

DSP Programming Homework – Fast Fourier Transform

W42 林子恒 2014011054

2014.10.24

目录

1	Problem	2
2	Solution	2
2.1	Realization of Discrete Fourier Transform & Fast Fourier Transform	2
2.1.1	Realization of Discrete Fourier Transform (dft.h)	2
2.1.2	Realization of Decimate-in-Time Fast Fourier Transform (dit_fft.h)	3
2.1.3	Realization of Decimate-in-Frequency Fast Fourier Transform (dif_fft.h)	5
2.2	Some Other Supporting Functions	7
2.2.1	Struct of Complex (complex.h)	7
2.2.2	Calculation of W_N^k (complex.h - Calc_WN())	8
2.2.3	Convert to Bit-reversed order (complex.h - reverse_bit())	8
2.2.4	Validate & Evaluate Function (validate_n_evaluate.h)	9
2.2.5	Main Function (fft.cpp)	9
2.3	Validation of Correctness	10
2.4	Performance Comparision of Algorithms	11

1 Problem

编写任意 2 的证书次幂点数的基 2 DIT-FFT 和基 2 DIF-FFT 通用 c/c++ 程序, 并与直接计算 DFT 比较点数 2^N ($N=10, \dots, 16$) 时运行时间的差异。

2 Solution

2.1 Realization of Discrete Fourier Transform & Fast Fourier Transform

2.1.1 Realization of Discrete Fourier Transform (dft.h)

Nothing special, calculated following the definition of DFT.

Note that for this DFT method and the other FFT methods in this article later, we would first calculated all of the W_N^k and store them in an array to save useless repeat computing time.

```
1 // DFT
2 // input_seq[]:
3 // N: size of input_seq
4 complex* DFT(complex input_seq[], int N) {
5
6     // Calc WN
7     complex* WN = new complex[N];
8     WN = Calc_WN(N);
9
10    complex* DFTed_seq = new complex[N];
11    for (int i = 0; i < N; ++i)
12    {
13        for (int j = 0; j < N; ++j)
14        {
15            int k_mod = (i*j) % N;
16            DFTed_seq[i] = ComplexAdd( DFTed_seq[i], ComplexMul(
17                input_seq[j], WN[k_mod]) );
18        }
19    }
20    return DFTed_seq;
21 }
```

2.1.2 Realization of Decimate-in-Time Fast Fourier Transform (dit_fft.h)

For first step, the function `DIT_FFT_reordered()` would call `reorder_seq()` to reorder the input sequence to bit-reversed order which is required by the DIT-FFT method.

For second step, we call the function `DIT_FFT()` to recursively calculate the DFT of the sequence.

```

1  complex* DIT_FFT_reordered(complex input_seq[], int N);
2  complex* DIT_FFT(complex input_seq[], int N, complex WN[], int
    recur_time_count);
3
4  // DIT-FFT
5      // input_seq[]:
6      // N: size of input_seq
7      // Must be a 2^k integer
8  complex* DIT_FFT_reordered(complex input_seq[], int N) {
9      // Initialize
10     complex* reordered_seq = new complex[N];
11     // Calc WN
12     complex* WN = new complex[N];
13     WN = Calc_WN(N);
14     // Reorder
15     reordered_seq = reorder_seq(input_seq, N);
16     // Calc DIF-FFT
17     reordered_seq = DIT_FFT(reordered_seq, N, WN, 0);
18     return reordered_seq;
19 }
20 complex* DIT_FFT(complex input_seq[], int N, complex WN[], int
    recur_time_count) {
21     // cout << "\tDIF_FFT executed!\n"; // for validation
22     // output seq
23     complex* return_seq = new complex[N];
24     if ( N != 2 ) {
25         int k = pow(2, recur_time_count);
26         // Split input_seq into 2
27         complex* first_half_input_seq = new complex[N/2];
28         complex* second_half_input_seq = new complex[N/2];
29         for (int i = 0; i < N/2; ++i) {
30             first_half_input_seq[i] = input_seq[i];
31         }
32         for (int i = 0; i < N/2; ++i) {
33             second_half_input_seq[i] = input_seq[i+N/2];
34         }
35         // DFT
36         complex* DFTed_first_half_seq = new complex[N/2];
37         DFTed_first_half_seq = DIT_FFT(first_half_input_seq, N/2,
            WN, recur_time_count+1);
38         complex* DFTed_second_half_seq = new complex[N/2];
39         DFTed_second_half_seq = DIT_FFT(second_half_input_seq, N/2,
            WN, recur_time_count+1);
40         // Calc
41         complex* output_first_half_seq = new complex[N/2];

```

```
42     complex* output_second_half_seq = new complex[N/2];
43     for (int i = 0; i < N/2; ++i) {
44         output_first_half_seq[i] = ComplexAdd(
45             DFTed_first_half_seq[i], ComplexMul(
46                 DFTed_second_half_seq[i], WN[i*k]) ) ;
47     }
48     for (int i = 0; i < N/2; ++i) {
49         output_second_half_seq[i] = ComplexAdd(
50             DFTed_first_half_seq[i], ComplexMul( ReverseComplex
51                 (DFTed_second_half_seq[i]), WN[i*k] ) );
52     }
53     // Append [output_first_half_seq] & [output_second_half_seq]
54     return_seq = append_seq(output_first_half_seq ,
55         output_second_half_seq , N/2);
56     return return_seq;
57 } else if ( N == 2 ) { // Smallest Butterfly Unit
58     // cout << "\tDIT_FFT N==2 triggered!\n"; // for validation
59     return_seq[0] = ComplexAdd(input_seq[0], ComplexMul(
60         input_seq[1], WN[0]) );
61     return_seq[1] = ComplexAdd(input_seq[0], ComplexMul(
62         ReverseComplex(input_seq[1]), WN[0] ) );
63     return return_seq;
64 }
65 // return [return_seq] # unordered
66 return return_seq;
67 }
```

2.1.3 Realization of Decimate-in-Frequency Fast Fourier Transform (dif_fft.h)

For first step, the function `DIT_FFT_reordered()` would call the function `DIT_FFT()` to recursively calculate the DFT of the sequence.

For second step, we call `reorder_seq()` to reorder the input sequence to bit-reversed order which is required by the DIF-FFT method.

```

1  complex* DIF_FFT_reordered(complex input_seq[], int N);
2  complex* DIF_FFT(complex input_seq[], int N, complex WN[], int
    recur_time_count);
3  // DIF-FFT
4      // input_seq[]:
5      // N: size of input_seq
6      // Must be a 2^k integer
7  complex* DIF_FFT_reordered(complex input_seq[], int N) {
8      // Initialize
9      complex* reordered_seq = new complex[N];
10     // Calc WN
11     complex* WN = new complex[N];
12     WN = Calc_WN(N);
13     // Calc DIF-FFT
14     reordered_seq = DIF_FFT(input_seq, N, WN, 0);
15     // Reorder
16     reordered_seq = reorder_seq(reordered_seq, N);
17     return reordered_seq;
18 }
19 complex* DIF_FFT(complex input_seq[], int N, complex WN[], int
    recur_time_count) {
20     // cout << "\tDIF_FFT executed!\n"; // for validation
21     // output seq
22     complex* return_seq = new complex[N];
23     if ( N != 2 ) {
24         complex* first_half_seq = new complex[N/2];
25         complex* second_half_seq = new complex[N/2];
26         int k = pow(2, recur_time_count);
27         // Calc
28         for (int i = 0; i < N/2; ++i) {
29             first_half_seq[i] = ComplexAdd(input_seq[i], input_seq[
                i+N/2]);
30         }
31         for (int i = 0; i < N/2; ++i) {
32             second_half_seq[i] = ComplexMul( ComplexAdd(input_seq[i
                ], ReverseComplex(input_seq[i+N/2])), WN[i*k] );
33         }
34         // DFT
35         complex* DFTed_first_half_seq = new complex[N/2];
36         DFTed_first_half_seq = DIF_FFT(first_half_seq, N/2, WN,
            recur_time_count+1);
37         complex* DFTed_second_half_seq = new complex[N/2];
38         DFTed_second_half_seq = DIF_FFT(second_half_seq, N/2, WN,
            recur_time_count+1);
39         // Append [DFTed_first_half_seq] & [DFTed_second_half_seq]

```

```
40         return_seq = append_seq(DFTed_first_half_seq ,
41                                   DFTed_second_half_seq, N/2);
42     return return_seq;
43 } else if ( N == 2 ) { // Smallest Butterfly Unit
44     // cout << "\tDIF_FFT N==2 triggered!\n"; // for validation
45     return_seq[0] = ComplexAdd(input_seq[0], input_seq[1]);
46     return_seq[1] = ComplexMul( ComplexAdd(input_seq[0],
47                                               ReverseComplex(input_seq[1])), WN[0] );
48     return return_seq;
49 }
50 // return [return_seq] # unordered
51 return return_seq;
52 }
```

2.2 Some Other Supporting Functions

2.2.1 Struct of Complex (complex.h)

```
1 // Complex Struct
2 typedef struct Complex
3 {
4     double re;
5     double im;
6     Complex() {
7         re = 0;
8         im = 0;
9     };
10    Complex(double a, double b) {
11        re = a;
12        im = b;
13    };
14 } complex;
15 complex* append_seq(complex seq_1[], complex seq_2[], int N);
16 complex* reorder_seq(complex input_seq[], int N);
17 complex* Calc_WN(int N);
18 int reverse_bit(int value, int N);
19 // Multiplier
20 complex ComplexMul(complex c1, complex c2)
21 {
22     complex r;
23     r.re = c1.re*c2.re - c1.im*c2.im;
24     r.im = c1.re*c2.im + c1.im*c2.re;
25     return r;
26 }
27 // Adder
28 complex ComplexAdd(complex c1, complex c2)
29 {
30     complex r;
31     r.re = c1.re + c2.re;
32     r.im = c1.im + c2.im;
33     return r;
34 }
35 // -c
36 complex ReverseComplex(complex c)
37 {
38     c.re = -c.re;
39     c.im = -c.im;
40     return c;
41 }
```

2.2.2 Calculation of W_N^k (complex.h - Calc_WN())

```

1 // Calc WN[], with N = input_N
2 complex* Calc_WN(int N) {
3
4     cout << "Calculating WN[] of N=" << N << "... \t";
5     complex* WN = new complex[N];
6
7     complex WN_unit; WN_unit.re = cos(2*PI/N); WN_unit.im = -sin(2*
8         PI/N);
9     WN[0].re=1; WN[0].im=0;
10
11     for (int i = 1; i < N; ++i)
12     {
13         WN[i] = ComplexMul(WN[i-1], WN_unit);
14     }
15
16     return WN;
17 }

```

2.2.3 Convert to Bit-reversed order (complex.h - reverse_bit())

```

1 // Reverse Bit
2 // input:
3 //   a decimal num,
4 //   N-based reverse method
5 // output: a decimal num
6 int reverse_bit(int value, int N) {
7
8     int ret = 0;
9     int i = 0;
10
11     while (i < N) {
12         ret <<= 1;
13         ret |= (value>>i) & 1;
14         i++;
15     }
16
17     return ret;
18 }

```


2.2.4 Validate & Evaluate Function (validate_n_evaluate.h)

The Code is too long and less important, please see validate_n_evaluate.h

I realize two functions in this file to be called by the main() function.

```

1 void validate_result(complex input_seq[], int N, int dft_dit_dif);
2 void evaluate_performance(complex input_seq[], int N_max, int
   dft_dit_dif);

```

2.2.5 Main Function (fft.cpp)

```

1 #include "complex.h" // definition of struct complex, Calc of WN
   []
2 #include "dft.h" // DFT
3 #include "dit_fft.h" // DIT-FFT
4 #include "dif_fft.h" // DIF-FFT
5 #include "validate_n_evaluate.h"
6
7 int main(int argc, char ** argv)
8 {
9     // Get argv
10     int N_max = atoi(argv[1]); // length of input sequence
11     // input 7 to run 2^{10,11,12,13,14,15,16}
12     // input 6 to run 2^{10,11,12,13,14,15}
13     int validate_or_evaluate = atoi(argv[2]);
14     // 1 for validate, 0 for ignore
15     int dft_dit_dif = atoi(argv[3]);
16     // 1:DFT, 2:DIT, 3:DIF, 4:To compute everything for
       validation
17     // Initialize
18     // Setup input sequence
19     complex* input_seq = new complex[N_max];
20     input_seq[0] = complex(1,0);
21     // For validation of the result
22     if (validate_or_evaluate == 1) {
23         validate_result(input_seq, N_max, dft_dit_dif);
24         return 0;
25     }
26     // For compare the performance(run time) of DFT/DIT/DIF
27     else if (validate_or_evaluate == 0) {
28         evaluate_performance(input_seq, N_max, dft_dit_dif);
29         return 0;
30     }
31     // end
32     return 0;
33 }

```

2.3 Validation of Correctness

Run the following script to obtain the result

```
1 make fft
2 ./fft 8 1 4
```

1. the 1st argument 8 indicate that we would calculate an 8-point DFT
2. the 2nd argument = 1 indicate that we are using validation mode
3. the 3rd argument = 4 indicate that we are using all 3 methods

And it follows with the result:

```
1 Lin, Tzu-Heng's Work, 2014011054, W42
2 Dept. of Electronic Enigeering, Tsinghua University
3 Starting, This project is to calc DFT in Original-DFT / DIT-FFT /
  DIF-FFT...
4   For Usage, Please see 'fft.cpp'
5
6 Calculating DFT...
7   X[0] = 2 + j*0
8   X[1] = 1.70711 + j*-0.707107
9   X[2] = 1 + j*-1
10  X[3] = 0.292893 + j*-0.707107
11  X[4] = 1.33227e-15 + j*-5.35898e-08
12  X[5] = 0.292893 + j*0.707107
13  X[6] = 1 + j*1
14  X[7] = 1.70711 + j*0.707107
15 Calculating DIT-FFT...
16  X[0] = 2 + j*0
17  X[1] = 1.70711 + j*-0.707107
18  X[2] = 1 + j*-1
19  X[3] = 0.292893 + j*-0.707107
20  X[4] = 0 + j*0
21  X[5] = 0.292893 + j*0.707107
22  X[6] = 1 + j*1
23  X[7] = 1.70711 + j*0.707107
24 Calculating DIF-FFT...
25  X[0] = 2 + j*0
26  X[1] = 1.70711 + j*-0.707107
27  X[2] = 1 + j*-1
28  X[3] = 0.292893 + j*-0.707107
29  X[4] = 0 + j*0
30  X[5] = 0.292893 + j*0.707107
31  X[6] = 1 + j*1
32  X[7] = 1.70711 + j*0.707107
```

We are using a test sample $x[n] = \{1, 1, 0, 0, 0, 0, 0, 0\}$, we can validate that the result is correct.

2.4 Performance Comparision of Algorithms

Run the following script to obtain the result

```

1  make fft
2  ./fft 7 0 4 # Calc  $2^{\{10\sim 16\}}$  using all 3 methods
3  # ./fft 6 0 4 # Calc  $2^{\{10\sim 15\}}$  using all 3 methods

```

Note that:

1. the 1st argument 7 indicate that we would calculate all 7 N's from 2^{10} to 2^{16} :
2. the 2nd argument = 0 indicate that we are using performance evaluation mode
3. the 3rd argument = 4 indicate that we are using all 3 methods

And it follows with the result:

```

1  Lin, Tzu-Heng's Work, 2014011054, W42
2  Dept. of Electronic Enigeering, Tsinghua University
3
4  This project is to calc DFT in Original-DFT / DIT-FFT / DIF-FFT...
5  For Usage, Please see 'fft.cpp'
6
7  Calculating DIT-FFT...
8  N = 1024      Run time = 2 ms
9  N = 2048      Run time = 6 ms
10 N = 4096      Run time = 14 ms
11 N = 8192      Run time = 24 ms
12 N = 16384     Run time = 53 ms
13 N = 32768     Run time = 106 ms
14 N = 65536     Run time = 200 ms
15 Calculating DIF-FFT...
16 N = 1024      Run time = 1 ms
17 N = 2048      Run time = 3 ms
18 N = 4096      Run time = 7 ms
19 N = 8192      Run time = 16 ms
20 N = 16384     Run time = 39 ms
21 N = 32768     Run time = 79 ms
22 N = 65536     Run time = 154 ms
23 Calculating DFT...
24 N = 1024      Run time = 38 ms
25 N = 2048      Run time = 180 ms
26 N = 4096      Run time = 771 ms
27 N = 8192      Run time = 2948 ms
28 N = 16384     Run time = 12811 ms
29 N = 32768     Run time = 61836 ms
30 N = 65536     Run time = 356046 ms

```

We can see that when $N = 2^{16}$, the DFT algorithm can be more than 2000 time slower than the FFT algorithm. And this number grows when N continues to become bigger.