Split-Radix FFT Algorithms Based on Ternary Tree

Ming Zhang

School of Computer Science and Technology, University of Science and Technology of China,

Hefei 230027, China

ustczm@126.com

Abstract

Fast Fourier Transform (FFT) is widely used in signal processing applications. For a 2n-point FFT, split-radix FFT costs less mathematical operations than many state-of-the-art algorithms. Most split-radix FFT algorithms are implemented in a recursive way which brings much extra overhead of systems. In this paper, we propose an algorithm of split-radix FFT that can eliminate the system overhead. Ternary tree schedule algorithms for split-radix FFT will be introduced. Additionally, we use some optimizing technologies such as loop unrolling, data prefetching and instruction pipeline adjustment to enhance performance. For large size split-radix FFT, we propose an efficient algorithm that uses "BFS+BFS" traversal model in the ternary tree to reduce the overhead of memory access. In the end, experimental results on Godson-3A2000 show that the algorithm in an iterative way performs 20% better than the algorithm in a recursive way. Compared with standard library FFTW3, the performance of the optimized algorithm in an iterative is promoted by 30%.

Keywords

Split-Radix FFT algorithm, Signal Processing, Godson-3A2000.

1. Introduction

Fast Fourier Transform (FFT), proposed by Cooley and Turkey in 1965 [1], is the fundamental algorithm in digital image processing which greatly enhances the performance of Discrete Fourier Transform (DFT). It reduces the number of mathematical operations, making a breakthrough in the research on image processing. At present, it is widely used in image filtering, noise reduction and image compression. According to the characteristics of twiddle factors, FFT uses the buttery structure and recursively expresses a DFT of length N in terms of two smaller DFTs of length $N/2$, reducing many operations. For a size- N transformation, the mathematical operations complexity of DFT is $O(N^2)$ while the complexity of FFT is only $O(N\log_2 N)$. For example, for the Radix-2 algorithm, which is the most common FFT algorithm, its specific calculation amount is $5N\log_2 N$.

In 1968, Yavne [2] presented what became known the base of split-radix FFT algorithm, and he gave the record op count of $4N\log_2 N-6N+8$. In 1984, Duhamel et al. [3] Proposed split-radix FFT algorithm which was a variant of the Cooley-Turkey algorithm. Split-radix FFT uses a blend method of radix-2 and radix-4. By using the divide and conquer methodology, split-radix FFT recursively decomposes an N -point DFT into one $N/2$ -point DFT and two $N/4$ -point DFTs. The mathematical number of operations of split-radix FFT is $4N\log_2 N$, which almost 20% is less than radix-2 algorithm.

Different from Radix-x algorithms, split-radix FFT divides the original DFT into three sub-DFTs, then recursively divides the sub-DFTs. The implementation of a split-radix FFT algorithm in a recursive way is very easily achieved. Due to the extra overhead of recursive function calls, the iterative form will perform much better when compared with recursive form. After repeated partitions, there will be many small DFTs and intermediate processes to be calculated. It seems difficult to compute the calculation methods and twiddle factors. Considering the similarity of split-radix FFT algorithm and ternary tree, this paper combines them together, and proposes a new implementation.

In this implementation, we first build the ternary tree by pre-calculating the partitions. We calculate the sub-DFT from root to leaves, and record the information for FFT in the corresponding node according to the ternary tree.

With recursive partitions processing, there will be many leaves that only have a few calculations, such as DFT4 and DFT2. The number of operations in DFT4 and DFT2 are so small that calculation ability of hardware could not be fully used. To get higher performance, this paper changes the computation unit from 2 & 4 to 4 & 8. This solution will greatly reduce the number of leaves that have a few operations. In the implementation, when the size of the sub-DFT reduces to the size of the computing unit, we will stop dividing the sub-DFT. simultaneously, we optimize the calculation of computational unit with Single Instruction Multiple Data (SIMD) instructions. Moreover, we fully incorporate the features of microstructure, such as instruction pipeline and the multi-emission mechanism of hardware to finish the code level optimization.

The remainder of this paper is organized as follows. In Section 2, the related work is given. Section 3 introduces the background. The proposed TTSSR is presented in Section 4. In Section 5, optimizations for split-radix FFT based on hardware features will be put forward. In Section 6, experimental results are presented, and finally Section 7 presents the conclusion.

2. Related Work

In order to improve the performance of split-radix FFT, much attention has been devoted to searching for new algorithms. So far, much work on reducing the mathematical operations complexity has been completed. In 2004, Van Buskirk [4] became the first person to contribute to improving the number of mathematical operations and he managed to save eight operations over Yavne [2] by hand optimization for 64-point split-radix FFT. Later, Lundy and Van Buskirk developed an automatic code-generation implementation that achieved almost $\frac{2}{3}N\log_2 N-\frac{38}{27}N$ fewer operations than Yavne [5], given an arbitrary fixed $N=2^r$. In light of Van Buskirk's [4] code-generation framework, Johnson et al. [6] presented a simple recursive modification of the split-radix algorithm that computed the DFT with asymptotically about 6% fewer operations than Yavne in 2007.

Moreover, Bouguezel et al. [7] presented an efficient split-radix FFT algorithm for computing the 2^r-point DFT. The new algorithm substantially reduced the operations such as data transfer, address generation, twiddle factor evaluation, or accessing the lookup table, which had contributed significantly to the execution time of FFT algorithms. Bouguezel et al. [7] made adjustments so that radix-2 and radix-8 index maps were used instead of radix-2 and radix-4 index maps, as in the classical split-radix FFT to prove the improved algorithm. Later, Takahashi [8] extended split-radix FFT algorithm, which had the same asymptotic arithmetic complexity. This algorithm costs fewer load and store instructions than the conventional split-radix FFT algorithm.

In order to obtain efficient performance on various platforms, Ocovaj et al. [9] presented an implementation of a conjugate-pair variant of split-radix FFT that was optimized for platforms with an SIMD instruction set extension. Karishma et al. [10] exploited the lower arithmetic obscurity of split-radix FFT to lower dynamic energy by gating the multipliers during trivial multiplication. In this paper, Watanabe et al. [11] proposed a new architecture to compute the RFFT on Field Programmable Gate Array (FPGA). Bouguezel et al. [12] proposed a new radix-2/4 FFT algorithm for computing the discrete Fourier transform of an arbitrary $q\times 2^m$ -point, where q is an odd integer. Ma et al. [13] proposed a new address scheme for efficiently implementing mixed radix FFTs, and they designed an elaborate accumulator that could generate access addresses for the operands, as well as the twiddle factors.

In the literature, people have only analyzed the algorithm itself and optimized the algorithm from mathematical operation numbers. However, they have not considered the implementation of algorithm and make little optimization on concrete hardware.

3. Backgrounds

Godson is a family of general-purpose CPUs developed at the Institute of Computing Technology, Chinese Academy of Sciences in China. Godson-3A2000 [14] is the latest product which is built on GS464E architecture. It is a general-purpose RISC quad-core CPU whose instruction set is a self-developed MIPS64 instruction set. Compared to the x86 instruction set, the structure of the MIP64S instruction set is more simple and efficient. Thus, the MIPS architecture is more suitable for scientific computing. More, its floating point capabilities are very powerful. However, it is significantly different from the x86-based CPU in architecture. Various softwares (e.g., ATLAS, FFTW) that use self-turning technology for optimization cannot get ideal performance.

Godson-3A2000 is a 4-way superscalar processors built on a 9-stage, super-pipelined architecture. It has the same characters of MIPS. It is configured with out-of-order execution units, two floating-point operation units, a memory management unit and an innovative crossbar interconnect. Unlike the random cache replacement policy for the Godson series, the Godson-3A2000 uses the LRU cache replacement for data and instruction cache.

4. Implementation of TTSSR

The split-radix FFT algorithm divides a large point DFT into smaller-point DFTs recursively. In order to intuitively represent the recursive partitions, the proposed ternary tree for split-radix FFT will be described in this section. First, we briefly introduce the basic algorithm of split-radix FFT. Then we propose the TTSSR in detail, and introduce several algorithms for building the ternary tree, traversing the ternary tree in BFS, and using the "BFS+BFS" model to optimize.

4.1 Split-Radix Basic Algorithm

For an N-point array $x_0, x_1, x_2, \cdots, x_{N-1}$, the DFT of x is defined in (1).

$$y_k = \sum_{n=0}^{N-1} W_N^{nk} x_n, \quad k = 0, \cdots, N-1 \tag{1}$$

Where $W_N (e^{-2ij\pi/N})$ is the primitive root-of-unity and $j = \sqrt{-1}$. For N that can be divided by four, we perform a decimation-in-time decomposition of x_n into three smaller DFTs. They are respectively x_{2k} (the even elements), x_{4k+1}, and x_{4k+3}. According to the periodicity of W_N, we obtain the derivations from (1), that are shown in (2).

$$\begin{aligned} X_{2k} &= \sum_{n=0}^{N/2-1} (x_n + x_{n+N/2}) W_N^{2nk} \\ X_{4k+1} &= \sum_{n=0}^{N/4-1} \left[(x_n - x_{n+N/2}) - j(x_{n+N/4} - x_{n+3N/4}) \right] W_N^{n} W_N^{4nk} \\ X_{4k+3} &= \sum_{n=0}^{N/4-1} \left[(x_n - x_{n+N/2}) + j(x_{n+N/4} - x_{n+3N/4}) \right] W_N^{3n} W_N^{4nk} \end{aligned} \tag{2}$$

The first stage of split-radix FFT is decimation-in-frequency decomposition. After the decomposition, N-point DFT changes to a coalition of three small DFTs. We can then calculate the small DFTs independently. For each small DFT, we can recursively call the split-radix FFT algorithm to solve it. If the DFT is small enough, that is, if DFT size equals the size of computational unit, we need only call the unit calculation function for the rest calculation, without any partition. In total, split-radix FFT algorithm takes advantage of the character of DFT, recursively callings the split-radix algorithm until the size of DFT reaches the size of computational units. Fig. 1 shows the procedure for a 16-point DFT that uses a split-radix FFT algorithm. The "L-shape" in the Fig.1 represents the partition level of the DFT with the split-radix FFT algorithm.

Algorithm 1: Standard Recursive Split-Radix 2/4 FFT Algorithm of length N.

Input: x, N
Output: x
function $x_{k=0\cdots N-1} \leftarrow \boldsymbol{SplitRadix}_N(x_{k=0\cdots N-1})$;
for $i = 0; i < N/2; i++$ **do**
  $x_i \leftarrow x_i + x_{i+N/2}$;
for $i = 0; i < N/4; i++$ **do**
  $x_{i+N/2} \leftarrow [x_i - x_{i+N/2} - j(x_{n+N/4} - x_{n+3N/4})]\omega^{iN_{org}/N}$;
  $x_{i+3N/4} \leftarrow [x_i - x_{i+N/2} + j(x_{n+N/4} - x_{n+3N/4})]\omega^{3iN_{org}/N}$;
$SplitRadix_{N/2}(x_{k=0\cdots N/2-1})$;
$SplitRadix_{N/4}(x_{k=N/2\cdots 3N/4-1})$;
$SplitRadix_{N/4}(x_{k=3N/4\cdots N-1})$;

The basic recursive split-radix FFT algorithm is shown as Algorithm 1. In the basic algorithm, the split-radix function recursively invokes itself until all sub-DFTs finish calculating. For large scale DFT, this algorithm takes much system time to recursively invoke itself.

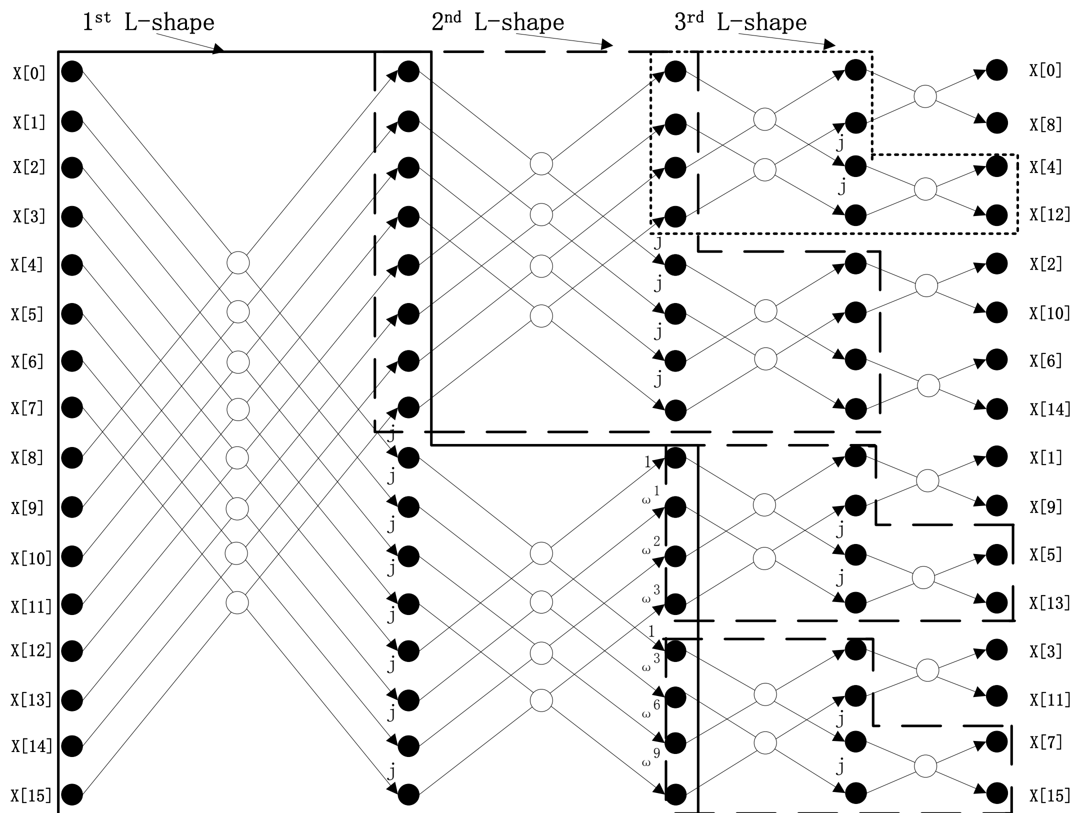

Fig. 1 16-point split-radix FFT algorithms

4.2 TTSSR

In the ternary tree representation, each node represents one DFT. Data structure spnode is assigned to each node, as shown in Listing 1. spnode. n represents the size of the DFT to be calculated. spnode. pos represents the beginning address to be calculated in the spnode. Additionally, spnode. posOmega represents the position of the twiddle factor in the twiddle factor array. The spnode. father points to the father node of the spnode. Similarly, spnode. left, spnode.mid and spnode. right point to its three children nodes. When the tree structure is finalized, all the leaves correspond to computational elements, and internal nodes represent the calculations for partition that are required to update the data for sub-DFTs.

Listing 1: Data Structure of Node

```
typedef struct spnode
{
    int n;
    int pos;
    int posOmega;
    struct spnode *left, *mid, *right, *father;
}spnode;
```

For example, in a 2^m-point $(m > 2)$ split-radix FFT, the FFT is divided into one 2^{m-1}-point FFT and two 2^{m-2}-point FFTs. In implementation, we use p to represent the root of a ternary tree. Meanwhile, nodes i, j and k, respectively represent the left child, middle child and right child. p. n equals to 2^m and i.n equals to 2^{m-1}. Similarly, j.n and k.n set to 2^{m-2}. p.pos, i.pos, j.pos and k.pos are respectively assigned to 0, 0, $n/2$, $3n/4$. Then, p.left, p.mid, p.right will be set to i, j and k. i.father, j. father and k. father will be set to p. As shown in Fig. 2, p. pos Omega, i. pos Omega, j.pos Omega and k. posOmega are assigned to 1, 2, 4 and 4.

Building Ternary Tree for split-radix FFT

As noted above, the definition of the ternary tree for split-radix FFT decides the procedure to build the tree. The algorithm for building the ternary tree for split-radix FFT is presented in Algorithm 2. This algorithm is implemented in a recursive way. First, the root node is initialized and information for calculation is stored in the data structure. Then, the roots of the sub-trees are initialized, and recursively invoke this algorithm to build the sub-trees. Finally, we update the root's information for the children trees and father information for the sub-trees. If the partition size reaches 4 or 8, we stop dividing it and quit. According to the Build Tree algorithm, a ternary tree can be constructed for the given transformation length. During the procedure of the establishment of a ternary tree, the variables root. Post Omega for each sub-DFTs are calculated and stored at the respective positions in memory. Fig. 2 illustrates a ternary tree for 64-point split-radix FFT. For the same point split-radix FFT, the ternary tree is the same. In order to enhance the performance, we can pre-build the ternary tree for split-radix FFT.

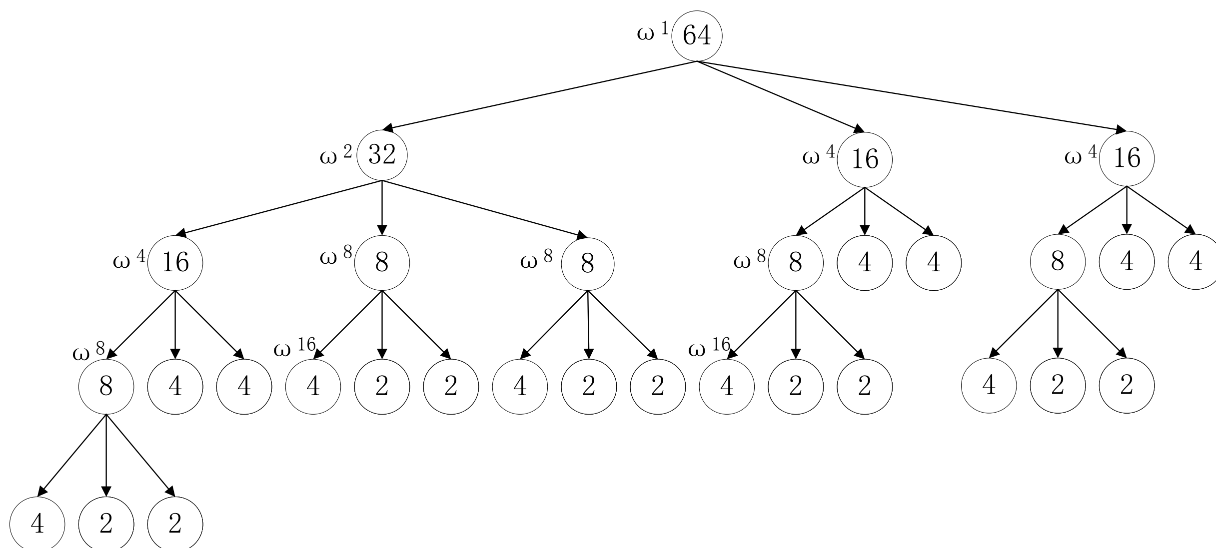

Fig. 2 Ternary Tree for 64-point split-radix FFT Algorithm

Split-Radix Algorithm Based on Ternary Tree

After the establishment of the ternary tree, the procedure for the split-radix FFT is straightforward. In split-radix FFT, we need to traverse all the nodes of the ternary tree. Depending on the information in the node, we finish all the calculations presented by the variables. When all the nodes have been

visited and all calculations finished, the split-radix FFT ends. For the ternary tree, there are two kinds of traversal methods DFS and BFS. DFS corresponds to recursively split-radix FFT while BFS corresponds to iteratively split-radix FFT. In order to obtain better performance, this paper will use the BFS algorithm as the traversing algorithm. In BFS, we save traversal paths with a queue that has the characteristics of first in first out (FIFO) and visit the sub-tree from left to right. First, we initialize one queue and put the root node of the ternary tree into the queue. Then, we recursively visit the head of the queue, putting the child of visited node into the queue, until the queue becomes empty. When one node is visited, we should finish the calculations presented by the node. Generally speaking, the BFS algorithm visits the ternary tree by levels, and the split-radix FFT runs in an iterative way. The concrete realization method is shown in Algorithm 3.

Algorithm 2: Build Ternary Tree for N-point Split-Radix FFT

Input: $Root,\ N,\ pos,\ pos_\omega$
Output: $Root$
function $BuildTree(spnode\ Root, int\ N, int\ pos, int\ pos_\omega)$;
if $N == 8$ **then**
  $InitNode(node, 8)$;
  return node;
else if $N == 4$ **then**
  $InitNode(node, 4)$;
  return node;
else
  $InitNode(subtree_{left}/subtree_{mid}/subtree_{right})$;
  $BuildTree(subtree_{left},\ N/2,\ pos,\ 2pos_\omega)$;
  $BuildTree(subtree_{mid},\ N/4,\ pos + N/2,\ 4pos_\omega)$;
  $BuildTree(subtree_{right},\ N,\ pos + 3N/4,\ 4pos_\omega)$;
  $Root \rightarrow left = subtree_{left}$;
  $Root \rightarrow middle = subtree_{mid}$;
  $Root \rightarrow left = subtree_{left}$;
  $subtree_{left} \rightarrow father = Root$;
  $subtree_{mid} \rightarrow father = Root$;
  $subtree_{right} \rightarrow father = Root$;

Blocked Split-radix Algorithm

Traversing the ternary tree by level means that all the data will be accessed in each level. If the size of split-radix exceeds the cache size, the cache content will always be replaced in each level. Because of the cache miss, simple BFS model will cause much time of memory access. In accordance with the characteristics of ternary trees, all the sub-trees of one node are independent and the calculation data presented by the each node are consequent. It will greatly reduce memory access to independently finish the calculations presented by nodes in a sub-tree of the node. After processing one sub-tree, we continue to deal with the other sub-trees. In the optimized algorithm, memory access of one block's data are performed many times. Now, this approach accesses the block data only once. Correspondingly, this independent computing in the ternary tree is a blocked algorithm for split-radix FFT.

Considering the relationship between split-radix FFT algorithm and the parameters of microstructure, it is necessary to discuss its quantity of memory access. For k-point split-radix FFT, k twiddle factors and k source complex numbers will be accessed. So as to minimize the rate of cache misses, we have to ensure that there are enough places for all data to be used in the calculation in cache. In other words, (3) must be satisfied. On the other hand, in order to obtain a higher ratio of calculation to memory access, we should maximize the k .

Algorithm 3: Split-Radix Algorithm Based on BFS

Input: *Root*
Output: N, pos, pos_{ω}, $pos_{\omega 3}$
function $BFStreesrfft(Root)$
InitQueue(Qtree);
EnQueue(Qtree, Root);
while *Empty(Qtree)!=True* **do**
    node = OutQueue(Qtree);
    $Visit(node, \&node.n, \&node.pos, \&node.pos_{\omega}, \&node.pos_{\omega 3})$;
    for $i = 0; i < node.n/4; i++$ **do**
        $x_i \leftarrow x_i + x_{i+node.n/2}$;
        $x_{i+node.n/4} \leftarrow x_{i+node.n/2} + x_{i+3node.n/4}$;
        $x_{i+node.n/2} \leftarrow [x_i - x_{i+node.n/2} - j(x_{n+node.n/4} - x_{n+3node.n/4})] \cdot \omega[i * node.pos_{\omega} \% N_{orig}]$;
        $x_{i+3node.n/4} \leftarrow [x_i - x_{i+node.n/2} + j(x_{n+node.n/4} - x_{n+3node.n/4})] \cdot \omega[i * node.pos_{\omega 3} \% N_{orig}]$;
    if $node \rightarrow left! = NULL$ **then**
        EnQueue(Qtree, $node \rightarrow left$);
        EnQueue(Qtree, $node \rightarrow mid$);
        EnQueue(Qtree, $node \rightarrow right$);

According to the size of cache, we could easily calculate the appropriate k. When k is selected, we can use "BFS+BFS" traversal method to reduce the time of memory access. In the ternary tree, the sub-tree of one node with $Node.n = k$ presents one k-point split-radix FFT. We utilize BFS algorithm to traverse the ternary tree. When one node with $Node.n = k$ is visited, we stop the outer BFS and directly use BFS algorithm to traverse all sub-trees of $Node$ and finish the calculation. Meanwhile, the offspring nodes of $Node$ will not be visited anymore in the outer BFS. In short, "BFS+BFS" model will be chosen to improve our algorithm. The blocked split-radix FFT algorithm based on ternary tree is shown in Algorithm 4.

$$k * sizof(source\ \ element) + k * sizeof(twiddle\ \ factors) \le sizeof(cache\ \ size) \quad (3)$$

5. Optimization for Spit-Radix FFT

In this section, we will make use of hardware features of Godson-3A2000 to optimize the performance of split-radix FFT.

In the kernel of split-radix FFT, most additions (or subtractions) need the results of multiplications, and they have read-after-write relationships. Addition instructions need to wait several cycles for result to return, and this causes some pipeline idleness. Only by inserting additional instructions, can idle pipelines be reduced. With the floating function units supporting multi-add instructions, addition and multiplication can be combined. This method can reduce the number of instruction, which can reduce the overhead of the registers.

Instruction scheduling is one technology to reduce the idle pipeline by adjusting the instruction pipeline ordering. There are two floating point arithmetic units and two memory access units in the Godson-3A2000. Two load instructions and two floating multi-add instructions will launch in each cycle. If there are no fixed instructions, the instruction position will be replaced by "nop" instruction to avoid conflicts between pipelines. The Godson-3A2000 provides a 128-bit memory access instruction. It can be used to simultaneously access two double floating numbers between registers and memory. In split-radix FFT, the imaginary part of complex numbers stored next to the real part. Also, the complex number is 16 byte aligned. With one 128-bit memory access instruction, we could access one complex number, reducing the number of memory access instructions.

Algorithm 4: Blocked split-radix FFT Algorithm Based on BFS

Input: $Root$, k_0,k_1
Output: N, pos, pos_ω, $pos_{\omega 3}$
function $Blockedsrfft(Root)$
call InitQueue(Qtree);
call EnQueue(Qtree, Root);
while *Empty(Qtree)!=True* **do**
  node = OutQueue(Qtree);
  $Visit(node, \&node.n, \&node.pos, \&node.pos_\omega, \&node.pos_{\omega 3})$;
  if $((node.n == k_0) \mid\mid (node.n == k_1))$ **then**
    call $BFStreesrfft(node)$;
  else
    for $i = 0; i < node.n/4; i++$ **do**
      $x_i \leftarrow x_i + x_{i+node.n/2}$;
      $x_{i+node.n/4} \leftarrow x_{i+node.n/2} + x_{i+3node.n/4}$;
      $x_{i+node.n/2} \leftarrow [x_i - x_{i+node.n/2} - j(x_{n+node.n/4} - x_{n+3node.n/4})] \cdot \omega[i * node.pos_\omega \% N_{orig}]$;
      $x_{i+3node.n/4} \leftarrow [x_i - x_{i+node.n/2} + j(x_{n+node.n/4} - x_{n+3node.n/4})] \cdot \omega[i * node.pos_{\omega 3} \% N_{orig}]$;
    if $node \rightarrow left! = NULL$ **then**
      EnQueue(Qtree, $node \rightarrow left$);
      EnQueue(Qtree, $node \rightarrow mid$);
      EnQueue(Qtree, $node \rightarrow right$);

Generally, in split-radix FFT, the larger the computational unit, the fewer the stages will be required for the calculation process. However, if the computational unit is too large, the registers required will exceed the supply by platform. Considering the performance and register, we chose 8 and 4 as the computational unit in godson-3A2000. According to Formulary 3, with 64KB L1cache, we can compute 2048-points in L1cache without cache misses.

When dealing with computational unit, several numbers will be multiplied by the imaginary number j. For reducing the number of multiplications, we can directly exchange the real part with imaginary part and make the real part negative. Considering the common denominator of twiddle factors in 8-point computational unit, all twiddle factors numbers could be expressed by $\sqrt{2}/2$. To reduce the amount of memory being accessed for twiddle factors, we accessed one number $\sqrt{2}/2$ instead of 8 complex factors. Moreover, we modified the execute instructions to satisfy the calculations for 8-point. In addition, we manually used *MIPS64* assembly language to write kernel functions for computational units.

6. Experimental Results

The experimental results were performed on the Godson-3A2000 which is clocked at 800Mz. The platform is configured with two 2GB dual-channel DDR3-1000 memory chips. The operating system is Loongson system with version "Linux 3.10.84". The GCC compiler with optimization level-O2 was used for all cases.

The FFT are tested whose sizes are powers of 2 and arrange from 2^7 to 2^{17}. The input array and the twiddle factors array are double complex numbers. The original iterative and recursive algorithms are implemented in the C language. The optimized iterative algorithms are implemented in C and MIPS64 assembly language. FFTW3 is used as the performance reference, and FFTW EXHAUSTIVE flag and wisdom mechanism were used. FFTW3 was executed once with the FFTW EXHAUSTIVE flag to discover the optimal plan. Then the wisdom mechanism saved the plan into memory for reloading many times.

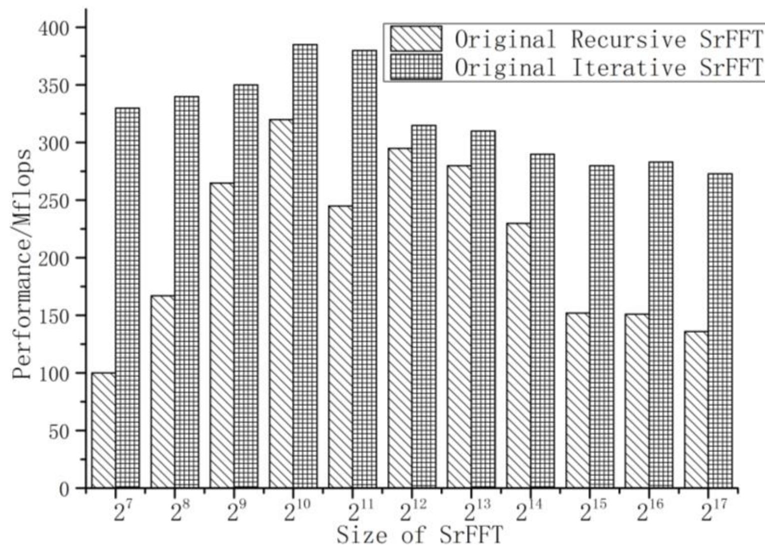

Fig. 3 Performance of Original Algorithms

Performance is compared between the original iterative and the recursive algorithms. As shown in Fig. 3, the iterative algorithm performs better than the recursive algorithm. When FFT size exceeds 2^{10}, the performance drops markedly. For the recursive algorithm, because of the extra system overhead, its performance drops 100 MFLOPS between 2^{10}-points to 2^{11}-points. With the size increasing, the overhead of memory access becomes the main limiting factor and performance decreases.

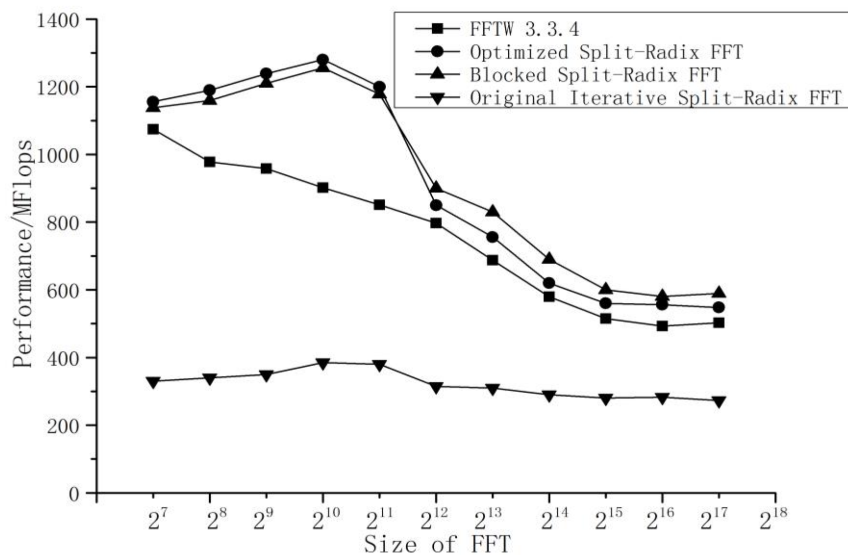

Fig. 4 Comparison between four algorithms

As shown in Fig. 4, optimized split-radix FFT algorithm performs better than the original algorithm. Unlike FFTW, its performance increases from 2^7-points to 2^{10}-points. When the size arranges increases 2^{11} to 2^{12}, there is a large performance decrease. This occurs when the data size is larger than the data cache. When data size is larger than the data cache, cache access tremendously increases, and performance rapidly degraded. The best performance by the optimized algorithm was 1285 MFLOPS which occurs at 2^{10} points. The performance of the split-radix FFT algorithm is improved by 20% compared with FFTW3.3.4.

Compared with the optimized algorithm, the blocked split-radix FFT algorithm uses blocking technology to enhance the reusability of data in cache. When the size of split-radix FFT ranges from 2^{12} to 2^{17}, its performance increases 95 MFLOPS better over the optimized algorithm. Because the judgment condition statements are added, the performance of FFT for small sizes drops about 20

MFLOPS. On the whole, blocked split-radix FFT algorithm brings 30% performance improvement over FFTW3.3.4.

7. Conclusion

In this paper, a new split-radix FFT methodology that achieves superior performance is presented. This methodology combines split-radix and ternary tree procedures. By utilizing the BFS method, we traverse the ternary tree and record the calculation for split-radix FFT. Moreover, we exploit optimizing the split-radix FFT algorithm on a Godson-3A2000 CPU. Experimental results show that the new algorithm surpasses FFTW by 30%.

Future work includes an analysis and implementation of parallel split-radix FFT algorithms on a multi-core platform. Moreover, we will optimize the parallel split-radix FFT performance by taking advantage of the features of the ternary tree.

References

[1] J.W. Cooley, J.W. Tukey: An algorithm for machine computation of complex Fourier series, Math. Comput., Vol . 19 (1965) No.9, p.297-301.

[2] Yavne R.: An economical method for calculating the discrete Fourier transform. the ACM fall joint computer conference (Dec 9-11, 1968), p.115-125.

[3] P. Duhamel, H. Hollman: Split-radix FFT algorithms, Electron. Letters, Vol. 20 (1984), p.14-16.

[4] J. Van Buskirk: comp.dsp Usenet posts Jan. 2004.

[5] Lundy T, Buskirk J V.: A new matrix approach to real FFTs and convolutions of length 2^k, Computing, Vol. 80 (2007), No.1, p.23-45.

[6] Johnson, S.G.; Frigo, M.: A Modified Split-Radix FFT With Fewer Arithmetic Operations, IEEE Transactions on signal processing, Vol. 55 (2007), No.1, p.111-119.

[7] Saad Bouguezel, M. Omair Ahmad, M. N. S. Swamy: An efficient split-radix FFT algorithm, the 2003 International Symposium on Circuits and Systems (2003), Vol. 4, p.65-68.

[8] Takahashi, D.: An extended split-radix FFT algorithm, IEEE Signal Processing Letters, Vol. 8 (2001), No.5, p.145-147.

[9] Ocovaj S, Lukac Z.: Optimization of conjugate-pair split-radix FFT algorithm for SIMD platforms, 2014 IEEE International Conference on Consumer Electronics (2014), p.373-374.

[10] Karishma A. Deshmukh, P. R. Indurkar, D. M. Khatri: High performance split radix FFT, International Journal of Innovative Research in Advanced Engineering (IJIRAE), Vol. 1 (2014) , No.6, p.380-386.

[11] Watanabe, C., Silva, C., Muñoz, J.: Implementation of Split-Radix Fast Fourier Transform on FPGA, Programmable Logic Conference (24-26 March, 2010), p.167-170.

[12] Saad Bouguezel, M. Omair Ahmad: A New Radix-$2/8$ FFT Algorithm for Length-$q \times 2^m$, IEEE Transactions on circuits and systems, Vol. 51 (2004), No. 9, p.1723-1732.

[13] Cuimei Ma, Yizhuang Xie, He Chen, Yi Deng, Wen Yan: Simplified addressing scheme for mixed radix FFT algorithms, IEEE International Conference on Acoustics, Speech and Signal Processing (4-9 May, 2014), p.8355-8359.

[14] Ruiyang Wu, Wenxiang Wang, Huandong Wang: Godson processor core architecture GS464E, Scientia Sinica Informationis, (2015), No.4, p.480-500.