Journal of Zhejiang University SCIENCE A
ISSN 1673-565X (Print); ISSN 1862-1775 (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

Novel algorithm for complex bit reversal: employing vector permutation and branch reduction methods

Feng YU, Ze-ke WANG, Rui-feng GE

(*Department of Instrument Engineering, Zhejiang University, Hangzhou 310027, China*)

E-mail: osfengyu@zju.edu.cn; wzk_6_3_8@163.com; gratty.gratty@gmail.com

Received May 18, 2009; Revision accepted Aug. 14, 2009; Crosschecked Aug. 14, 2009

Abstract: We present novel vector permutation and branch reduction methods to minimize the number of execution cycles for bit reversal algorithms. The new methods are applied to single instruction multiple data (SIMD) parallel implementation of complex data floating-point fast Fourier transform (FFT). The number of operational clock cycles can be reduced by an average factor of 3.5 by using our vector permutation methods and by 1.1 by using our branch reduction methods, compared with conventional implementations. Experiments on MPC7448 (a well-known SIMD reduced instruction set computing processor) demonstrate that our optimal bit-reversal algorithm consistently takes fewer than two cycles per element in complex array operations.

Key words: Bit reversal, Vector permutation, Branch reduction, Single instruction multiple data (SIMD), Fast Fourier transform (FFT)

doi:10.1631/jzus.A0920290 **Document code:** A **CLC number:** TP30

INTRODUCTION

In the application of fast Fourier transform (FFT) algorithms, it is usually assumed that the input data are in a complex format (with both real and image parts). So it is logical to develop and optimize FFT algorithms for complex data. Without loss of generality, we assume that the storage of complex data is in a split format; i.e., that complex data consist of two arrays, one real, the other imaginary. We also assume that each element in the complex array is a single-precision floating-point number.

Many FFT algorithms mandate that the data are ordered by a bit-reversed permutation operation. This operational stage is always executed either before or after FFT butterfly computation and is called bit reversal. If the bit reversal were not implemented efficiently, FFT algorithms would slow down significantly (Chakraborty and Chakrabarti, 2008). Pre- or post-permutation can account for from 10% to 50% of the total computation time of FFT algorithms (Burrus, 1988). An FFT algorithm works best with in-place operation, where the output data in each stage can safely overwrite the input data to take advantage of the butterfly structure. In-place operations can have significant performance advantages over out-of-place operations because their cache size requirements are significantly lower.

A very efficient scalar computational algorithm for FFT bit reversal was given by Sundararajan *et al.* (1994). The reason for its high efficiency is because if the bit reversal of all even numbers from 0 to $N/4-1$ is computed, then the bit reversal of the remaining numbers can be computed at the cost of one operation of addition for each number; that is, it reverses a continuous array of N numbers by running a loop $N/8$ times. An earlier implementation presented by Yong (1991) repeats a similar loop $N/4$ times. Moreover, Sundararajan *et al.*(1994) presented an approach to reduce further the number of the above loops to $N/16$, or even fewer.

With the rapid advancement of semiconductor process, the clock speed of processors has increased dramatically in the past decade. Modern processors can execute instructions more efficiently. A single instruction multiple data (SIMD) parallel processing

vector engine further improves the performance of the microprocessor, especially for signal and multimedia processing applications. For example, the execution of vector permutation instruction costs just one processor cycle (Freescale Semiconductor, 2005), taking only 1 ns in a 1 GHz processor. So vectorization can yield significant performance advantages for many types of applications.

Modern processors are heavily pipelined, so program branching is rather costly. For example, when a pipeline encounters a branch, it may not have enough time to finish evaluating the branch condition before it is time to decide whether the branch will be taken. In this scenario, what the processor can do is prediction. If the prediction is wrong, instructions are on the misleading path and any results of the speculative execution must be purged from the pipeline, and then instructions must be fetched from the correct path (Freescale Semiconductor, 2005). Several efficient ways to eliminate this effect have been presented, such as loop unrolling and resolving branches before prediction (Freescale Semiconductor, 2005; 2007). In our opinion, a useful approach to handle this problem is to find a way to reduce the number of branches and to rewrite algorithms that work for all possible input conditions. Even though this practice may significantly increase the amount of code, it is still a rewarding approach.

In this paper, we present novel vector permutation and branch reduction methods to minimize the number of execution cycles for bit reversal algorithms. The proposed implementations, when using the vector permutation method only, can significantly reduce loop times and the number of load/store pairs compared with the corresponding scalar implementations. The proposed branch reduction method further improves the performance by eliminating the main loop 'if-else' statements that determine whether to swap between the index and its bit reversal, without increasing the computational complexity or the number of load/store pairs. Our experimental results show that the vector permutation method can significantly reduce the number of clock cycles, while the branch reduction method can further reduce the execution time.

We first present the background of the bit reversal algorithm and some conventional scalar bit reversal implementations. We then employ the vector permutation method to improve the in-place bit reversal algorithm and propose a novel branch reduction method to accelerate the bit reversal algorithm. Finally, experimental results from a PowerPC G4 processor (MPC7448) are presented.

BACKGROUND

Many popular and efficient algorithms for the bit reversal process have been developed since the invention of the Cooley-Tukey FFT algorithm. However, conventional algorithms, calculating the bit reversal one by one using for-loop, are not efficient. Popular improved algorithms can be classified into two classes: the first class needs an auxiliary small-sized table (Evans, 1987; 1989; Walker, 1990), and the second class performs an efficient bit reversal algorithm without tables (Yong, 1991; Sundararajan *et al.*, 1994). The first class is generally faster, provided that the size of the array plus the auxiliary table does not exceed the size of the cache, and that the auxiliary table can be reused. This approach reduces the computational time at the cost of adding an auxiliary table. However, this class may require bit reversal of constant length, and it works poorly for large-sized arrays. When the size of the auxiliary table is large, cache misses may occur frequently.

We prefer the computational approach of the second class, because Yong (1991) and Sundararajan *et al.* (1994) have presented efficient bit reversal algorithms to reduce the computational redundancy.

The heuristic approach presented by Yong (1991) reverses a continuous array of N numbers by running a loop $N/4$ times, while the conventional method repeats a similar loop $N-1$ times. Specifically, if the bit reversal of all even numbers from 0 to $N/2-1$ is formed, then the bit reversal of each of the remaining numbers can be computed at the cost of one addition operation for each number. Given an index couple (I, J), where the bit reversal of I is J: if I is an even integer from 0 to $N/2-1$, then J is even since the most significant bit (MSB) of I ($I<N/2$) is zero. If $I<J$, then the elements of an array indexed by I and J are to be swapped, as well as the couple ($I+1+N/2$, $J+1+N/2$). However, ($I+N/2$, $J+1$) does not need to be swapped since $I+N/2$ is always greater than $J+1$, while ($I+1$, $J+N/2$) is certain to be swapped because $I+1$ is definitely less than $J+N/2$.

Further improvements to this algorithm were proposed by Sundararajan *et al.*(1994), who presented a more efficient algorithm to repeat a similar loop $N/8$ times; that is, if the bit reversal of all even numbers from 0 to $N/4-1$ is formed, then the bit reversal of each of the remaining numbers can be computed. In brief, if I is less than J, then the couples (I, J), ($I+N/2+1$, $J+N/2+1$) are to be swapped. If $I+N/4$ is less than $J+2$, then ($I+N/4$, $J+2$) and ($I+N/4+N/2+1$, $J+2+N/2+1$) are also to be swapped. ($I+1$, $J+N/2$) and ($I+1+N/4$, $J+N/2+2$) must be swapped, while the couples ($I+N/2$, $J+1$), ($I+N/2+N/4$, $J+2+1$) must definitely not be swapped. Sundararajan *et al.*(1994) proposed a methodology to further reduce the loop times to $N/16$, or even fewer.

Other approaches for performing bit reversal algorithms in vector operation (Pei and Chang, 2007) have produced significant results. Lokhmotov and Mycroft (2007) proposed that each source vector (W-element vector) provides for W target vectors, and each target vector takes elements from W source vectors, where W is equal to 4. Note that this algorithm can be applied also to in-place bit reversal.

Now we assume that there is an index couple (I, J), where I is an integer multiple of four (less than $N/4-1$) and the bit reversal of I is J. J is also an integer multiple of four (less than $N/4-1$), since the two MSBs and two LSBs (least significant bits) of I are 0. The bit reversal of $I+1$, $I+2$, and $I+3$ are $J+N/2$, $J+N/4$, and $J+3N/4$, respectively; that is, we can obtain four couples: (I, J), ($I+1$, $J+N/2$), ($I+2$, $J+N/4$), and ($I+3$, $J+3N/4$). Then, we can obtain another four couples: ($I+N/4$, $J+2$), ($I+1+N/4$, $J+N/2+2$), ($I+2+N/4$, $J+N/4+2$), and ($I+3+N/4$, $J+3N/4+2$) by adding $2^{\log(N/2)-2}=N/4$ to the number I and 2 to its bit reversal J. Similarly, we can obtain the other 8 couples: ($I+N/2$, $J+1$), ($I+1+N/2$, $J+N/2+1$), ($I+2+N/2$, $J+N/4+1$), ($I+3+N/2$, $J+3N/4+1$), ($I+3N/4$, $J+3$), ($I+1+3N/4$, $J+N/2+3$), ($I+2+3N/4$, $J+N/4+3$), and ($I+3+3N/4$, $J+3N/4+3$), as shown in Fig.1 (see p.1496). For example, the element indexed by ($I+3+N/4$) of the source array, with the value '7', is stored in the destination array indexed by ($J+2+3N/4$). Details of the vector permutation process were given by Lokhmotov and Mycroft (2007).

BIT REVERSAL IMPLEMENTATION USING VECTOR PERMUTATION

We propose a new computational algorithm for in-place complex bit reversal using vector permutation, based on Yong (1991) and Lokhmotov and Mycroft (2007). We denote the new computational implementation 'Yong-Lokhmotov':

Assume that a complex array indexed by I contains 16 elements whose offsets are I, $I+1$, …, $I+3$, $I+N/4$, …, $I+N/4+3$, $I+N/2$, …, $I+3N/4$, …, $I+3N/4+3$, respectively, and a complex array indexed by J also contains 16 elements whose offsets are J, $J+1$, …, $J+3$, $J+N/4$, …, $J+N/4+3$, $J+N/2$, …, $J+3N/4$, …, $J+3N/4+3$, respectively (Fig.2). The index I is an integer multiple of four (less than $N/4-1$) and its bit reversal J is also an integer multiple of four (less than $N/4-1$). If $I<J$, then the elements of the complex array indexed by I, J are performed under proper vector permutation and then swapped (Fig.2, p.1496).

The vector permutation method, by using a 4-element vector register, reduces the number of load/store pairs to one quarter of the corresponding scalar implementation at the cost of adding 8 vector permutation registers per 16 complex elements (8 vector registers). Details of the process of vector permutations were presented by Lokhmotov and Mycroft (2007).

If $I=J$, then real and imaginary elements indexed by I (or J) should be under proper permutation (Fig.3, p.1496); if $I<J$, do nothing; otherwise, swap back to the original sequence.

We have also proposed another new algorithm based on Sundararajan *et al.*(1994) and Lokhmotov and Mycroft (2007), and we denote the new algorithm 'Sundararajan-Lokhmotov'. Since this proposed algorithm is similar to the Yong-Lokhmotov algorithm, we omit the details here.

REVERSAL IMPLEMENTATION WITH THE NOVEL BRANCH REDUCTION METHOD

In the following, we present a novel branch reduction method to further accelerate the above algorithms, including the Yong, Sundararajan, Sundararajan-Lokhmotov, and Yong-Lokhmotov algorithms.

This method is based on the theory presented by Drouiche (2001): the number I and its bit reversal J are one-to-one mapping:

$$\sigma_p: W_p \to W_p,$$
$$I \to J = \sigma_p(I),$$

where $W_p=\{I\in\mathbb{N},\ 0\le I<2^p\}\subset\mathbb{N}$, and J denotes the bit reversal of I. We represent each index I as a p-bit binary notation $I=\alpha_{p-1}\alpha_{p-2}\cdots\alpha_2\alpha_1\alpha_0$.

If $I\in W_p$ and $I=\sum_{n=0}^{p-1}\alpha_n\times 2^n$, $\alpha_n\in\{0, 1\}$, then

$$\sigma_p(I)=J=\sum_{n=0}^{p-1}\alpha_n\times 2^{p-n-1}=\alpha_0\alpha_1\cdots\alpha_{p-3}\alpha_{p-2}\alpha_{p-1}.$$

Clearly, σ_p is involutive and bijective. Specifically, if I varies from 0 to $N-1$, then $J=\sigma_p(I)$ varies correspondingly in the range of $[0, N-1]$.

Now we apply this theory to the four algorithms.

Based on the algorithm presented by Yong (1991), we have analyzed the data flow of complex bit reversal and derived the following structure using one-to-one mapping between an index and its bit reversal. Given the couple (I, J), where J is bit reversal of I, we can analyze the Ith element of a real array and the Jth element of an imaginary array on one loop, and we store the imaginary array indexed by J with the element from the real array indexed by I, and vice versa. The same process also applies to the other three couples, ($I+1+N/2$, $J+1+N/2$), ($I+N/2$, $J+1$), and ($I+1$, $J+N/2$). When the index I loops from 0 to $N/2-1$ with step 2, the algorithm would terminate, with the whole real data stored in the imaginary array and the whole imaginary data stored in the real array. This method preserves the computational complexity and one load/store pair for each piece of data, compared with Yong's implementation. However, it does reduce the test condition ($I<J$ or $I=J$) in the main loop, compared with Yong's algorithm. The novel algorithm, denoted 'wzk-Yong', is presented as follows:

```
void inline swap_ri_ir(float *in_r, float *in_i, int i, int j)
{
   float temp1, temp2;
   temp1=in_r[i]; temp2=in_i[j];
   in_i[j]=temp1; in_r[i]=temp2;
}
```

```
/* in_r and in_i are addresses of real and imaginary arrays,
      respectively */
/* n is the number of real (or imaginary) arrays */
void wzk_Yong(float *in_r, float *in_i, int n)
{
   int i, k, j;   float temp1, temp2;
   int NB2, NB4, NB8, NBTP1, NBTP2;
   NB2=n/2;  NB4=n/4;
   NBTP1=NB2+1;  j=0;
   for (i=0; i<NB2; i+=2)
   {
      /* swap the couple (i, j) */
      swap_ri_ir(in_r, in_i, i, j);
      /* swap the couple (i+NBTP1, j+NBTP1) */
      swap_ri_ir(in_r, in_i, i+NBTP1, j+NBTP1);
      /* swap the couple (i+1, j+NB2) */
      swap_ri_ir(in_r, in_i, i+1, j+NB2);
      /* swap the couple (i+NB2, j+1) */
      swap_ri_ir(in_r, in_i, i+NB2, j+1);
      k=NB4;
      while (k<=j)  {j-=k;  k>>=1;}
      j+=k;   /* j is the index of the imaginary part */
   }
}
```

We have also applied the novel branch reduction method to Sundararajan's algorithm, resulting in another algorithm denoted 'wzk-Sundararajan'. wzk-Sundararajan is similar to wzk-Yong, except that index I loops from 0 to $N/4-1$ with step 2, so we omit the details here.

We apply the novel branch reduction method to the algorithm Yong-Lokhmotov using vector permutation. In particular, we first load 16 elements to 8 vector registers from a real array indexed by I and an imaginary array indexed by J. Second, we perform proper permutation on the 8 vector registers. Third, we store elements loaded from the real array indexed by I in an imaginary array indexed by J, and vice versa (Fig.4).

For example, the element of a real array indexed by $I+3N/4+2$, with its value e, is stored in an imaginary array indexed by $J+N/4+3$.

When I loops from 0 to $N/4-1$ with step 4, its bit reversal J will also be an integer multiple of four (less than $N/4-1$), and then the process of bit reversal will terminate, with its real data stored in the imaginary array and its imaginary data stored in the real array. The new algorithm is denoted 'wzk-Yong-Lokhmoto' (see the Appendix).

The similar new algorithm, denoted 'wzk-Sundararajan-Lokhmoto', can easily be derived based on the above algorithms wzk-Yong-Lokhmoto and Sundararajan. We omit the details here.

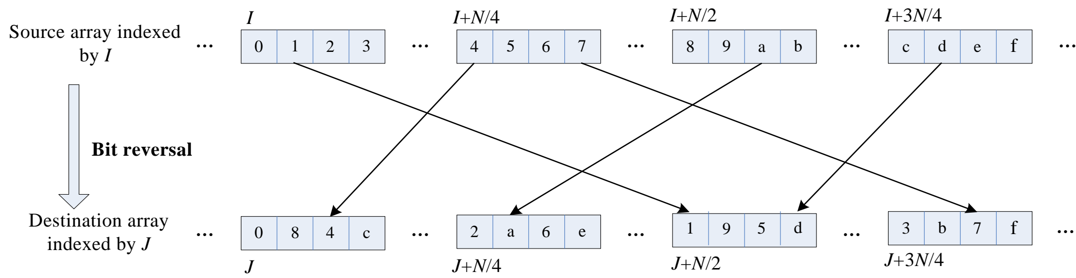

Fig.1 Source array maps to the destination array

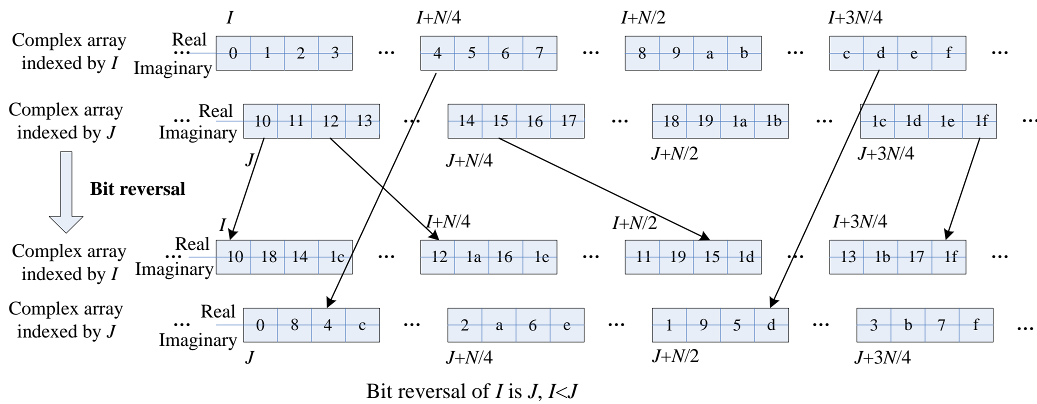

Fig.2 Elements indexed by *I* and *J* of a complex array are swapped (*I*<*J*)

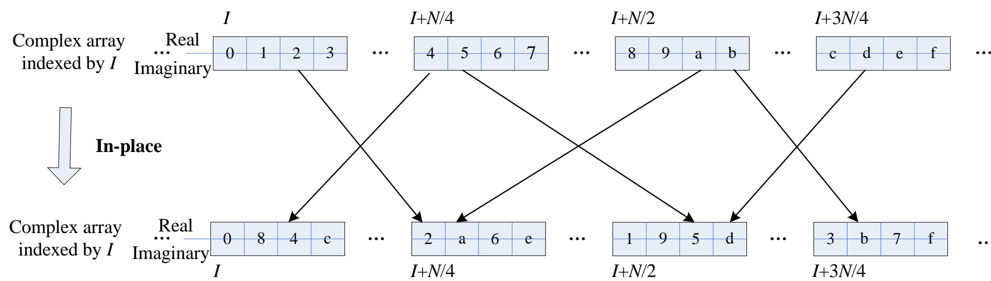

Fig.3 In-place permutation on the index *I* of a complex array

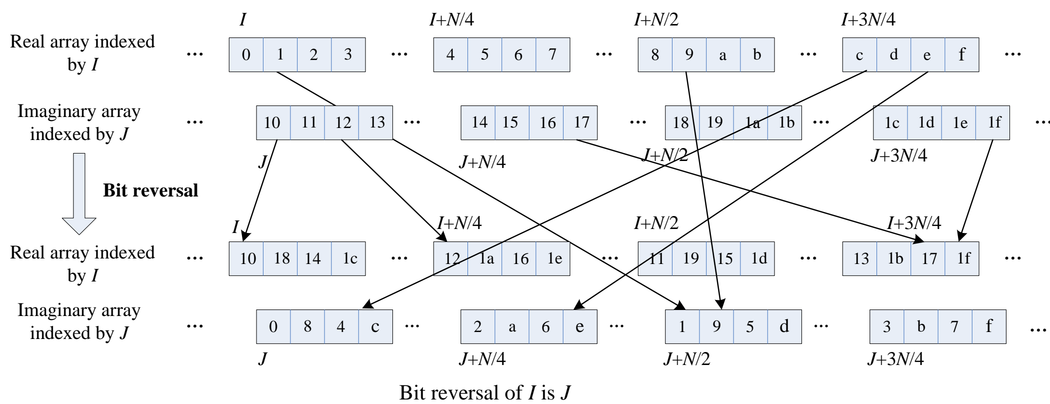

Fig.4 Branch reduction implementation for bit reversal using vector permutations

PERFORMANCE EVALUATION OF THE BRANCH REDUCTION METHOD

We have applied vector permutation and branch reduction methods to several conventional algorithms presented by Yong (1991), Sundararajan *et al.*(1994), and Lokhmotov and Mycroft (2007), yielding several novel algorithms, including scalar forms (wzk-Yong and wzk-Sundararajan) and vector forms (Yong-Lokhmotov, Sundararajan-Lokhmotov, wzk-Yong-Lokhmotov, and wzk-Sundararajan-Lokhmotov).

We have implemented the above six novel bit reversal algorithms in PowerPC G4 (MPC7448), which consists of a 1.0 GHz processor core, 32 kB separate L1 instruction and data caches, a 1 MB L2 cache, and vector engine Altivec (Freescale Semiconductor, 2005).

The above algorithms are compiled with the maximum optimization level (-O3) and executed in a real time operating system VxWorks 6.5. We have employed performance monitor facility of G4 to measure the used number of clock cycles (Freescale Semiconductor, 2005). We assume that all required instructions and data are presented in the L1 cache, whose size in G4 is 32 kB. So the size of bit reversal should be less than 32 kB.

The performances of the eight implementations are compared in Table 1. We assume that the conventional implementations refer to Yong's or Sundararajan's algorithm.

Intuitively, a complex array operation always takes less than two cycles per element when using our novel implementation (wzk-Sundararajan Lokhmotov).

The experimental results show that the bit reversal algorithm implementations, with both vector permutation and branch reduction methods, can achieve average reductions in the number of loops by a factor of 4, in the number of branches by a factor of 6.46, and in the number of clock cycles by a factor of 4.11, compared with the conventional implementation. For example, we compared the two implementations Sundararajan and wzk-Sundararajan-Lokhmotov. Then, we analyzed the vector permutation and branch reduction methods, respectively. Using only the vector permutation method, the bit reversal implementations (for example, Yong-Lokhmotov) can, on average, reduce the number of loops by a factor of 4 because of the 4-element vector register, the number of branches by a factor of 4.44 because of the reduced loop number, and the number of clock cycles by a factor of 3.54 compared with the conventional implementations (e.g., Yong). Using only the branch reduction method, the bit reversal implementations (e.g., wzk-Yong), on average, reduce the number of branches by a factor of 1.36, and the number of clock cycles by a factor of 1.11 compared with the conventional implementations (e.g., Yong), while retaining the same loop number. However, after we apply the branch reduction method to our implementation, we should swap the first addresses of the real array and

Table 1 Performance comparison of eight implementations

Impl.	Loop number					
	S=128	256	512	1024	2048	4096
Yong	32	64	128	256	512	1024
w-Y	32	64	128	256	512	1024
Sun.	16	32	64	128	256	512
w-S	16	32	64	128	256	512
Y-L	8	16	32	64	128	256
w-Y-L	8	16	32	64	128	256
S-L	4	8	16	32	64	128
w-S-L	4	8	16	32	64	128
Impl.	**Branch number**					
	S=128	256	512	1024	2048	4096
Yong	75	155	311	631	1263	2543
w-Y	63	127	255	511	1023	2047
Sun.	57	120	243	498	1001	2024
w-S	45	92	187	378	761	1528
Y-L	25	45	91	171	343	663
w-Y-L	15	31	63	127	255	511
S-L	12	25	45	91	171	343
w-S-L	7	15	31	63	127	255
Impl.	**Processor execution cycles**					
	S=128	256	512	1024	2048	4096
Yong	1030	2113	4181	8538	17041	34528
w-Y	992	1983	3875	7719	15405	30586
Sun.	880	1844	3697	7539	15156	30916
w-S	913	1794	3571	7140	14244	30803
Y-L	336	682	1291	2523	5016	10589
w-Y-L	278	532	1040	2025	4022	8322
S-L	274	508	953	1848	3696	7508
w-S-L	251	445	889	1729	3409	6936

Impl.: implementation; *S*: bit reversal size; w-Y: wzk-Yong; Sun.: Sundararajan; w-S: wzk-Sundararajan; Y-L: Yong-Lokhmotov; w-Y-L: wzk-Yong-Lokhmotov; S-L: Sundararajan-Lokhmotov; w-S-L: wzk-Sundararajan-Lokhmotov

the imaginary array when the process of bit reversal terminates, since the real data have been stored in the imaginary array, and vice versa. This method preserves the same number of load/store pairs and the same computational complexity as the corresponding conventional implementations.

CONCLUSION

In this paper, we present novel vector permutation and branch reduction methods to minimize the number of execution cycles. The proposed vector permutation method firstly reduces the number of loop times to one quarter of that of the corresponding conventional implementations, and secondly reduces the number of load/store pairs to one quarter of that of the corresponding scalar implementations, by using a 4-element vector register. The proposed branch reduction method eliminates from the main loop the 'if-else' statements that determine whether to swap between the index and its bit reversal. The experimental results on a PowerPC vector processor show that the vector permutation method can significantly reduce the clock cycles by a factor of 3.54, while the branch reduction method enables a further reduction by a factor of 1.11, compared with the conventional implementations.

However, we assume that the size of bit reversal is less than 32 kB and that all the computed data are already in a data cache. Also, we have not yet evaluated the performance of our proposed methods in other situations, for example, using computed data not in the cache. There are several bit reversal algorithms based on cache-optimal improvement, such as the cache-optimal bit-reversal algorithm using vector permute operations (COBRAVO) by Lokhmotov and Mycroft (2007) and the cache optimal bit-reversal algorithm (COBRA) by Carter and Gatlin (1998). We have not yet considered applications of bit reversal other than FFT, such as bit reversal on optical multi-trees (Jana and Sinha, 2008) and permuting streaming data using RAMs (Püschel *et al.*, 2009). Furthermore, Marti-Puig (2009) has implemented radix-2 FFT algorithms having the property that both inputs and outputs are addressed in natural order. We also have not exploited the parallelism potential of bit reversal in multi-core architecture and graphics processing units (GPUs) environment, where the FFT algorithm is optimized (Chen *et al.*, 2007; Lloyd *et al.*, 2008).

References

Burrus, C.S., 1988. Unscrambling for fast DFT algorithms. *IEEE Trans. Acoust. Speech Signal Process.*, **36**(7):1086-1087. [doi:10.1109/29.1631]

Carter, L., Gatlin, K.S., 1998. Towards an Optimal Bit-reversal Permutation Program. Proc. 39th Annual Symp. on Foundations of Computer Science, p.544-553. [doi:10.1109/SFCS.1998.743505]

Chakraborty, T.S., Chakrabarti, S., 2008. On Output Reorder Buffer Design of Bit Reversed Pipelined Continuous Data FFT Architecture. IEEE Asia Pacific Conf. on Circuits and Systems, p.1132-1135. [doi:10.1109/APCCAS.2008.4746224]

Chen, L., Hu, Z., Lin, J.M., Gao, G.R., 2007. Optimizing the Fast Fourier Transform on Multi-core Architecture. IEEE Int. Parallel and Distributed Processing Symp., p.1-8. [doi:10.1109/IPDPS.2007.370639]

Drouiche, K., 2001. A new efficient computational algorithm for bit reversal mapping. *IEEE Trans. Signal Process.*, **49**(1):251-254. [doi:10.1109/78.890370]

Evans, D., 1987. An improved digital-reversal permutation algorithm for the fast Fourier transforms. *IEEE Trans. Acoust. Speech Signal Process.*, **35**(8):1120-1125. [doi:10.1109/TASSP.1987.1165252]

Evans, D., 1989. A second improved digital-reversal permutation algorithm for the fast Fourier transforms. *IEEE Trans. Acoust. Speech Signal Process.*, **37**(8):1288-1291. [doi:10.1109/29.31278]

Freescale Semiconductor, 2005. MPC7450 RISC Microprocessor Family Reference Manual [online]. Available from http://www.freescale.com/files/32bit/doc/ref_manual/MPC7450UM.pdf [Rev.5].

Freescale Semiconductor, 2007. MPC7450 RISC Microprocessor Family Software Optimization Guide [online]. Available from http://www.freescale.com/files/32bit/doc/app_note/AN2203.pdf [Rev.2].

Jana, P.K., Sinha, K., 2008. Permutation algorithms on optical multi-trees. *Comput. Math. Appl.*, **56**(10):2656-2665. [doi:10.1016/j.camwa.2008.03.060]

Lloyd, B., Boyd, C., Govindaraju, N.K., 2008. Fast Computation of General Fourier Transforms on GPUs. IEEE Int. Conf. on Multimedia and Expo, p.5-8. [doi:10.1109/ICME.2008.4607357]

Lokhmotov, A., Mycroft, A., 2007. Optimal Bit-reversal Using Vector Permutations. Proc. 19th Annual ACM Symp. on Parallel Algorithms and Architectures, p.198-199. [doi:10.1145/1248377.1248411]

Marti-Puig, P., 2009. Two families of radix-2 FFT algorithms with ordered input and output data. *IEEE Signal Process. Lett.*, **16**(2):65-68. [doi:10.1109/LSP.2008.2003993]

Pei, S.C., Chang, K.W., 2007. Efficient bit and digital reversal algorithm using vector calculation. *IEEE Trans. Signal*

Process., **55**(3):1173-1175. [doi:10.1109/TSP.2006.887567]
Püschel, M., Milder, P.A., Hoe, J.C., 2009. Permuting streaming data using RAMs. *J. ACM*, **56**(2):Article No. 10, p.1-34. [doi:10.1145/1502793.1502799]
Sundararajan, D., Ahmad, M.O., Swamy, M.N.S., 1994. A fast FFT bit-reversal algorithm. *IEEE Trans. Circuits Syst. II: Anal. Dig. Signal Process.*, **41**(10):701-703. [doi:10.1109/82.329741]
Walker, J., 1990. A new bit-reversal algorithm. *IEEE Trans. Acoust. Speech Signal Process.*, **38**(8):1472-1473. [doi:10.1109/29.57586]
Yong, A.A., 1991. A better FFT bit-reversal algorithm without tables. *IEEE Trans. Signal Process.*, **39**(10):2365-2367. [doi:10.1109/78.91199]

APPENDIX

A C language routine of the bit-reversal algorithm wzk-Yong-Lokhmoto is presented as follows:

```
/* in_r and in_i are the first addresses of the real and imaginary
   arrays, respectively */
int wzk_Yong_Lokhmotov(float *in_r, float *in_i, int n)
{
  unsigned int base, k, j, i, v_n;
  /* countr is used to count the number of branches */
  unsigned int countr=0;
  unsigned int number_1_4, number_2_4, number_3_4;
  vector float vr_0_4, vr_1_4, vr_2_4, vr_3_4;
  vector float vi_0_4, vi_1_4, vi_2_4, vi_3_4;
  vector float vtemp1, vtemp2, vtemp3, vtemp4;
  vector float *vin_r, *vin_i, *vin_r_pr, *vin_i_pr;
  vin_r=(vector float *)in_r;
  vin_i=(vector float *)in_i;
  v_n=n/4;
  number_2_4=v_n<<3;
  number_1_4=v_n<<2;
  number_3_4=number_1_4+number_2_4;
  base=v_n>>2;  k=base;  j=0;
  for (i=0; i<(number_1_4>>4); i++)
  {
    vin_r_pr=i+vin_r;
    vin_i_pr=(j>>1)+vin_i;
    /* count++, one branch due to for-statement */
    k=base;
    while (k<=j)
    {  j=j−k;  k=k>>1;
    /* count++, one branch due to while-statement */ }
    j=j+k;     /* j is the index of the imaginary part */
    vr_0_4=vec_ld(0, vin_r_pr);
    vi_0_4=vec_ld(0, vin_i_pr);
    vr_2_4=vec_ld(number_2_4, vin_r_pr);
    vi_2_4=vec_ld(number_2_4, vin_i_pr);
    vr_1_4=vec_ld(number_1_4, vin_r_pr);
    vi_1_4=vec_ld(number_1_4, vin_i_pr);
    vr_3_4=vec_ld(number_3_4, vin_r_pr);
    vi_3_4=vec_ld(number_3_4, vin_i_pr);
    vtemp1=vec_mergeh(vr_0_4, vr_1_4);
    vtemp2=vec_mergel(vr_0_4, vr_1_4);
    vtemp3=vec_mergeh(vr_2_4, vr_3_4);
    vtemp4=vec_mergel(vr_2_4, vr_3_4);
    vr_0_4=vec_mergeh(vtemp1, vtemp3);
    vr_1_4=vec_mergeh(vtemp2, vtemp4);
    vr_2_4=vec_mergel(vtemp1, vtemp3);
    vr_3_4=vec_mergel(vtemp2, vtemp4);
    vec_st(vr_0_4, 0, vin_i_pr);
    vec_st(vr_2_4, number_2_4, vin_i_pr);
    vec_st(vr_1_4, number_1_4, vin_i_pr);
    vec_st(vr_3_4, number_3_4, vin_i_pr);
    vtemp1=vec_mergeh(vi_0_4, vi_1_4);
    vtemp2= ec_mergel(vi_0_4, vi_1_4);
    vtemp3=vec_mergeh(vi_2_4, vi_3_4);
    vtemp4=vec_mergel(vi_2_4, vi_3_4);
    vi_0_4=vec_mergeh(vtemp1, vtemp3);
    vi_1_4=vec_mergeh(vtemp2, vtemp4);
    vi_2_4=vec_mergel(vtemp1, vtemp3);
    vi_3_4=vec_mergel(vtemp2, vtemp4);
    vec_st(vi_0_4, 0, vin_r_pr);
    vec_st(vi_2_4, number_2_4, vin_r_pr);
    vec_st(vi_1_4, number_1_4, vin_r_pr);
    vec_st(vi_3_4, number_3_4, vin_r_pr);
  }
  return 0;
}
```