Petre, Adam T. Lake, and Allen Hux. 2016. OpenCL&Trade; FFT Optimizations for Intel&Reg; Processor Graphics. In *Proceedings of the 4th International Workshop on OpenCL (IWOCL '16)*. ACM, New York, NY, USA, Article 12, 4 pages. https://doi.org/10.1145/2909437.2909451
[38] D. T. Popovici, T. M. Low, and F. Franchetti. 2018. Large Bandwidth-Efficient FFTs on Multicore and Multi-socket Systems. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 379–388. https://doi.org/10.1109/IPDPS.2018.00048
[39] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* 93, 2 (2005), 232–275.
[40] C Rader and NJIToA Brenner. 1976. A new principle for fast Fourier transformation. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 24, 3 (1976), 264–266.
[41] Charles M Rader. 1968. Discrete Fourier transforms when the number of data samples is prime. *Proc. IEEE* 56, 6 (1968), 1107–1108.
[42] Thomas G. Stockham, Jr. 1966. High-speed Convolution and Correlation. In *Proceedings of the April 26-28, 1966, Spring Joint Computer Conference (AFIPS '66 (Spring))*. ACM, New York, NY, USA, 229–233. https://doi.org/10.1145/1464182.1464209

[43] Paul N. Swarztrauber. 1982. Vectorizing the ffts. In *Parallel Computations*, GARRY RODRIGUE (Ed.). Academic Press, 51 – 83. https://doi.org/10.1016/B978-0-12-592101-5.50007-5

[44] Daisuke Takahashi. 2014. FFTE: A Fast Fourier Transform Package. http://www.ffte.jp/.

[45] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 167–188.

[46] Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua. 2001. SPL: A Language and Compiler for DSP Algorithms. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*. ACM, New York, NY, USA, 298–308. https://doi.org/10.1145/378795.378860

[47] Dongxiao Zhang, Zhibin Chen, Cheng Xiao, Mengze Qin, and Hao Wu. 2019. Accurate simulation of turbulent phase screen using optimization method. *Optik* 178 (2019), 1023–1028.