Fast Fourier Transform (FFT)

Forrest Zhang
8/9/2024

The Fast Fourier Transform (FFT) is an algorithm in computational mathematics and digital signal processing, which supports efficient computation of the Discrete Fourier Transform (DFT). The DFT is a mathematical technique that transforms a sequence of complex numbers (representing time-domain data) into another sequence of complex numbers representing the frequency-domain components of the original sequence. However, the direct computation of the DFT using its definition has high computational cost. For an input of size n, the direct DFT computation requires O(n^2) operations. This quadratic time complexity becomes prohibitive for large datasets, and it makes FFT a significant improvement with its O(nlogn) time complexity. The FFT algorithm also reduces the computational burden by exploiting the symmetry and periodicity properties of the DFT.

The FFT is widely used in areas such as signal processing, image analysis, and data compression. Among all FFT algorithms, the Cooley-Tukey algorithm is the most commonly used. It uses a divide-and-conquer strategy that recursively breaks down a DFT of any composite size n into several smaller DFTs. This reduction demonstrates the periodic nature of the Fourier transform, which allows efficiently combining results from smaller problems into a solution for the larger problem.

The Cooley-Tukey algorithm first requires the reordering of the input data into a bit-reversed order. This bit-reversal step can ensure that the recursive structure of the algorithm can be effectively applied. The data is then processed in stages, each corresponding to a different level of the recursion. At each stage, pairs of elements are combined using the so-called "butterfly" operation, which involves complex multiplications and additions. The correctness of the FFT algorithm is proved by the properties of the DFT and the structure of the recursive divide-and-conquer approach. The algorithm can be proved correct because it accurately decomposes the DFT into smaller components, combines them using the butterfly operations, and correctly reorders the input data to align with the required bit-reversal pattern.

The following formula is the definition of DFT:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi kn/N}$$

where X(k)represents the frequency component at index k, and x(n) is the time-domain signal. The FFT recursively computes smaller DFTs for even and odd indices, combining them to form the full DFT using the following equation:

$$X(k) = E(k) + W_N^k \cdot O(k)$$

where E(k) and O(k) are the DFTs of the even and odd-indexed elements, respectively, and the term with W is the complex twiddle factor. This recursive structure is guaranteed to produce the correct result because it mirrors the DFT's original formula but reduces the computational effort by combining results from smaller subproblems.

The primary advantage of FFT over the direct DFT computation is its improved time complexity. The Cooley-Tukey algorithm operates with a time complexity of O(nlogn). This improvement arises from the recursive division of the problem into smaller subproblems. At each level of recursion, the problem size is halved, resulting in (nlogn) levels of recursion. At each level, n operations are required to combine the subproblems, leading to the overall O(nlogn) complexity.

The space complexity of FFT is O(n) when using an in-place algorithm, which requires only a small amount of additional memory for storing intermediate results. If the algorithm is implemented non-recursively, additional memory for the call stack can be avoided, further optimizing the space usage.

The design and implementation of the FFT algorithm in C++ starts with the preparation of the input data through bit-reversal reordering. This step is important because it rearranges the input array to align with the recursive nature of the Cooley-Tukey FFT algorithm. Following this, the algorithm proceeds with iterative "butterfly" operations that combine the results of smaller Discrete Fourier Transforms (DFTs) to build up the final frequency-domain representation of the input signal. The core of the implementation relies on efficient complex number arithmetic, which is supported using by C++'s standard library support for complex data types. Furthermore, the use of in-place computation ensures that the space complexity remains linear, making the implementation both time-efficient and memory-efficient.

FFT_Algo (Global Scope)

```cpp
/*
 * Project: Fast Fourier Transform (FFT) Implementation
 * Author: Forrest Zhang
 * Date: 8/8/2024
 * Description:
 * This program implements the Cooley-Tukey Fast Fourier Transform (FFT) algorithm,
 * which efficiently computes the Discrete Fourier Transform (DFT) of a sequence.
 * The algorithm operates with a time complexity of O(n log n) and is widely used in
 * signal processing, image analysis, and other fields where frequency analysis is needed.
 */

#include <iostream>
#include <complex>
#include <vector>
#include <cmath>

 // Define a constant for PI (used in the FFT calculations)
const double PI = 3.141592653589793;

// Define complex number type using C++'s standard library
typedef std::complex<double> Complex;
typedef std::vector<Complex> CArray;

/*
 * description:
 * Reverses the order of bits in an integer.
 *
 * Parameters:
 *   - x: The integer whose bits need to be reversed.
 *   - n: The number of bits to consider in the reversal (log2 of array size).
 *
 * Returns:
 *   - An integer with the bits reversed in comparison to the input integer.
 *
 * Explanation:
 *   Bit reversal is essential in the Cooley-Tukey FFT algorithm to reorder
 *   the input array before performing the FFT. This reordering ensures that
 *   the recursive combination of smaller DFTs is done correctly.
 */
unsigned int reverse_bits(unsigned int x, unsigned int n) {
    unsigned int result = 0;
    for (unsigned int i = 0; i < n; ++i) {
        if (x & (1 << i))  // Check if the i-th bit is set
            result |= 1 << (n - 1 - i);  // Set the corresponding bit in result
    }
    return result;
}
```

(Figure 1: FFT code implementation)

```
unsigned int reverse_bits(unsigned int x, unsigned int n) {
    unsigned int result = 0;
    for (unsigned int i = 0; i < n; ++i) {
        if (x & (1 << i))  // Check if the i-th bit is set
            result |= 1 << (n - 1 - i);  // Set the corresponding bit in result
    }
    return result;
}

/*
 * description:
 * Computes the Fast Fourier Transform (FFT) of the input array.
 *
 * Parameters:
 *   - x: A reference to the array of complex numbers representing the input signal.
 *
 *   The FFT function processes the input array in three main steps:
 *   1. Reorder the input array elements using bit-reversal to prepare for the FFT.
 *   2. Iteratively combine pairs of elements to form the DFT for increasingly larger subproblems.
 *   3. Perform the final combination to obtain the full DFT of the input signal.
 */
void fft(CArray& x) {
    const size_t N = x.size();  // Get the size of the input array
    const unsigned int bits = log2(N);  // Calculate the number of bits needed for bit-reversal

    // Step 1: Bit reversal reordering
    for (unsigned int i = 0; i < N; ++i) {
        unsigned int j = reverse_bits(i, bits);  // Get the bit-reversed index
        if (i < j)
            std::swap(x[i], x[j]);  // Swap elements to reorder the array
    }

    // Step 2 and 3: FFT computation
    for (size_t len = 2; len <= N; len <<= 1) {  // Loop over the size of subproblems (2, 4, 8, ...)
        double angle = -2 * PI / len;  // Calculate the angle for the complex roots of unity
        Complex wlen(cos(angle), sin(angle));  // Precompute the root of unity for the current stage
        for (size_t i = 0; i < N; i += len) {  // Loop over each subproblem
            Complex w(1);  // Initialize the complex rotation factor
            for (size_t j = 0; j < len / 2; ++j) {  // Loop over the elements within each subproblem
                Complex u = x[i + j];  // Extract the first element of the pair
                Complex v = x[i + j + len / 2] * w;  // Apply the rotation factor to the second element
                x[i + j] = u + v;  // Combine the elements (Butterfly operation)
                x[i + j + len / 2] = u - v;  // Store the difference in the second half
                w *= wlen;  // Update the rotation factor for the next pair
            }
        }
    }
}
```

(Figure 2: FFT code implementation)

```cpp
/*
 * Function: main
 * --------------
 * Entry point of the program. Initializes a sample input array, performs the FFT,
 * and prints the results to the console.
 *
 * Explanation:
 *   This function demonstrates the usage of the FFT function by providing a simple
 *   example where the input array consists of 1s followed by 0s. The output is the
 *   frequency domain representation of the input signal.
 */
int main() {
    const int N = 8;  // Sample size (must be a power of 2 for this implementation)
    Complex test[N] = { 1, 1, 1, 1, 0, 0, 0, 0 };  // Example input array (time domain signal)

    CArray data(test, test + N);  // Initialize the data array with the sample input

    std::cout << "Input data:" << std::endl;
    for (int i = 0; i < N; ++i) {
        std::cout << data[i] << std::endl;  // Print the input data
    }

    fft(data);  // Perform the FFT on the input data

    std::cout << "\nFFT output:" << std::endl;
    for (int i = 0; i < N; ++i) {
        std::cout << data[i] << std::endl;  // Print the FFT output (frequency domain signal)
    }

    return 0;
}
```

(Figure 3: FFT code implementation)

```
Microsoft Visual Studio Debu
Input data:
(1,0)
(1,0)
(1,0)
(1,0)
(0,0)
(0,0)
(0,0)
(0,0)

FFT output:
(4,0)
(1,-2.41421)
(0,0)
(1,-0.414214)
(0,0)
(1,0.414214)
(0,0)
(1,2.41421)

C:\Users\17205\Desktop\Repository\FFT_Algorithm\FFT_Algo\x64\Debug\FFT_Algo.exe (process 42044) exited with code 0.
Press any key to close this window . . .
```

(Figure 4: FFT program output display)

References

- GeeksforGeeks. "Fast Fourier Transformation | Polynomial Multiplication." https://www.geeksforgeeks.org/fast-fourier-transformation-poynomial-multiplication/

- MIT OpenCourseWare. "Lecture 25: Fast Fourier Transform (FFT)." MIT https://ocw.mit.edu/courses/mathematics/18-06-linear-algebra-spring-2010/lecture-videos/lecture-25-fast-fourier-transform-fft/