Implementation and Optimization of Multi-dimensional Real FFT on ARMv8 Platform

Xiao Wang[1,2], Haipeng Jia[1(✉)], Zhihao Li[1,2], and Yunquan Zhang[1]

[1] State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
{wangxiao17s,jiahaipeng,lizhihao,zhangyunquan}@ict.ac.cn

[2] School of Computer and Control Engineering, University of Chinese Academy of Sciences, Beijing, China

Abstract. Fourier Transform is one of the most critical algorithms, and is applied in a wide range of fields like signal processing and data compression. In real world applications, such as image compression (JPEG), Fourier Transform is concentrated in processing real number input. These transforms are called real DFT (real discrete fourier transform) in this paper. Thus it is critical to optimize real DFT for specific platforms. In this paper, we implement 1D and 2D real DFT on ARMv8 platform which is the flagship architecture of ARM. Real DFT kinds implemented and optimized include R2HC, HC2R, DHT, DCTI-IV, DSTI-IV and are especially optimized when input size is $2^q 3^n 5^m$. In order to achieve high performance, optimization is carried out in following aspects: (1) Reduction of the computation complexity of real DFT. (2) Implementation of high performance 1D complex DFT algorithm to support real DFT. (3) For the 2D real DFT, we propose a cache-aware blocking approach to improve cache performance. Experimental results show that: Compared with FFTw 3.3.7, 1D-Float DFT gains 1.52x speedup in average across all real DFT kinds, maximum speedup reaches 1.79x; 1D-Double DFT gains 1.34x speedup in average across all real DFT kinds, maximum speedup reaches 1.61x; 2D-Float DFT gains 1.41x speedup in average across all real DFT kinds, maximum speedup reaches 1.70x; 2D-Double DFT gains 1.10x speedup across all real DFT kinds, maximum speedup reaches 1.25x.

Keywords: Real Fast Fourier Transform · Program optimization ARMv8

1 Introduction

Fourier transform has been applied widely across various fields including signal processing and data compression [1,2]. On one hand, input of most real world applications is of real number format, such as pixel value or super parameters

J. Vaidya and J. Li (Eds.): ICA3PP 2018, LNCS 11335, pp. 338–353, 2018.
https://doi.org/10.1007/978-3-030-05054-2_27

of neural network [5–7], On the other hand, with thriving of ARM ecosystem, ARMv8 platform is being promoted to server market, computation efficiency is becoming critical in ARM platforms. Therefore, a high performance real number discrete fourier transform library on ARMv8 platform is of paramount importance.

In this paper, we implement and optimize a high performance 1D and 2D real DFT library using Cooley-Tukey FFT algorithm on ARMv8 platform. Implemented real DFT kinds include R2HC, HC2R, DHT, DCTI-IV, DSTI-IV. $2^q 3^n 5^m$ computation size is especially optimized. In order to achieve high performance, challenges need to be coped: (1) Diversity of real DFT brings difficulties. Different definitions of real DFT bring challenges to the choice of optimization approaches. (2) Real DFT depends on complex DFT. So the first step of developing real DFT is to develop complex DFT with high performance. Although there are various algorithms have been proposed for FFT, developing a high performance FFT library on new hardware still is a challenging work. (3) Architecture exploration of ARMv8 platform. Although there are already some libraries have been developed on ARM architecture, few optimization techniques for FFT on ARMv8 platform is recorded. This work explores utilization of SIMD instructions and registers besides methods for tuning cache performance.

As a summary, our contributions are focused on addressing following challenges: (1) We first summarize and abstract real DFT optimization algorithms into an unified two reduction form. With benefits of this, optimizations on real DFT are of an unified form. Meanwhile, we reduce original real DFT to less computational intensive transforms by taking advantage of symmetry of real DFT. (2) We implement and optimize 1D Cooley-Tukey complex FFT algorithm with high performance on ARMv8 platform through re-constructing butterfly network, simplifying butterfly calculations and using SIMD assembly instructions. (3) For 2D real DFT, we propose a cache-aware algorithm for ARMv8 platform to improve cache performance. After adopting these optimization techniques, high performance is obtained.

Because there is only few libraries support ARMv8 platform, and FFTw's excellent performance across all platforms, FFTw 3.3.7 is selected as our comparison baseline. Although ARM Performance Library implements complex DFT on ARMv8 CPUs, real DFT is still not supported yet. Experimental results show that: Compared with FFTw 3.3.7, 1D-Float DFT achieves around 1.52x speedup in average across all real DFT kinds, maximum speedup reaches 1.79x; 1D-Double DFT gains speedup 1.34x in average across all real DFT kinds, maximum speedup reaches 1.61x; 2D-Float DFT achieves 1.41x speedup in average across all real DFT kinds, maximum speedup reaches 1.70x; 2D-Double achieves 1.10x speedup across all real DFT kinds, maximum speedup reaches 1.25x.

The rest of this paper is organized as following: Sect. 2 summarizes related works; Sect. 3 introduces details of optimization of real DFT; Sect. 4 presents details of implementation and optimization of 1D complex DFT; Sect. 5 introduces 2D cache-aware algorithm; Sect. 6 analyzes experimental results; summary and future work considerations are presented in Sect. 7.

2 Related Work

There are a lot researches on efficient real DFT algorithms. Two approaches mentioned in [3] are complex DFT based approaches and approach of customizing real DFT computation within every FFT stage. For the sake of diversity of real DFT and code size, a stage level customized optimization is not practical besides poor extensibility for newly added real DFT kind.

Therefore, in this paper, we implement real DFT based on complex DFT, and unify these transforms into an unified form. Transforms with random input such as R2HC (real to Half complex transform), HC2R (Half complex to real) and DHT (discrete hartley transform) are optimized as a halved complex transform; Transforms with symmetrical input such as DCT/DST I-IV are optimized case by case: For DCT/DST I, method that reduces the original transform into a real DFT with half size is mentioned in [13]. But the loss of accuracy is unacceptable. Therefore, we solve DCT/DST I directly as DHT and R2HC, HC2R cases. For DCT II and III, a concise computation approach has been proposed in [12]. Original problem is solved with a halved real DFT, meanwhile, [14] points out DST II and III is equivalent to corresponding DCTs inherently. For consideration of flexibility, method in [12,14] is adopted, as more conditions are handled compared with methods mentioned in [8]. In the end, DCT/DST IV are solved based on DCT/DST II/III with method mentioned in [15] to integrate all real DFT kinds together.

There are already several FFT computation libraries, such as FFTW [10], PFFT [4], MPFFT [11], PKUFFT [9] and ARM Performance Library [16], but only FFTw and Arm Performance Library have implemented and optimized FFT on ARMv8 platform. Further, real DFT kinds are still not supported by ARM Performance Library. Therefore, FFTw3.3.7 is chosen as our comparison baseline.

3 Optimization of Algorithm for 1D Real DFT

3.1 Introduction to DFT

Given a sequence of sampled complex number: $x_0, x_1, ..., x_{n-1}, x_n$. DFT transform this sequence into frequency domain by Eq. 1:

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{nk} \tag{1}$$

Here W_N^{nk} is also called twiddle factors which is defined as $e^{\frac{-2nkj\pi}{N}}$ essentially, DFT can be expressed as a matrix multiplication between input vector and a pre-defined DFT matrix, take five points for example:

$$X = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & W_N^1 & W_N^2 & W_N^3 & W_N^4 \\ 1 & W_N^2 & W_N^4 & W_N^6 & W_N^8 \\ 1 & W_N^3 & W_N^6 & W_N^9 & W_N^{12} \\ 1 & W_N^4 & W_N^8 & W_N^{12} & W_N^{16} \end{bmatrix} \times x, \tag{2}$$

real DFT is a special DFT with input sequence is real number. The rest of this section classify these real DFT kinds based on property of their input vector and clarify proposed two reduction approaches: (1) Reduction from real DFT to halved complex DFT; (2) Reduction from real DFT to halved real DFT. Reductions above are called real reduction and complex reduction in Table 1 respectively. Two reduction approaches are both used for decreasement of computation of real DFT. For clarity, Table 1 shows each implemented real DFT's adopted reduction approach. As Table 1 shows, complex reduction is adopted by all real DFT kinds. In fact, complex reduction is adopted to reduce the output of real DFT if real reduction is adopted. Rest of this section presents details of implementation of two reduction approaches.

Table 1. Relation between all real DFT kinds and reduction approach

Real DFT kind	R2HC	HC2R	DHT	DCT I	DCT II	DCT III	DCT IV	DST I	DST II	DST III	DST IV
Real reduction	No	No	No	No	Yes	Yes	Yes	No	Yes	Yes	Yes
Complex reduction	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

3.2 Reduction from Real DFT to Halved Complex DFT

To solve a transform with input is pure real number sequence, a naive method is to regard real DFT as complex DFT with each input elements' imaginary part is zero. However, it brings unnecessary calculations and extra storage space. Therefore, this paper reduce real DFT into complex transform with half size and split result from complex transform's output. Given Eq. 1, its right part can be splited as sum of F_r, G_r:

$$F_r = \sum_{l=0}^{\frac{N}{2}-1} f_l W_{\frac{N}{2}}^{rl} \quad G_r = \sum_{l=0}^{\frac{N}{2}-1} g_l W_{\frac{N}{2}}^{rl} \tag{3}$$

Basic motivation of following steps is to extract F_r and G_r from result of a complex transform of only half size.

As $f_l = x_{2l}$, $g_l = x_{2l+1}$, we regard the adjacent two number f_l and g_l as a complex number $f_l + g_l j$. Based on Eq. (1), we achieve a transform of only half size:

$$Y_r = \sum_{l=0}^{\frac{N}{2}-1} (f_l + jg_l) W_{\frac{N}{2}}^{rl} = F_r + jG_r \tag{4}$$

Next step, we split F_r and G_r from Y_r based on Eq. 5:

$$F_r = \frac{1}{2}(Y_r + \overline{Y}_{\frac{N}{2}-r}) \quad G_r = \frac{j}{2}(\overline{Y}_{\frac{N}{2}-r} - Y_r) \tag{5}$$

Therefore, we reduce a real DFT into a halved complex transform. The distinct part of different real DFT kinds is relied on the way of organizing imaginary parts and real parts. Take DHT (Discrete Hartley Transform) for example:

$$X_k = \sum_{n=0}^{N-1} x_n[cos(\frac{2\pi njk}{N}) + sin(\frac{2\pi njk}{N})] \tag{6}$$

Result is achieved directly as sum of imaginary part and real part once we retrieve X_k's imaginary part and real part from F_r and G_r. As a summary, general reduction algorithm is given as Algorithm 1. Computation steps of line 1–8 is the common part. After we retrieve real and imaginary parts, re-construction steps are carried out according to specified transform's definition.

Algorithm 1. ComplexReduction(Input x, Output X, Direction dir, kind k)

```
1: Complex DFT(x,Y, Direction)
2: Compute X[N/2], X[0];
3: for each $i \in [1, N/4]$ do
4:     $Y_r \leftarrow Y[i]$
5:     $Y_{nr} \leftarrow \overline{Y[\frac{N}{2}]}$
6:     $Fr \leftarrow Y_r + Y_{nr}$
7:     $gr \leftarrow j * (Y_{nr} - Y_r)$
8:     $Gr \leftarrow gr * W_N^r$
9:     Retrive real part and imginary part from $Fr$ $Gr$
10:    $X[r] \leftarrow$ Reconstruct real part and imaginary part based on real DFT type(k).
11: end for
12: return;
```

3.3 Reduction from Real DFT to Halved Real DFT

Different from transforms above, DCT/DST possess special symmetry property within input. So generally input vectors of these transforms are often given only half of input. Based on choice of symmetry position, we implement the four most common kinds of DCT and DST respectively.

In this part, we give definition of these transforms and introduce specific considerations brought by symmetry: DCT/DST I are solved by Algorithm 1 due to accuracy consideration mentioned in [10]. Thus there is no extra introduction. DCT/DST III is reduced to another real DFT with half size. DCT/DST IV are divided into sub-transform of DCT/DST III [15]. Given DCT II/III:

$$DCTII : X_k = x_0 + (-1)^k x_{N-1} + 2\sum_{n=1}^{N-2} x_n cos(\frac{\pi nk}{N-1}) \tag{7}$$

$$DCTIII : X_k = x_0 + (-1)^k x_{N-1} + 2\sum_{n=1}^{N-2} x_n cos(\frac{\pi nk}{N-1}) \tag{8}$$

Based on Fast DCT algorithm from [12], DCT II is re-expressed as Eq. 9:

$$X_k = 2RealPart[W_{2N}^{k} \sum_{n=0}^{N/2-1} v_n W_{N/2}^{nk})] \tag{9}$$

$$v_k = \frac{2}{N} \sum_{n=0}^{\frac{N}{2}-1} V_n W_{N/2}^{-nk} \tag{10}$$

$$DCTIV : X_k = 2 \sum_{n=0}^{N-1} x_n cos(\frac{\pi(n+1/2)(k+1/2)}{N}) \tag{11}$$

For DCT II, based on Eq. 9, original transform is reduced into a real DFT of v_k with N/2 size. v_n is constructed by interleaving even indexed and odd indexed elements from x_n. For DCT III, based on Eq. 10, original transform is reduced

Algorithm 2. DCT/DST(Input x, Output X, Direction dir, Kind k)

```
1: N here is input size of DCT/DST, which is near half of logical transform size.
2: if k equals DSTIV/DCTIVs then
3:     $dct - input/dst - input \leftarrow x$.
4:     Algorithm 2(dct-input, $X_1$, backward, dctIII);
5:     Algorithm 2(dst-input, $X_2$, backward, dstIII);
6:     for each $i \in [0, N/2]$ do
7:         $X[i] \leftarrow X_1[i] * sptws[i].r + X_2[i] * sptw[i].i$
8:         $X[i + N/2] \leftarrow X_1[N/2-1-i] * sptw[N/2-1-i].i + X_2[N/2-1-i] * sptw[N/2-1-i].r$
9:     end for
10:    return;
11: end if
12: if k equals DST II then
13:     $xm[i] \leftarrow x[i] * (-1)^{i \ mod \ 2} \ i \ from \ 0 \ to \ N$
14:     Algorithm2 (xm, $X$, forward, dctII);
15: end if
16: if k equals DST III then
17:     $xm[i] \leftarrow x[N-1-i]$
18:     Algorithm2 (xm, $X$, forward, dctIII);
19: end if
20: if k equals DCT II then
21:     for each $i \in [0, N/2-1]$ do
22:         $xm_i \leftarrow x[2i]$
23:         $xm_{N-i} \leftarrow x[2i+1]$
24:     end for
25: end if
26: if k equals DCT III then
27:     $xm[i] \leftarrow x[N-1-i]$
28: end if
29: Algorithm1 (xm, X, dir, kind);
```

into a real DFT of V_n. V_n is defined as: $V_k = \frac{1}{2}W_{2N}^{-k}[X_k - jX_{N-k}]$. Then we need to split x_k from v_k. To avoid repeated memory visit cost, split operation for v_k is finished by re-mapping result and output vector's index. For DST II/III, they can be derived from DCT II/III by flopping sign of input vector or reversing sequence order. DCT IV is given as Eq. 11. Based on matrix factorization method, we can factorize a DCT IV transform into a DCT III and a DST III sub-transforms with N/2 input size, meanwhile, DST IV is solved based on DCT IV through flipping operations. As a summary, DCT II/III is the core transforms as other transforms are derived from DCT II/III with split operations. Algorithm 2 gives the summary of integration of DCT I-IV. In description of Algorithm 2, for clarity, we pack operations needed for v_n and V_n into Algorithm 1.

4 Implementation and Optimization of 1D Complex DFT

From discussion above, although we have cut off much redundant computations by transform reduction, final computational steps are still relied on 1D complex DFT. Therefore, in this paper, we implement and optimize a high performance 1D complex DFT library using cooley-Tukey FFT algorithm to support real transforms. Optimization considerations include designing a SIMD friendly FFT butterfly network without data copying, reducing computational complexity of butterfly, optimizing butterfly with SIMD techniques.

4.1 Butterfly Network Optimization

In solving FFT, Cooley-Tukey algorithm is one of the most famous FFT algorithms, In this algorithm, DFT is solved stage by stage, with butterfly kernel computation processed repeatedly in each stage. So the way butterfly network is organized affects optimization as a whole. Generally, there are two approaches in implementing this algorithm: (1) Decimation-in-time, DIT (2) Decimation-in-frequency, DIF. Take network of DIT of eight points and radix is 2 for example in Fig. 1.

When using DIT, input vector is of bit-reversed order, output vector is of nature order. For DIF, this condition is reversed. However, bit-reversed order not only brings extra memory cost, but also increases difficulties for blending mix

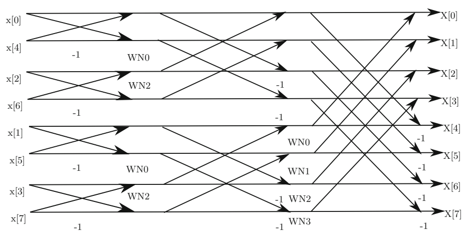

Fig. 1. DIT radix-2 butterfly network with 8 points

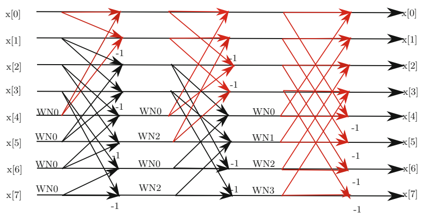

Fig. 2. Unified network with 8 points. (Color figure online)

radixes into an unified framework. Thus, this paper adopts an unified butterfly network structure shown in Fig. 2.

This network has three advantages compared with the network described above: (1) No need to be bit-reversed. Both DIT and DIF need bit-reverse operation to calibrate input or output elements' order. Thus this extra memory accessing cost is saved by our network structure. (2) Simd-friendly. To efficiently wield SIMD, data to be loaded from and stored into memory should be consecutive. Within structure of this network, input and output of consecutive butterflies is located consecutively. Besides that, section in every stage is an independent computational unit. It is convenient to arrange our SIMD parallelization within the same section. In Fig. 2, one red color part represents one section in each stage. (3) Mix-radix friendly, different radix algorithm is solved coherently in an unified approach. Because order of input and output of every stage is of natural order, it is convenient to concatenate stages of different radixes together. Computation is processed stage by stage naturally. To gain a better performance, we split the first stage out of the general computation network. There are two reasons: (1) As twiddles used in first stage is constant 1, it is unnecessary to read twiddles from memory and compute with them. So, this method reduces unnecessary memory access and computation cost. (2) The layout of first stage output is different from other stags, Thus zip instructions must be applied to rearrange result. Separating the first stage from other stags is convenient for us to do special SIMD optimization for this stage without affecting other stags.

4.2 Bufferfly Computation Optimization

In computational process, butterfly computation is invoked repeatedly, naturally, attention needs to be paid on this process to gain better performance. This paper takes radix-5 computation for example to illustrate the way of optimizing a kernel computation. This method can be generalized to other radix. given x_0, x_1, x_2, x_3, x_4 as kernel input, X_0, X_1, X_2, X_3, X_4 as kernel output. The original computation steps are given as:

$$X_0 = x_0 + x_1 + x_2 + x_3 + x_4$$

$$X_1 = x_0 + W_5^1 x_1 + W^2 x_2 + W^{-2} x_3 + W_5^{-1} x_4$$

$$X_2 = x_0 + W_5^2 x_1 + W_5^{-1} x_2 + W_5^1 x_3 + W_5^{-2} x_4$$

$$X_3 = x_0 + W_5^{-2}x_1 + W_5^1x_2 + W_5^{-1}x_3 + W_5^2x_4$$
$$X_4 = x_0 + W_5^{-1}x_1 + W_5^{-2}x_2 + W_5^2x_3 + W_5^1x_4$$

Because W_N^k and W_N^{-k} is symmetrical with the x-axis. we can merge the same terms:

$$X_0 = x_0 + (x_1 + x_4) + (x_2 + x_3)$$
$$X_1 = x_0 + (A - B)\ X_2 = x_0 + (C + D)$$
$$X_3 = x_0 + (C - D)\ X_4 = x_0 + (A + B)$$
$$A = (x_1 + x_4) * W_5^1.r + (x_2 + x3) * W_5^2.r$$
$$B = [(x_1 - x_4) * W_5^1.i + (x_2 - x3) * W_5^2.i] * (-j)$$
$$C = (x_1 + x_4) * W_5^2.r + (x_2 + x3) * W_5^1.r$$
$$D = [(x_1 - x_4) * W_5^2.i - (x_2 - x3) * W_5^1.i] * j$$

the repeated term here are: $x_1 + x_4,\ x_1 - x_4,\ x_2 - x_3,\ x_2 + x_3\ and\ A,\ B,\ C,\ D.$ through combination of same term, extra float computation can be saved compared with direct computation. This method can be extended to radix-3, radix-7 and other radix cases. So we can implement various radixes with the most streamlined computational complexity.

4.3 Butterfly SIMD Optimization

ARMv8 is the most up-to-date architecture of ARM cooperation. Both 32-bit execution status and 64-bit status are supported. Besides 31 64-bit general purpose registers (X0-X30), this architecture also provides 32 128-bit vector/scalar registers (V0-V31/Q0-Q31). These vector registers could store 4 floats or 2 doubles number, therefore, 4 float or 2 double operations are finished in parallel.

(1) Inter-Butterfly Parallelization: With benefits from the structure of butterfly-network described above, input and output data of continuous butterflies are arranged consecutively, with good locality. Therefore, it's very suitable to use SIMD technology to process multiple butterflies in the same time. For clarity, we take radix-5 for example.

In Fig. 3, four colors stand for four different butterflies, thus, input of four butterflies can be loaded into a 128-bit vector register and processed in the same time. For example, input from x_0[0] to x_4[0] is loaded into a vector register which is the first input across four butterflies.

(2) Assembly Instruction Selection: We improve performance of core computational part by using assembly instructions. Through tuning the execution order of instructions, we avoid pipeline bubbles. Through optimizing the usage of vector register, on-chip memory is efficiently used. Meanwhile, we also adopted extra optimizations to improve performance further: (1) Through zip1 instruction to rearrange output elements' order within the first stage. (2) Use ld2, faddq, st2 and other instructions to efficiently do complex number arithmetic operation. (3) Apply fmla/fmls properly to gain better computational performance.

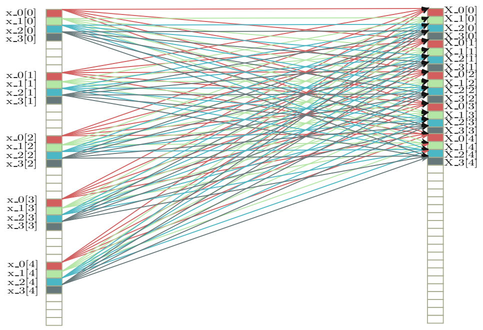

Fig. 3. Parallelization of 4 butterfly computations when radix is 5 (Color figure online)

(3) Reuse of Vector Register: To cope with shortage of vector register when solving very large radix computation, register reuse is applied. Through defining register using table and reusing rules, vector register is reused efficiently: Registers are enough when radix is 3, 4, 5, so we split register into four groups: input register groups input; output register group; intermediate result register group; twiddles register group. When radix is so large that it is necessary to reuse register. We define register groups' reuse attribution, combined with computational condition to decide if to reuse register group.

(4) Optimization for Small Scale: When input scale is small enough (3, 5, 7, etc.), special implementations and optimizations are taken to achieve high performance. Optimization techniques include: (1) Twiddle factors are precomputed and prepared in micro format to save relevant computations and memory accessing costs. (2) Loop-unrolling is properly applied. (3) Manage to pack as many as possible computations into the same function to avoid function invocation cost.

5 Implement and Optimization of 2D Real DFT

2D DFT is essentially based on 1D real DFT. The critical issue of 2D transform is non-consecutive memory visit when transform column data. Thus the essential part of 2D optimization is to improve cache performance.

5.1 Consideration of 2D Real DFT Optimization

Based on this consideration, we propose following techniques to improve cache performance:

(1) Cache block method: To avoid non-consecutive memory visit in visiting column data, we need to transpose output every time we have finished row computation. However, this behavior has poor cache performance. To decrease cache miss rate, in plan stage, an extra buffer is prepared whose size is fit with L2 cache size. Every time before dense computation is carried out, data is read

into this buffer. This behavior assures data using in computation step is on the cache.

(2) Memory alignment: Although cache performance is improved through cache blocking, due to the uncertainty of computation size. It is quit possible that input matrix is not aligned which bates the efficiency of cache line usage. To handle this issue, we take ARM cache line size into consideration and align row of buffer with 64 bytes. With benefits of this, cache miss rate is reduced further.

5.2 Procedure of 2D Real DFT Optimization

Figure 4 shows procedures of cache blocking approach: Scan input matrix with an aligned row buffer which is fit with L2 cache size. Every time we fill row buffer with block of data, dense computation is carried out with output is transposed into another buffered matrix.

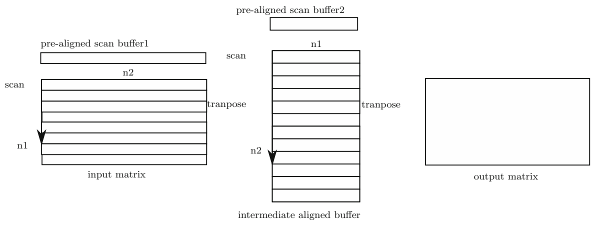

Fig. 4. General procedure of 2D transform

6 Experimental Results and Analysis

6.1 Test Platform and Comparison Baseline

Our experiment is carried out with (1) Hardware: CPU in this paper is ARM Cortex A57, 2.1 GHZ. (2) Software: Operation system is Ubuntu 15.04 with main memory size is 64 GB. FFTw 3.3.7 is chosen as comparison baseline. (3) Performance metric: Gflops = Floats Operations/Wall time. Float Operations are calculated as sum of logarithmic each dimension size multiplicated by whole computation. It is defined as $Float\ Operations = (\prod_{i=1}^{max-dim} Ni)*(\sum_{j=1}^{max-dim} logNj)$.

6.2 Experimental Results and Evaluations

In these figures, cold-coloured full line is used to stand for our transforms called OpenFFT, light-coloured dotted line is used to stand for FFTw 3.3.7's transforms. Figures 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 and 21 show our experimental results compared with FFTw3.3.7. Figures 5, 6, 7 and 8 show that our 1D float transforms outperform FFTw3.3.7 significantly with even greater advantage when input size is becoming larger. Speedup is from 1.22x to

1.79x across all real DFT kinds. Figures 9, 10, 11 and 12 show that our 1D double transforms also outperform FFTw3.3.7 a lot, except for DCT/DST IV when input size is small, the cause will be analyzed in rest of this section. Speedup is from 1.04x to 1.61x across all real DFT kinds.

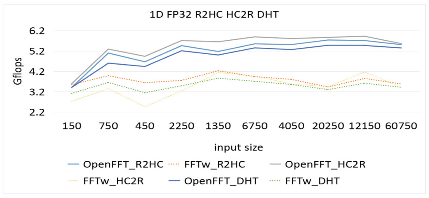

Fig. 5. 1DFP32 R2HC/HC2R/DHT (Color figure online)

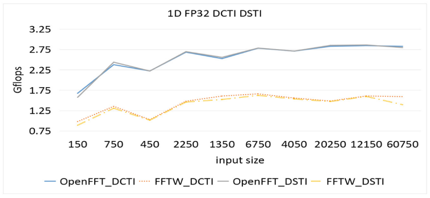

Fig. 6. 1DFP32 DCT/DST I (Color figure online)

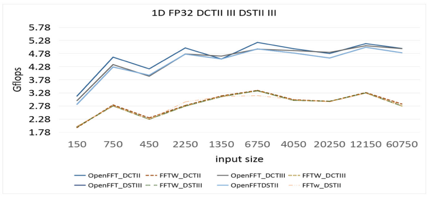

Fig. 7. 1DFP32 DCT/DST II/III (Color figure online)

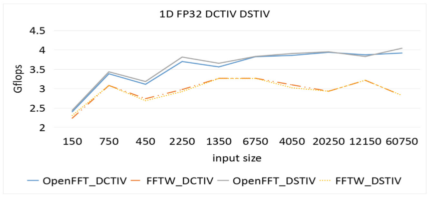

Fig. 8. 1DFP32 DCT/DST IV (Color figure online)

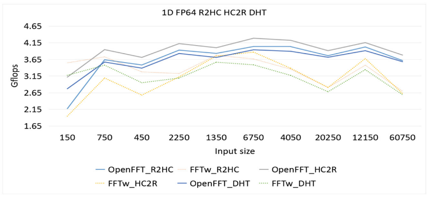

Fig. 9. 1DFP64 R2HC/HC2R/DHT (Color figure online)

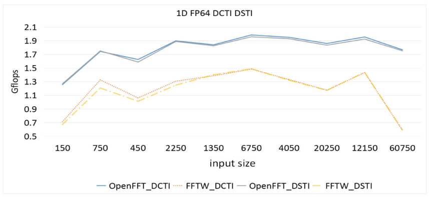

Fig. 10. 1DFP64 DCT/DST I (Color figure online)

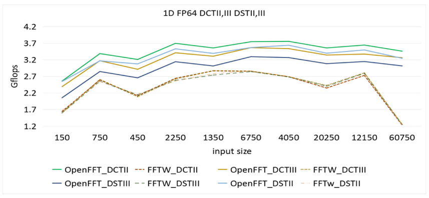

Fig. 11. 1DFP64 DCT/DST II/III (Color figure online)

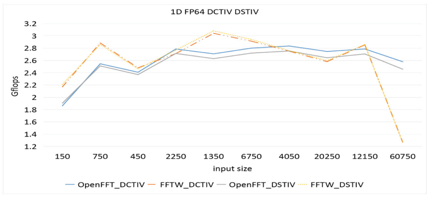

Fig. 12. 1DFP64 DCT/DST IV (Color figure online)

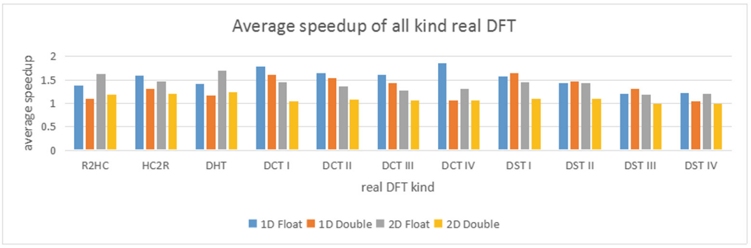

Fig. 13. Speedup across all transform kinds and types (Color figure online)

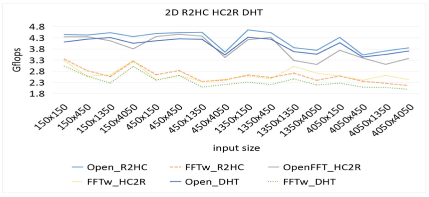

Fig. 14. 2DFP32 R2HC/HC2R/DHT (Color figure online)

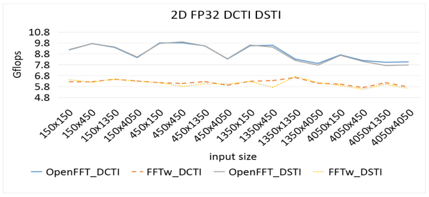

Fig. 15. 2DFP32 DCT/DST I (Color figure online)

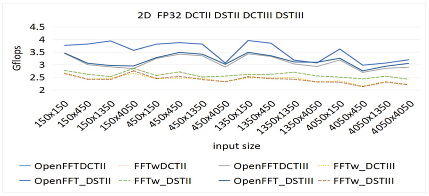

Fig. 16. 2DFP32 DCT/DST II/III (Color figure online)

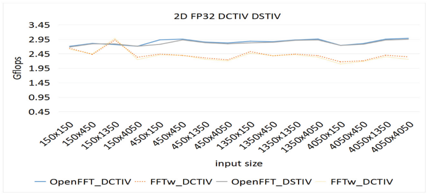

Fig. 17. 2DFP32 DCT/DST IV (Color figure online)

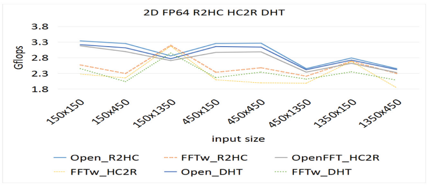

Fig. 18. 2DFP64 R2HC/HC2R/DHT (Color figure online)

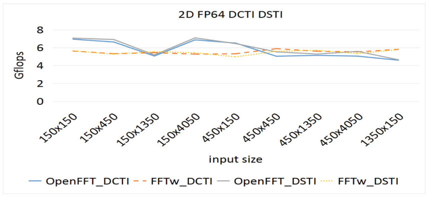

Fig. 19. 2DFP64 DCT/DST I (Color figure online)

Due to alignment and double data type, memory size of our test machine is limited for 2D transforms. Thus the test size of 2D transforms is tuned smaller. Figures 14, 15, 16 and 17 show our 2D float transforms outperform FFTw3.3.7

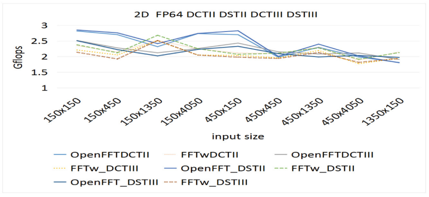

Fig. 20. 2DFP64 DCT/DST II/III (Color figure online)

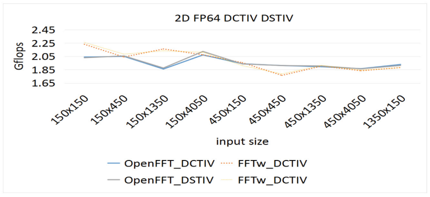

Fig. 21. 2DFP64 DCT/DST IV (Color figure online)

a lot, and speedup is from 1.20x to 1.70x. The abnormal peak point in Fig. 17 is analyzed in the rest of this section. Figures 18, 19, 20 and 21 show 2D double transforms' speedup across all kind is from 0.99x to 1.25x. Performance degeneration in DCT/DST I/IV 2D is analyzed in next section.

6.3 Performance Analysis

This section summarizes and analyzes performance comparison between ours and FFTw3.3.7's: As a whole, our transforms outperform FFTw3.37's a lot except for some transform cases. Moreover, speedup of double data type is not as significant as float as shown in Fig. 13. Causes are presented:

(1) 1D Double DCT/DST IV: To obtain transform result, operations applied to combine two sub-transform's results are not optimized well on double data type. costs brought by these operations are significant when input size is relatively small.

(2) General Analysis of Performance Degeneration Between 2D and 1D Transforms: Performance degeneration is caused by pre-process of input. For example sign flip operations of DCT/DST II/III and extension operations of DCT/DST I. And the degree of it is related to the complexity of these operations. Further overheads are accumulated with increased transform dimensions.

(3) Analysis of Performance of 2D DCT/DST I/IV: As DCT/DST I are solved based on Algorithm 1, Extension operations of intermediate result of row transforms are inevitable before solving column transforms. Cache misses brought by data copying of extension operation bring non-ignorable overhead. Except for general pre-process of input, DCT/DST IV need extra combining operations to achieve final results, which are not optimized enough for multi-dimensional condition.

(4) Abnormal Performance Peak Point of Fig. 17: Order of radix derived from prime factorization affects performance as a whole. As our implementation does not support optimization on radix generation, the radix order is not optimal when size is 1350.

(5) Influence of Double Data Type: Type of data has influence on performance as a whole. For the limitation of vector length, the parallelized double operations are inherently limited than float. Therefore more cares of optimization need to be taken into double cases, and effect of optimization is inevitably abated.

7 Conclusion and Future Work

In this paper, we implement and optimize 11 1D/2D real DFT kinds on ARMv8 platform. Experiments show that we outperform FFTw3.3.7 in most cases. In the future, we plan to further optimize double DCT/DST IV, moreover, a strategy to optimize radix order needs to be designed. In the end, optimization for 2^n input size needs to be researched in the future.

Acknowledgments. This work is supported by the National Key Research and Development Program of China under Grant No.2017YFB0202105 and No.2016YFE0100300; The National Natural Science Foundation of China under Grant No.61432018, No.61521092 and No.61502405; Key Technology Research and Development Programs of Guangdong Province under Grant No.2015B010108006.

References

1. Oran Brigham, E.: The Fast Fourier Transform and Its Applications, vol. 1. Prentice Hall, Englewood Cliffs (1988)
2. Reddy, B.S., Chatterji, B.N.: An FFT-based technique for translation, rotation, and scale-invariant image registration. IEEE Trans. Image Process. **5**(8), 1266–1271 (1996)
3. Sorensen, H.V., Jones, D., Heideman, M., Burrus, C.: Real-valued Fast Fourier Transform algorithms. IEEE Trans. Acoust. Speech Signal Process. **35**(6), 849–863 (1987)
4. Pippig, M.: PFFT: an extension of FFTW to massively parallel architectures. SIAM J. Sci. Comput. **35**(3), C213–C236 (2013)
5. Abtahi, T., Kulkarni, A., Mohsenin, T.: Accelerating convolutional neural network with FFT on tiny cores. In: IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1–4. IEEE (2017)
6. Cecotti, H., Graeser, A.: Convolutional neural network with embedded Fourier Transform for EEG classification. In: 19th International Conference on Pattern Recognition, ICPR 2008, pp. 1–4. IEEE (2008)
7. Lavin, A., Gray, S.: Fast algorithms for convolutional neural networks, pp. 4013–4021 (2016)
8. Lee, B.: FCT-a fact cosine transform. In: IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 1984, vol. 9, pp. 477–480. IEEE (1984)
9. Chen, Y., Cui, X., Mei, H.: Large-scale FFT on GPU clusters, pp. 315–324 (2010)
10. Frigo, M., Johnson, S.G.: FFTW: an adaptive software architecture for the FFT, vol. 3, pp. 1381–1384. IEEE (1998)
11. Li, Y., Zhang, Y.-Q., Liu, Y.-Q., Long, G.-P., Jia, H.-P.: MPFFT: an auto-tuning FFT library for OpenCL GPUs. J. Comput. Sci. Technol. **28**(1), 90–105 (2013)
12. Makhoul, J.: A fast cosine transform in one and two dimensions. IEEE Trans. Acoust. Speech Signal Process. **28**(1), 27–34 (1980)
13. Press, W.H.: Numerical Recipes: The Art of Scientific Computing, 3rd edn. Cambridge University Press, New York (2007)

14. Shao, X., Johnson, S.G.: Type-II/III DCT/DST algorithms with reduced number of arithmetic operations. Signal Process. **88**(6), 1553–1564 (2008)
15. Wang, Z.: On computing the discrete fourier and cosine transforms. IEEE Trans. Acoust. Speech Signal Process. **33**(5), 1341–1344 (1985)
16. ARM Performance Library. https://developer.arm.com/products/software-development-tools/hpc/arm-performance-libraries