DSP Programming Homework – Fast Fourier Transform

W42 林子恒 2014011054

2014.10.24

目录

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,...,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.

```
// DFT
    // input_seq[]:
    // N: size of input_seq
complex* DFT(complex input_seq[], int N) {

    // Calc WN
    complex* WN = new complex[N];
    WN = Calc_WN(N);

    complex* DFTed_seq = new complex[N];
    for (int i = 0; i < N; ++i)
    {
        for (int j = 0; j < N; ++j)
        {
            int k_mod = (i*j) % N;
            DFTed_seq[i] = ComplexAdd( DFTed_seq[i], ComplexMul(
                input_seq[j], WN[k_mod]) );
        }
    }

    return DFTed_seq;
}
```

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.

```
complex* DIT_FFT_reordered(complex input_seq[], int N);
complex* DIT_FFT(complex input_seq[], int N, complex WN[], int
    recur_time_count);

// DIT-FFT
    // input_seq[]:
    // N: size of input_seq
        // Must be a 2^k integer
complex* DIT_FFT_reordered(complex input_seq[], int N) {
    // Initialize
    complex* reordered_seq = new complex[N];
    // Calc WN
    complex* WN = new complex[N];
    WN = Calc_WN(N);
    // Reorder
    reordered_seq = reorder_seq(input_seq, N);
    // Calc DIF-FFT
    reordered_seq = DIT_FFT(reordered_seq, N, WN, 0);
    return reordered_seq;
}
complex* DIT_FFT(complex input_seq[], int N, complex WN[], int
    recur_time_count) {
    // cout << "\tDIF_FFT executed!\n"; // for validation
    // output seq
    complex* return_seq = new complex[N];
    if ( N != 2 ) {
        int k = pow(2,recur_time_count);
        // Split input_seq into 2
        complex* first_half_input_seq = new complex[N/2];
        complex* second_half_input_seq = new complex[N/2];
        for (int i = 0; i < N/2; ++i) {
            first_half_input_seq[i] = input_seq[i];
        }
        for (int i = 0; i < N/2; ++i) {
            second_half_input_seq[i] = input_seq[i+N/2];
        }
        // DFT
        complex* DFTed_first_half_seq = new complex[N/2];
        DFTed_first_half_seq = DIT_FFT(first_half_input_seq, N/2,
            WN, recur_time_count+1);
        complex* DFTed_second_half_seq = new complex[N/2];
        DFTed_second_half_seq = DIT_FFT(second_half_input_seq, N/2,
            WN, recur_time_count+1);
        // Calc
        complex* output_first_half_seq = new complex[N/2];
```

```
        complex* output_second_half_seq = new complex[N/2];
        for (int i = 0; i < N/2; ++i) {
            output_first_half_seq[i] = ComplexAdd( DFTed_first_half_seq[i], ComplexMul( DFTed_second_half_seq[i], WN[i*k]) ) ;
        }
        for (int i = 0; i < N/2; ++i) {
            output_second_half_seq[i] = ComplexAdd( DFTed_first_half_seq[i], ComplexMul( ReverseComplex(DFTed_second_half_seq[i]), WN[i*k] ) );
        }
        // Append [output_first_half_seq] & [output_second_half_seq]
        return_seq = append_seq(output_first_half_seq, output_second_half_seq, N/2);
        return return_seq;
    } else if ( N == 2 ) { // Smallest Butterfly Unit
        // cout << "\tDIT_FFT N==2 triggered!\n"; // for validation
        return_seq[0] = ComplexAdd(input_seq[0], ComplexMul( input_seq[1], WN[0]) );
        return_seq[1] = ComplexAdd(input_seq[0], ComplexMul( ReverseComplex(input_seq[1]), WN[0] ) );
        return return_seq;
    }
    // return [return_seq] # unordered
    return return_seq;
}
```

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.

```
complex* DIF_FFT_reordered(complex input_seq[], int N);
complex* DIF_FFT(complex input_seq[], int N, complex WN[], int
    recur_time_count);
// DIF-FFT
    // input_seq[]:
    // N: size of input_seq
        // Must be a 2^k integer
complex* DIF_FFT_reordered(complex input_seq[], int N) {
    // Initialize
    complex* reordered_seq = new complex[N];
    // Calc WN
    complex* WN = new complex[N];
    WN = Calc_WN(N);
    // Calc DIF-FFT
    reordered_seq = DIF_FFT(input_seq, N, WN, 0);
    // Reorder
    reordered_seq = reorder_seq(reordered_seq, N);
    return reordered_seq;
}
complex* DIF_FFT(complex input_seq[], int N, complex WN[], int
    recur_time_count) {
    // cout << "\tDIF_FFT executed!\n"; // for validation
    // output seq
    complex* return_seq = new complex[N];
    if ( N != 2 ) {
        complex* first_half_seq = new complex[N/2];
        complex* second_half_seq = new complex[N/2];
        int k = pow(2,recur_time_count);
        // Calc
        for (int i = 0; i < N/2; ++i) {
            first_half_seq[i] = ComplexAdd(input_seq[i], input_seq[
                i+N/2]) ;
        }
        for (int i = 0; i < N/2; ++i) {
            second_half_seq[i] = ComplexMul( ComplexAdd(input_seq[i
                ], ReverseComplex(input_seq[i+N/2])), WN[i*k] ) ;
        }
        // DFT
        complex* DFTed_first_half_seq = new complex[N/2];
        DFTed_first_half_seq = DIF_FFT(first_half_seq, N/2, WN,
            recur_time_count+1);
        complex* DFTed_second_half_seq = new complex[N/2];
        DFTed_second_half_seq = DIF_FFT(second_half_seq, N/2, WN,
            recur_time_count+1);
        // Append [DFTed_first_half_seq] & [DFTed_second_half_seq]
```

```
        return_seq = append_seq(DFTed_first_half_seq,
            DFTed_second_half_seq, N/2);
        return return_seq;
    } else if ( N == 2 ) { // Smallest Butterfly Unit
        // cout << "\tDIF_FFT N==2 triggered!\n"; // for validation
        return_seq[0] = ComplexAdd(input_seq[0], input_seq[1]);
        return_seq[1] = ComplexMul( ComplexAdd(input_seq[0],
            ReverseComplex(input_seq[1])), WN[0] );
        return return_seq;
    }
    // return [return_seq] # unordered
    return return_seq;
}
```

2.2 Some Other Supporting Functions

2.2.1 Struct of Complex (complex.h)

```
// Complex Struct
typedef struct Complex
{
    double re;
    double im;
    Complex() {
        re = 0;
        im = 0;
    };
    Complex(double a,double b) {
        re = a;
        im = b;
    };
} complex;
complex* append_seq(complex seq_1[], complex seq_2[], int N);
complex* reorder_seq(complex input_seq[], int N);
complex* Calc_WN(int N);
int reverse_bit(int value, int N);
// Multiplier
complex ComplexMul(complex c1, complex c2)
{
    complex r;
    r.re = c1.re*c2.re - c1.im*c2.im;
    r.im = c1.re*c2.im + c1.im*c2.re;
    return r;
}
// Adder
complex ComplexAdd(complex c1, complex c2)
{
    complex r;
    r.re = c1.re + c2.re;
    r.im = c1.im + c2.im;
    return r;
}
// -c
complex ReverseComplex(complex c)
{
    c.re = -c.re;
    c.im = -c.im;
    return c;
}
```

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

```
// Calc WN[], with N = input_N
complex* Calc_WN(int N) {

    cout << "Calculating WN[] of N = " << N << " ... \t";
    complex* WN = new complex[N];

    complex WN_unit; WN_unit.re = cos(2*PI/N); WN_unit.im = -sin(2*
        PI/N);
    WN[0].re=1; WN[0].im=0;

    for (int i = 1; i < N; ++i)
    {
        WN[i] = ComplexMul(WN[i-1], WN_unit);
    }

    return WN;
}
```

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

```
// Reverse Bit
    // input:
        // a decimal num,
        // N-based reverse method
    // output: a decimal num
int reverse_bit(int value, int N) {

    int ret = 0;
    int i = 0;

    while (i < N) {
        ret <<= 1;
        ret |= (value>>i) & 1;
        i++;
    }

    return ret;
}
```

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.

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

2.2.5 Main Function (fft.cpp)

```
# include "complex.h"   // definition of struct complex, Calc of WN
    []
# include "dft.h"       // DFT
# include "dit_fft.h"   // DIT-FFT
# include "dif_fft.h"   // DIF-FFT
# include "validate_n_evaluate.h"

int main(int argc, char ** argv)
{
    // Get argv
        int N_max = atoi(argv[1]); // length of input sequence
            // input 7 to run 2^{10,11,12,13,14,15,16}
            // input 6 to run 2^{10,11,12,13,14,15}
        int validate_or_evaluate = atoi(argv[2]);
            // 1 for validate, 0 for ignore
        int dft_dit_dif = atoi(argv[3]);
            // 1:DFT, 2:DIT, 3:DIF, 4:To compute everything for
                validation
    // Initialize
    // Setup input sequence
        complex* input_seq = new complex[N_max];
        input_seq[0] = complex(1,0);
    // For validation of the result
        if (validate_or_evaluate == 1) {
            validate_result(input_seq, N_max, dft_dit_dif);
            return 0;
        }
    // For compare the performance(run time) of DFT/DIT/DIF
        else if (validate_or_evaluate == 0) {
            evaluate_performance(input_seq, N_max, dft_dit_dif);
            return 0;
        }
    // end
        return 0;
}
```

2.3 Validation of Correctness

Run the following script to obtain the result

```
make fft
./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:

```
Lin, Tzu-Heng's Work, 2014011054, W42
Dept. of Electronic Enigeering, Tsinghua University
Starting, This project is to calc DFT in Original-DFT / DIT-FFT /
    DIF-FFT...
     For Usage, Please see 'fft.cpp'

Calculating DFT...
    X[0] = 2 + j*0
    X[1] = 1.70711 + j*-0.707107
    X[2] = 1 + j*-1
    X[3] = 0.292893 + j*-0.707107
    X[4] = 1.33227e-15 + j*-5.35898e-08
    X[5] = 0.292893 + j*0.707107
    X[6] = 1 + j*1
    X[7] = 1.70711 + j*0.707107
Calculating DIT-FFT...
    X[0] = 2 + j*0
    X[1] = 1.70711 + j*-0.707107
    X[2] = 1 + j*-1
    X[3] = 0.292893 + j*-0.707107
    X[4] = 0 + j*0
    X[5] = 0.292893 + j*0.707107
    X[6] = 1 + j*1
    X[7] = 1.70711 + j*0.707107
Calculating DIF-FFT...
    X[0] = 2 + j*0
    X[1] = 1.70711 + j*-0.707107
    X[2] = 1 + j*-1
    X[3] = 0.292893 + j*-0.707107
    X[4] = 0 + j*0
    X[5] = 0.292893 + j*0.707107
    X[6] = 1 + j*1
    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

```
make fft
./fft 7 0 4 # Calc 2^{10~16} using all 3 methods
# ./fft 6 0 4   # Calc 2^{10~15} 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:

```
Lin, Tzu-Heng's Work, 2014011054, W42
Dept. of Electronic Enigeering, Tsinghua University

This project is to calc DFT in Original-DFT / DIT-FFT / DIF-FFT...
    For Usage, Please see 'fft.cpp'

Calculating DIT-FFT...
N = 1024     Run time = 2 ms
N = 2048     Run time = 6 ms
N = 4096     Run time = 14 ms
N = 8192     Run time = 24 ms
N = 16384    Run time = 53 ms
N = 32768    Run time = 106 ms
N = 65536    Run time = 200 ms
Calculating DIF-FFT...
N = 1024     Run time = 1 ms
N = 2048     Run time = 3 ms
N = 4096     Run time = 7 ms
N = 8192     Run time = 16 ms
N = 16384    Run time = 39 ms
N = 32768    Run time = 79 ms
N = 65536    Run time = 154 ms
Calculating DFT...
N = 1024     Run time = 38 ms
N = 2048     Run time = 180 ms
N = 4096     Run time = 771 ms
N = 8192     Run time = 2948 ms
N = 16384    Run time = 12811 ms
N = 32768    Run time = 61836 ms
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.