Journal of Intelligent & Fuzzy Systems 31 (2016) 957–965
DOI:10.3233/JIFS-169025
IOS Press

A performance-efficient and datapath-regular implementation of modified split-radix fast Fourier transform

Weihua Zheng[a,b], Shenping Xiao[a], Kenli Li[b], Keqin Li[b,c] and Weijin Jiang[d,*]
[a]*College of Electrical and Information Engineering, Hunan University of Technology, Zhouzhou, China*
[b]*College of Information Science and Engineering, Hunan University, Changsha, China*
[c]*Department of Computer Science, State University of New York, New Paltz, New York, USA*
[d]*School of Computer and Information Engineering, Hunan University of Commerce, Changsha, China*

Abstract. Discrete Fourier transform (DFT) finds various applications in signal processing, image processing, artificial intelligent, and fuzzy logic etc. DFT is often computed efficiently with Fast Fourier transform (FFT). The modified split radix FFT (MSRFFT) algorithm implements a length-N=2^m DFT achieving a reduction of arithmetic complexity compared to split-radix FFT (SRFFT). In this paper, a simplified algorithm is proposed for the MSRFFT algorithm, reducing the number of real coefficients evaluated from $5/8N - 2$ to $15/32N - 2$ and the number of groups of decomposition from 4 to 3. A implementation approach is also presented. The approach makes data-path of the MSRFFT regular similar to that of the radix-2 FFT algorithm. The experimental results show that (1) MSRFFT consumes less time on central processing units (CPUs) with sufficient cache than existing algorithms; (2) the proposed implementation method can save execution time on CPUs and general processing units (GPUs).

Keywords: Fast Fourier transform (FFT), general processing unit (GPU) parallelism, modified split-radix (MSR), split-radix (SR)

1. Introduction

Discrete Fourier transform (DFT) is one of the most important tools in signal processing, image processing, artificial intelligent, fuzzy logic, intelligent systems, and web-based applications etc [23, 29, 30]. The DFT is generally implemented with a class of algorithms called fast Fourier transform (FFT). Since the introduction of Cooley and Tukey's algorithm in 1965 [5], considerable research has been conducted resulting in a large number of algorithms for length-2^m DFTs [1, 6, 14, 26]. One of these algorithms, split-radix FFT (SRFFT), achieved a reduction of order of magnitude of computational complexity over Cooley and Tukey's algorithm. The algorithm was presented by Yavne in 1968 [31] and subsequently rediscovered simultaneously by various authors [6, 19, 28] in 1984. Grigoryan and Agaian [11] presented another algorithm, which reduces the number of arithmetic operations for DFTs of lengths greater than 256 at the expense of more complicated structure compared with SRFFT; however, it is computationally less efficient for smaller lengths. The record of computational complexity was broken by an invariant of SRFFT, i.e., the modified SRFFT (MSRFFT) algorithm, proposed by Johnson

*Corresponding author. Weijin Jiang, School of Computer and information engineering, Hunan University of Commerce, Changsha 410205, China. E-mail: xsph_519@163.com.

and Frigo in 2007 [13]. Subsequently, the record was broken by two algorithms using scaled radix-2/8 FFT (SR28FFT) in 2014 [33]. Other two algorithms using SR28FFT in [33] can compute a power-of-two DFT with less arithmetic operations, higher computation accuracy, and less accesses to the lookup table of twiddle factors over SRFFT.

Today FFT [5] plays an important role in numerous applications. In particular FFT requires $O(N)$ memory access versus only $O(N \log N)$ floating point operations, requiring not only high computation throughput but also high memory bandwidth. Moreover, FFT requires extensive stride memory access, so simple mapping to stream programming could result in significant loss in performance. However, for many years, the time to perform a DFT is dominated by real number arithmetic, and considerable effort was devoted to achieving and proving the lower bounds of the operation count (real additions and multiplications), i.e., "flops" (floating-point operations) [8]. On the recent computer hardware, performance of a fast Fourier transform (FFT) algorithm is determined by many factors rather than sole arithmetic operations [7, 9]. Optimizing the number of arithmetic operations has been the focus of extensive researches, but memory management is of comparable importance on modern processors [3]. First, the implementation of FFTs involves extensive memory accesses to inputs and twiddle factors. Second, memory access is expensive due to long latency and high power consumption. Finally, memory size dominates chip area of FFT processors.

Besides central processing units (CPUs), general processing units (GPUs) are increasingly starting to be used as commodity accelerators. The computation cores of GPUs are optimized for repeating simple graphical operations, resulting in sporting much higher memory bandwidth as well as floating-point performance. Although their current power consumption as a unit is fairly high (often 100 Watts or more), because of their massive compute power their flops per-watt figure is much lower than that of conventional CPUs (In reality, the works of CPUs contain many other works besides computations).

The difficulty, of course, is that GPUs are much less general purpose than conventional CPUs, so their applicability in a wide-ranging set of high performance computing (HPC) applications must be carefully studied, especially to identify their pros and cons as well as devising effective algorithms, programming models, and programming techniques & methodologies, so as to attain optimal performance, thereby eliminating the "hype" factor and becoming truly General-Purpose Graphics Processing Units (GPGPUs). However, usages of GPUs for scientific computing have been mostly dominated by those with needs for a large number of tightly-coupled floating-point operations such as N-body problem [21, 25], or those based on kernel matrix multiplication [16] accelerations. As the nature of these applications is largely synonymous to graphics processing, i.e., abundance of independent parallelism, the ratio of floating point operations to memory access being large, as well as memory being accessed in a successive, "stream" fashion, they (obviously) can be programmed and accelerated fairly easily, using conventional shader graphics languages such as NVIDIA Cg [18] or Microsoft High Level Shader Language (HLSL), and using stream programming abstractions. BrookGPU [4] and Microsoft Accelerator [27] are extensions of C, and further develop these concepts by hiding away the complexities of underlying shader programming and allowing the programmer to focusing on stream programming. However, when the kernel algorithm(s) for an application requires descriptions beyond stream programming, GPU applicability is not well investigated.

To improve MSRFFT in practicality and computational complexity, we carry out a research. This work and Ref. [33] are the two parts of the research. The purpose of this paper is to enhance the practicality of the MSRFFT algorithm. A modification is made for the algorithm, which reduces the number of twiddle factors evaluated and saves one group of decomposition. In addition, we propose an implementation method for the MSRFFT algorithm. The method transforms an L-shape butterfly of the MSRFFT algorithm into several non-L-shape butterflies, and makes the data-path of MSRFFT regular similar to the radix-2 FFT. We implement the algorithm on an AMD Opteron processor 4122 CPU and an NVIDIA GeForce GT 750M GPU by the proposed implementation method, and the results show that the MSRFFT algorithm consumes less time than radix-2/4 and radix-2/8 FFT algorithms for small scale DFTs.

The rest of this paper is organized as follows. The simplified MSRFFT algorithm is described and its performance is analyzed in Section 2. Section 3 presents the implementation method for the simplified MSRFFT algorithm, as well as gives a comparison of CPU time of radix-2 FFT, SRFFT and MSRFFT. Section 4 discusses the GPU parallelism and gives GPU time of radix-2 FFT and MSRFFT. Section 5 concludes this paper.

2. Simplified MSRFFT algorithm

Given a length-L sequence $x(l)$, its DFT is also a length-L sequence defined by

$$X(k) = \sum_{l=0}^{L-1} x(l) w_L^{lk}, \qquad k = 0, 1, ..., L-1, \quad (1)$$

where $w_L^{lk} = e^{-j(2\pi lk/L)}$, $j=\sqrt{-1}$, and the length-L is assumed to be power-of-two.

2.1. Algorithm

Many algorithms can compute the DFT in Equation (1). It is well known that for powers-of-two DFTs, the MSRFFT algorithm requires the lowest arithmetic operations among existing algorithms in the literature until 2014. However, the following several aspects affect its performance and practicality.

– The L-shape butterfly makes its hardware implementation more complex.
– No specifical method is presented for its implementation. If its implementation is conducted according to the methods presented in [10, 15, 22, 24, 32], it will be very difficult.
– MSRFFT requires more evaluations of twiddle factors than other algorithms.

Hence, it is interesting to (1) optimize the memory management of the MSRFFT algorithm and (2) present an implementation method for the algorithm. In the following, we consider to simplify the MSRFFT algorithm. The simplified MSRFFT algorithm provides

$$\begin{aligned} X(k) = U(k) &+ (Z(k)t(L,k) \\ &+ Z^{'}(k)t^{*}(L,k))s(L,k), \quad (2) \\ &k = 0, 1, ..., L-1, \end{aligned}$$

for the decomposition the DFT in Equation (1), where $t(L, k)$ and $s(L, k)$ are defined by

$$s(L,k) = \begin{cases} 1, & L \le 4, \\ s(L/4,k)\cos(2\pi k_4/L), & L \le L/8, \\ s(L/4,k)\sin(2\pi k_4/L), & L/8 < L < L/4, \end{cases} \quad (3)$$

and

$$t(L,k) = \begin{cases} 1, & k = 0, \\ 1 - j\tan(2k\pi/L), & L \le L/8, \\ \cot(2k\pi/L) - j, & L/8 < N < L/4, \end{cases} . \quad (4)$$

The three sub-DFT of $U(k)$, $Z(k)$, and $Z^{'}(k)$ are defined as follows:

$$U(k) = \sum_{l=0}^{L/2-1} x(l) w^{lk}, \quad k = 0, 1, ..., L/2-1, \quad (5)$$

$$Z(k) = \sum_{l=0}^{L/4-1} x(4l+1) w^{lk}, \quad k = 0, 1, ..., L/4-1, \quad (6)$$

$$Z^{'}(k) = \sum_{l=0}^{L/4-1} x(4l-1) w^{lk}, \quad k = 0, 1, ..., L/4-1. \quad (7)$$

The sub-DFT $U(k)$ in Equation (2) will be decomposed recursively according to Equation (2) until its length is smaller than 4. The decomposition of the sub-DFTs $Z(k)$ and $Z^{'}(k)$ in Equation (2) provides

$$\tau(k) = V(k) + B(k)t(M,k) + B^{'}(k)t^{*}(M,k), \quad k = 0, 1, ..., M-1, \quad (8)$$

for more efficient computation, where M is the length of this sub-transform, $\tau(k)$ in Equation (8) is derived from $Z(k)$ and $Z^{'}(k)$ in Equation (2), and the three sub-DFTs of $V(k)$, $B(k)$, and $B^{'}(k)$ in Equation (8) are defined by

$$V(k) = \sum_{m=0}^{M/2-1} x(m) w^{mk}, \quad k = 0, 1, ..., M/2-1, \quad (9)$$

$$B(k) = \sum_{m=0}^{M/4-1} x(4m+1) w^{mk}, \quad k = 0, 1, ..., M/4-1, \quad (10)$$

$$B^{'}(k) = \sum_{m=0}^{M/4-1} x(4m-1) w^{mk}, \quad k = 0, 1, ..., M/4-1. \quad (11)$$

$B(k)$ and $B^{'}(k)$ in Equation (8) will be decomposed recursively according to Equation (8) until lengths of all their subsequent sub-DFTs are no greater than 4. The sub-DFT $V(k)$ in Equation (8) provides decomposition of

$$V(k) = F(k)s(N/2, k)/s(2N, k) + (C(k)t(N, k) + C^{'}(k) t^{*}(N, k))s(N, k)/s(2N, k), \qquad (12)$$

for more efficient computation, where N is the length of this sub-transform. The sub-DFTs of $F(k)$, $C(k)$, and $C^{'}(k)$ are defined as follows:

$$F(k) = \sum_{n=0}^{N/2-1} x(n)w^{nk}, \quad k = 0, 1, ..., N/2 - 1, \qquad (13)$$

$$C(k) = \sum_{n=0}^{N/4-1} x(4n+1)w^{nk}, \quad k = 0, 1, ..., N/4 - 1, \qquad (14)$$

$$C^{'}(k) = \sum_{n=0}^{N/4-1} x(4n-1)w^{nk}, \quad k = 0, 1, ..., N/4 - 1. \qquad (15)$$

The three sub-DFTs in Equation (12) will be recursively decomposed according to Equation (8) until its lengths are no greater than 4.

We now summarize the scheme of the simplified MSRFFT algorithm for computing length-N=2^m DFTs. The length-N DFT is decomposed into one length-$N/2$ sub-DFT and two length-$N/4$ sub-DFTs in Equation (2). The length-$N/2$ sub-DFT is recursively decomposed in Equation (2). Each of the length-$N/4$ sub-DFTs is decomposed into one length-$N/8$ sub-DFT and two length-$N/16$ sub-DFTs in Equation (8). The length-$N/16$ sub-DFTs are recursively decomposed in Equation (8). The length-$N/8$ sub-DFT is decomposed into one length-$N/16$ sub-DFT and two length-$N/32$ sub-DFTs in Equation (12). The sub-DFTs created by Equation (12) are recursively decomposed in Equation (8).

2.2. Arithmetic operations

Of course, we need to compare the number of flop operations required for computing a length-N DFT in the simplified algorithm and the original algorithm.

It is clearly seen that, in an algorithm based on Equation (2), the substitution of $t(N, k)$ for w_N^k saves 4 real multiplications compared to $\mathbf{newfft}_N(x_n, s)$ of [13] per value of k of the loop except for the special case of k=0, which offsets 4 real multiplications by $s(N, k)$ in the same condition. In other words, the number of arithmetic operations in the algorithm based on Equation (2) is the same as that in $\mathbf{newfft}_N(x_n, s)$ of [13]. In an algorithm based on Equation (8), the number of arithmetic operations is clearly the same as that in $\mathbf{newfftS}_N(x_n, s)$ of [13], since both Equation (8) and $\mathbf{newfftS}_N(x_n, s)$ of [13] are all the same. In Equation (12), the number of operations required by computing $F(k)s(N/2, k)/s(2N, k)$ is the same as that in $\mathbf{newfftS4}_N(x_n)$ in [13], and the rest of Equation (12) is the same as $\mathbf{newfftS2}_N(x_n)$ of [13]. That is, the number of operations required in an algorithm based on Equation (12) is the same as the total flops required by $\mathbf{newfftS2}_N(x_n)$ and $\mathbf{newfftS4}_N(x_n)$ in [13]. From the above discussion, one can see that both algorithms have the same number of arithmetic operations.

2.3. Evaluation of twiddle factors

We now count the number of real coefficients evaluations or accesses to the lookup table for computing a length-N DFT in the original and simplified algorithms. Assume that all coefficients will be evaluated and stored in a lookup table in advance, and no coefficient will be initiated and kept in registers of a processor during the processing time. For clarity, in counting process we neglect the special case of k=0 and k=$N/8$.

The function $\mathbf{newfft}_N(x_n, s)$ of [13] needs to evaluate the coefficients $\sin(2\pi k/N)s(N/4, k)$ and $\cos(2\pi k/N)s(N/4, k)$ for $0 \le k < N/4$. The coefficient $s(N, k)$ has the property that $s(N, N/4 - k)$ = $s(N, k)$ for $0 \le k < N/8$. It is easy known that $\sin(2\pi(N/4 - k)/N)s(N/4, N/4 - k) = \cos(2\pi k/N)s(N/4, k)$ for $0 \le N < N/4$. The required number is $L/4$ for a recursion with a length-L sub-DFT (or DFT). The total required number of coefficient evaluation is $N/2 - 1$ for all recursion. Equation (2) needs to evaluate real coefficients $\tan(2\pi k/N)$ for $0 \le k < N/8$ and $s(N, k)$ for $0 \le k < N/4$. The required number of coefficient $\tan(2\pi k/N)$ evaluations is $N/8$. The number of coefficients $s(N, k)$ evaluation is $N/4 - 1$ for all recursion. The total number of coefficient evaluation is $3/8N - 1$ in Equation (2).

Equation (8) and $\mathbf{newfftS}_N(x_n, s)$ of [13] all contain coefficient $\tan(2\pi k/N)$ for $0 \le k < N/8$ where N is the length of the corresponding sub-DFT.

Equation (8) does not need to evaluate this coefficient, since this coefficient has been evaluated in Equation (2). The function $\mathbf{newfftS}_N(x_n, s)$ of [13] requires to evaluate this coefficient, and the number of coefficient evaluation is $1/32N$ since the maximum length of sub-DFTs computed in Equation (8) is $N/4$ for computing one length-N DFT.

Both Equation (12) and $\mathbf{newfftS2}_N(x_n)$ of [13] all require to evaluate the coefficient $s(N, k)/s(2N, k)$ for $0 \le k < N/2$ where N is the length of the corresponding sub-DFT. The coefficient has the property that $s(N, N/2 - k)/s(2N, N/2 - k)=s(N, k)/s(2N, k)$ for $0 \le k < N/4$. The number of coefficient evaluation is $N/16 - 1$ for all recursion in computing one length-N DFT since the maximum length of sub-DFTs computed in $\mathbf{newfftS2}_N(x_n)$ of [13] is $N/8$. Both Equation (12) and $\mathbf{newfftS4}_N(x_n)$ of [13] all require to evaluate the coefficient $s(N/2, k)/s(2N, k)$. The coefficient is $\cos(2\pi k/2N)$ for $0 \le k < N/4$ and $\sin(2\pi k/2N)$ for $N/4 \le k < N/2$. This means that the required number of coefficient evaluation is $N/32$ for all recursion in computing one length-N DFT. The total number required by Equation (12), and functions $\mathbf{newfftS2}_N(x_n)$ and $\mathbf{newfftS4}_N(x_n)$ of [13] are all $3/32N - 1$.

From the above discussion, one can see that the number of real coefficients evaluation is $15/32N - 2$ in the simplified algorithm, and $5/8N - 2$ in the original algorithm. That is, the simplified algorithm reduces the number of coefficient evaluation by $5/32N$ compared to the original algorithm.

3. Serial implementation of simplified MSRFFT

3.1. Proposed implementation method

From architecture point of view, regularity is more significant than computational complexity. Regularity of FFT algorithms can be categorized by data-path and computation. We can use dynamic voltage and frequency scaling (DVFS) [17] and component dynamic scheduling (CDS) [10], [32] techniques to minimize the irregularity of computation. The DVFS and CDS techniques can reduce energy consumption and even the number of components required by a FFT processor. The irregularity of data-path can be solved by permuting orders of intermediate results [22], [15]. In this section, we presents an implementation method that does not need to change the orders of intermediate results, and makes the data-path of the MSRFFT algorithm regular similar to that of the radix-2 FFT algorithm.

To implement the radix-2/4 FFT algorithm on general processors, the paper [24] developed an efficient, serial Fortran program. The structure of the program is very similar to that of Cooley-Tukey FFT algorithm. In the program, a butterfly indexing scheme was used which allows the program to calculate the radix-2/4 FFT algorithm stage by stage. The proposal of the indexing scheme is to find the next part in which butterflies need to be calculated. The process is bouncing and discontinuous, namely, there exists memory and control divergence which will cause significant performance delay on GPUs for all DFT, and on general processors for large scale DFTs. For some specific applications (hardware implementation, necessity of reducing the overhead, or the memory occupation) it may be useful to increase the regularity of the algorithm. This can be done by the use of a butterfly with permuted outputs [22]. The contributions of the paper [15] consists of mapping the radix-2/4 FFT algorithm to a constant geometry structure, improving the regularity on their proposed pipelined architecture.

The proposed method implements a DFT in terms of block rather than butterfly. A block consists of consecutive butterfly-units of two-to-two (BUs-2). A BU-2 is similar to a butterfly of radix-2 FFT algorithm, with two inputs, two outputs, a complex additions, a complex subtract, and 0 to 2 complex multiplications. The BUs-2 can be classified by complex multiplications into the following four types:

$$\begin{cases} x(n) = x(n) + x(n + N/2)s(N, n), \\ x(n + N/2) = x(n) - x(n + N/2)s(N, n), \end{cases} \tag{16}$$

$$\begin{cases} x(n) = x(n)t(2N, n) + x(n + N/2)t^*(2N, n), \\ x(n + N/2) \\ \quad = -j(x(n)t(2N, n) - x(n + N/2)t^*(2N, n)), \end{cases} \tag{17}$$

$$\begin{cases} x(n) = x(n) + x(n + N/2) \\ x(n + N/2) = x(n) - x(n + N/2), \end{cases} \tag{18}$$

$$\begin{cases} x(n) = x(n)(s(N/2, n)/s(2N, n)) \\ \qquad\qquad + x(n + N/2)(s(N, n)/s(2N, n)) \\ x(n + N/2) = x(n)(s(N/2, n)/s(2N, n)) \\ \qquad\qquad - x(n + N/2)(s(N, n)/s(2N, n)), \end{cases} \tag{19}$$

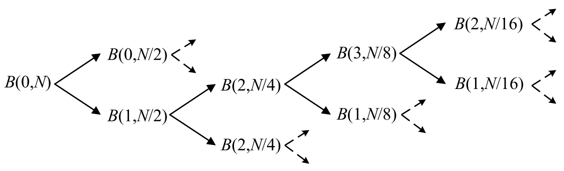

Fig. 1. Decomposition of a DFT in terms of Block.

where n=0,1,...,N/2-1, and N is the width of the block that is twice the number of BUs-2 in the block.

A block only allows a class of BUs-2 in it. Blocks are classified by the types of BUs-2. Blocks are classified by BUs-2 in a block and their type. A block in the BUs-2 in Equations (16), (17), (18), or (19) is banked type 0, 1, 2 or 3. The types of blocks can be one-to-one correspondence with decomposition of blocks. Let $B(i, N)$ be a block of type i and width N, and let symbol $\rightarrow$ be an action of decomposition. A DFT of length-N is $B(0, N)$. The recursive decomposition of blocks starts with the block $B(0, N)$ until lengths of all sub-blocks are equal 2. Block decomposition obey the following four rules.

1. $B(0, N) \rightarrow B(0, N/2)B(1, N/2)$. This decomposition appears only in the first stage.
2. $B(1, N) \rightarrow B(2, N/2)B(2, N/2)$. This decomposition appears in the right part of Equations (2), (8), and (12).
3. $B(2, N) \rightarrow B(3, N/2)B(1, N/2)$. This decomposition corresponds to the left part of Equation (8).
4. $B(3, N) \rightarrow B(2, N/2)B(1, N/2)$. This block decomposition corresponds to the left decomposition of Equation (12).

Figure 1 shows decomposition of a DFT in terms of blocks. In the figure, the dashed lines represent the omitted decomposition. Figure 2 shows the dataflow of one 16-points DFT, where BU$_i$ denotes that the type of the BU-2 is i. Let us see the figure from right to left. In the first stage, there is only one length-16 block with type 0. The length-16 block is split into two length-8 blocks with types 0 and 1 in the second stage. The two length-8 blocks are split into four length-4 blocks with types 0, 1, 2, and 2 in the third stage. Finally, the four length-4 blocks are decomposed into eight length-2 blocks with types 0, 1, 2, 2, 3, 1, 3, and 1 in the four stage. In this way, the data-path of the simplified MSRFFT algorithm is the same as that of radix-2 FFT algorithm, except for the positions of rotating factors.

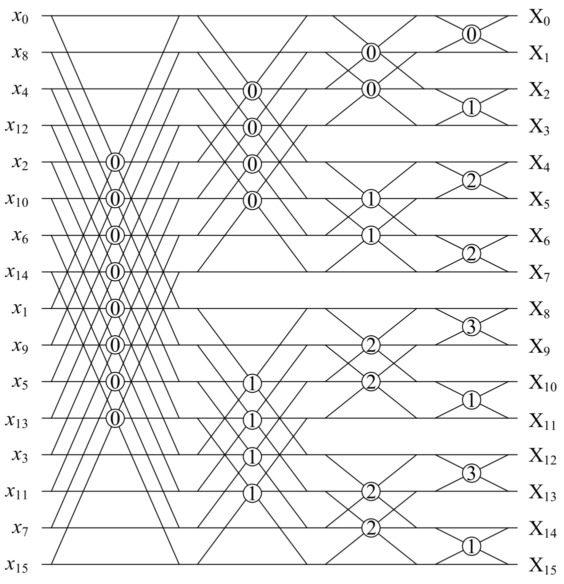

Fig. 2. Dataflow of a 16-points DFT.

A type-table is used to store the types of blocks. The type-table can be obtained by block decomposing. For a fixed-length DFT, in order to reduce computational complexity, its type-table can be evaluated in advance. The implementation of a block is to read its block type from the type-table and then perform the corresponding BUs-2 according to the type of the BUs-2. Each type requires 2-bits memory to store. A length-N DFT requires to store $N/4$ block types, since the number of blocks is at most $N/2$ and the block whose serial number is out of $N/4$ can also directly refer to the type-table. For a length-N DFT, no matter what stage the j-block is in, the type of the j-th block can refer to

1. the j-th entry in the type-table for $j < N/4$.
2. the (j mod $N/8$)-th entry in the type-table for $j > N/4$ and $j \neq 3N/8$.
3. the $N/8$-th entry for j=$N/4$ or $j = 3N/8$, if the value of the $N/8$-th entry in the type-table is 2 or 3, the type of j-th block is 3 or 2.

Figure 3 shows the type-table of length-256 DFTs, and length-N=4, 8, 16, 32, 64, 128 and 256 DFTs can lookup this table to determine their types of blocks.

To summarize, the butterflies of the simplified MSRFFT algorithm can be expressed in BUs-2 that are non-L-shape. The BUs-2 are categorized by multiplications into four types. A block consists of consecutive BUs-2 of the same type. Blocks are classified by the types of BUs-2 into four types. The

0															15
0	1	2	2	3	1	3	1	2	1	2	2	2	1	2	2

16															31
3	1	2	2	3	1	3	1	3	1	2	2	3	1	3	1

32															47
2	1	2	2	3	1	3	1	2	1	2	2	2	1	2	2

48															63
2	1	2	2	3	1	3	1	2	1	2	2	2	1	2	2

Fig. 3. State Table of 256-points DFTs.

types of blocks can be computed and stored to a state-table in advance. The DFT can be implemented through looking up the state-table and performing the corresponding BU-2. In this way, the data-path of the simplified MSRFFT algorithm is completely the same as that of the radix-2 FFT algorithm.

3.2. *Three loops program*

A C/C++ program of the simplified MSRFFT algorithm based on the type-table has been worked out, and its algorithm is showed in Algorithm 1. By comparing Algorithm 1 with the program of SRFFT in [24], we can see that the data-path of the simplified MSRFFT algorithm is regular. For computing a length-N=2^M DFT, the three loops C++ program steps through $M-1$ stages in the outer loop, deals with all blocks in the middle loop, and performs all BUs-2 in the inner loop. Before the three loops, a coefficient-table and a type-table are evaluated and stored, and the input terms are permuted.

Algorithm 1 Regular implementation of the simplified MSRFFT algorithm

```
function  $x_{n=0...N-1} \leftarrow \mathbf{MSRFFT\_DIT}_N(x_n)$
  {Computes length-$N$=$2^M$ DFT with the simplified MSRFFT DIT algorithm}
  Initiate coefficient-table and type-table
  Permute the orders of input terms
  span ← 1
  number_block ← $N/2$
  for $i \leftarrow M$ to 1 do
    for $j \leftarrow number\_block$-1 to 0 do
      $m \leftarrow j \times$span$\times 2$
      $n \leftarrow m + span$
      for $k \leftarrow 0$ to $span - 1$ do
        For a BU-2 of inputs $x(m)$ and $x(n)$,
          calculate the outputs according to
          the type of the $j$-th block in the type-table
        $m \leftarrow m + 1$
        $n \leftarrow n + 1$
      end for
    end for
    $number\_block \leftarrow number\_block/2$
    $span \leftarrow span \times 2$
  end for
  return
```

Table 1
Execution time of algorithms (ms)

N	Radix-2	Radix-2/4	Radix-2/8	MSRFFT
1024	0.062	0.016	0.016	0.016
2048	0.063	0.031	0.047	0.031
4096	0.140	0.078	0.094	0.078
8192	0.359	0.188	0.234	0.172
16384	0.734	0.438	0.531	0.406
32768	1.625	0.984	1.141	0.906
65536	3.766	2.135	2.469	2.063
131072	7.969	4.547	5.547	4.984
262144	23.672	11.203	12.484	13.641

Ref. [13] did not make a comparison of execution time. Due to more memory required, some researchers mentioned that the MSRFFT algorithm dissipates more time for computing a DFT than some previous published algorithms. Now, our programs are tested running on an AMD Opteron processor 4122 CPU which provides L2 cache of 512kB$\times$4 and L3 cache of 6MB. All programs are executed repeatedly for 100 times. Execution time, given in Table 1, shows that the simplified MSRFFT consumes less time than algorithms in [2, 5, 6] for $N < 131072$. Due to the requirement of more memory, execution time of the simplified algorithm actually increases when N is greater than 131072.

4. Parallelism on GPU

In order to estimate the effect of our implementation method on GPU, we devise a compute unified device architecture (CUDA) program for GPU parallelism where we can compare our implementation method to other methods in GPU time. For a length-$N = 2^m$ DFT on GPU, we implement a mixture of MSRFFT and the radix-2 FFT. The first $m-4$ stages are implemented with MSRFFT, and the rest four stages are implemented with the radix-2 FFT.

NVIDIA's CUDA is a new GPU architecture. It is a programming language based on C, and allows more flexible operations beyond stream programming for such "irregular" kernels by extensive multi-threading and the ability for the threads to share data rapidly via shared memory. The programming language CUDA allows for a block of threads to be specified easily as arrays, SIMD-like specifications. However, there are still many CUDA peculiarities that makes their straightforward application difficult to attain the expected performances. One difficulty is that, many of the accelerators are installed in I/O expansion slots such as the PCI-Express interface; therefore the data transfer between the host CPU and device often occupies a large percentage of the total execution time. This makes it difficult for accelerators to improve the performance of memory intensive applications like FFT. As a result, the currently reported results of FFT on GPUs [12, 20] have been only on par with conventional CPUs at best, indicating that real performance of GPUs have not been exploited yet despite the hopes. NVIDIA's CUDA provides a FFT library, CuFFT, which can accelerate the implementation of FFT achieving the maximum 10 times speedup. However, the efficiency of CuFFT on some GPUs such as NVIDIA GeForce GT 750M is very poor.

Table 2 gives GPU time of radix-2 FFT and MSRFFT on NVIDIA GeForce GT 750M for a length-2048 DFT. For NVIDIA GeForce GT 750M, saders are 384 unified, bus width is 128 Bit, and memory size is 2048 MB. A higher-performance FFT parallelism is very complex, in which one have to think about various techniques to improve the performance, including the regularity of the FFT algorithm, the shared memory utilization, in-place or out-of-place computation, the method to permute data-order, and the lookup table utilization etc. In our program, we consider only the regularity of the implementation, and measure whether the proposed implementation method can or not improve the FFT parallelism on GPU. Thus, our program is simple, containing nothing but FFT regularity. The longest length that our program can compute is 2048 points. In order to improve the FFT regularity, we develop a specialized function for computing length-16 DFTs of the fourth stages from outputs. To improve the performance, the function utilizes only registers and not uses the on-chip shared memory. We devise four schemes to compute a DFT on GPU, i.e., the radix-2 FFT, the MSRFFT algorithm, the mixture algorithm of MSRFFT and the specialized function for length-16 DFTs, and the mixture algorithm of MSRFFT and the radix-2 FFT (in which the radix-2 FFT is used in the final four stages). The GPU time in Table 2 shows that the fourth scheme consumes the least time for computing the length-2048 DFT on NVIDIA GeForce GT 750M and the MSRFFT algorithm consumes the most time, indicating the proposed implementation method has provided a higher-performance implementation for MSRFFT GPU parallelism.

Table 2
GPU time of radix-2 FFT and MSRFFT for a length-2048 DFT

Radix-2	MSR	MSR+Register	MSR+Radix-2
0.021502	0.022728	0.020704	0.019712

5. Conclusion

This paper modified and simplified the MSRFFT algorithm, and the number of real coefficients evaluated by $5/32N$ and the number of groups of decomposition from 4 to 3. A novel implementation method has been proposed for the simplified MSRFFT algorithm. The method makes the datapath of the simplified MSRFFT algorithm regular similar to that of the radix-2 FFT algorithm. The method can be applied to other algorithms such as radix-2/4, and radix-2/8 etc, making their data-path regular similar to mixed-radix algorithms. A three loops serial program and a GPU parallelism program have been proposed for the simplified algorithm, and the results show (1) the simplified MSRFFT algorithm consumes less time than the radix-2, radix 2/4, and radix 2/8 FFT algorithms for computing a length-$N < 131072$ DFT on the general processors, and (2) the proposed scheme for MSRFFT GPU parallelism can obtain higher-performance than the radix-2 FFT due to the improvement of irregularity of MSRFFT.

Acknowledgments

This work was partially funded by the National Natural Science Foundation of China (Grant Nos. 61472136), the project of the National Science Foundation for Distinguished Young Scholars of Hunan (12JJ1011), Torch Program of Ministry of science and technology (2015GH712901), and Natural Science Foundation of Hunan Province of China under Grant (2015JJ5021).

References

[1] G. Bi, G. Li and X. Li, A unified expression for split-radix dft algorithms, *In Communications, Circuits and Systems (ICCCAS), 2010 International Conference on*, 2010, pp. 323–326. IEEE.

[2] S. Bouguezel, M.O. Ahmad and M.N.S. Swamy, A new radix-2/8 fft algorithm for length-q$\times 2^{m}$ dfts, *Circuits and*

Systems I: Regular Papers, IEEE Transactions on **51**(9) (2004), 1723–1732.

[3] K.J. Bowers, R.A. Lippert, R.O. Dror and D.E. Shaw, Improved twiddle access for fast fourier transforms, *Signal Processing, IEEE Transactions on* **58**(3) (2010), 1122–1130.

[4] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston and P. Hanrahan, Brook for gpus: stream computing on graphics hardware, *In ACM Transactions on Graphics (TOG)* **23** (2004), 777–786. ACM.

[5] J.W. Cooley and J.W. Tukey, An algorithm for the machine calculation of complex fourier series, *Math Comput* **19**(90) (1965), 297–301.

[6] P. Duhamel and H. Hollmann, Split-radix fft algorithm, *Electron Lett* **20** (1984), 14–16.

[7] P. Duhamel and M. Vetterli, Fast fourier transforms: a tutorial review and a state of the art, *Signal Processing* **19**(4) (1990), 259–299.

[8] P. Duhamel and H. Hollmann, Split radix'fft algorithm, *Electronics Letters* **20**(1) (1984), 14–16.

[9] M. Frigo and S.G. Johnson, The design and implementation of fftw3, *Proceedings of the IEEE* **93**(2) (2005), 216–231.

[10] J. Garcia, J.A. Michell and A.M. Buron, Vlsi configurable delay commutator for a pipeline split radix fft architecture, *Signal Processing, IEEE Transactions on* **47**(11) (1999), 3098–3107.

[11] A.M. Grigoryan and S.S. Agaian, Split manageable efficient algorithm for fourier and hadamard transforms, *Signal Processing, IEEE Transactions on* **48**(1) (2000), 172–183.

[12] E. Gutierrez, S. Romero, M.A. Trenas and E.L. Zapata, Memory locality exploitation strategies for fft on the cuda architecture, *In High Performance Computing for Computational Science-VECPAR 2008*, 2008, pp. 430–443. Springer.

[13] S.G. Johnson and M. Frigo, A modified split-radix fft with fewer arithmetic operations, *IEEE Trans Signal Processing* **55**(1) (2007), 111–119.

[14] I. Kamar and Y. Elcherif, Conjugate pair fast fourier transform, *Electronics Letters* **25**(5) (1989), 324–325. doi:10.1049/el:19890225

[15] J. Kwong and M. Goel, A high performance split-radix fft with constant geometry architecture, *In Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2012, pp. 1537–1542. IEEE.

[16] E.S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware, *In Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (CDROM)*, 2001, pp. 55–55. ACM.

[17] E.G. Larsson and O. Gustafsson, The impact of dynamic voltage and frequency scaling on multicore dsp algorithm design [exploratory dsp], *Signal Processing Magazine, IEEE* **28**(3) (2011), 127–144.

[18] W.R. Mark, R.S. Glanville, K. Akeley and M.J. Kilgard, Cg: A system for programming graphics hardware in a c-like language, *In ACM Transactions on Graphics (TOG)*, volume 22, 2003, pp. 896–907. ACM.

[19] J.B. Martens, Recursive cyclotomic factorization–a new algorithm for calculating the discrete fourier transform, *Acoustics, Speech and Signal Processing, IEEE Transactions on* **32**(4) (1984), 750–761.

[20] K. Moreland and E. Angel, The fft on a gpu, *In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 2003, pp. 112–119. Eurographics Association.

[21] L. Nyland, M. Harris and J. Prins, Fast n-body simulation with cuda, *GPU Gems* **3** (2007), 677–695.

[22] P. Duhamel, Implementation of split-radix fft algorithms for complex,real, and real symmetric data, *IEEE Transactions on Acoustics, Speech and Signal Processing* **34**(2) (1986), 285–295.

[23] H. Rashmanlou and R.A. Borzooei, Product vague graphs and its applications, *Journal of Intelligent & Fuzzy Systems* (2015), 1–12. Preprint(Preprint).

[24] H. Sorensen, M. Heideman and C. Burrus, On computing the split-radix fft, *IEEE Trans Acoust. Speech, Signal Processing* **34**(1) (1986), 152–156.

[25] M.J. Stock and A. Gharakhani, Toward efficient gpuaccelerated n-body simulations, *AIAA paper* **608** (2008), 7–10.

[26] D. Takahashi, An extended split-radix fft algorithm, *Signal Processing Letters, IEEE* **8**(5) (2001), 145–147.

[27] D. Tarditi, S. Puri and J. Oglesby, Accelerator: Using data parallelism to program gpus for general-purpose uses, *In ACM SIGARCH Computer Architecture News*, volume 34, 2006, pp. 325–335. ACM.

[28] M. Vetterli and H.J. Nussbaumer, Simple fft and dct algorithms with reduced number of operations, *Signal Processing* **6**(4) (1984), 267–278.

[29] C. Wang, Some properties of entropy of vague soft sets and its applications, *Journal of Intelligent & Fuzzy Systems* **29**(4) (2014), 1443–1452.

[30] N. Yalaoui, L. Amodeo, F. Yalaoui and H. Mahdi, Efficient methods to schedule reentrant flowshop system, *Journal of Intelligent & Fuzzy Systems* **26**(3) (2014), 1113–1121.

[31] R. Yavne, An economical method for calculating the discrete fourier transform, *in Proc AFIPS Fall Joint Computer Conf* **33** (1968), 115–125.

[32] W.-C. Yeh and J. Chein-Wei, High-speed and low-power split-radix fft, *Signal Processing IEEE Transactions on* **51**(3) (2003), 864–874.

[33] W. Zheng, K. Li and K. Li, Scaled radix-2/8 algorithm for efficient computation of length-$N = 2^m$ DFTs, *IEEE Transactions on Signal Processing* **62** (2014), 2492–2503. doi: 10.1109/TSP.2014.2310434