A Simple and Efficient FFT Implementation in C++, Part I

Vlodymyr Myrnyy - May 10, 2007

Vlodymyr teaches at Brandenburg University of Technology, Cottbus, Germany. He can be reached at myrnyy@math.tu-cottbus.de. This article provided courtesy of Dr. Dobb's Journal.

This article describes a new efficient implementation of the Cooley-Tukey fast Fourier transform (FFT) algorithm using C++ template metaprogramming. Thank to the recursive nature of the FFT, the source code is more readable and faster than the classical implementation. The efficiency is proved by performance benchmarks on different platforms.

Introduction

Fast Fourier Transformation (FFT) is not only a fast method to compute digital Fourier transformation (DFT)—having a complexity $O(N\log(N))$ (where $N$ must be power of 2, $N=2^p$), it is a way to linearize many kinds of real mathematical problems of nonlinear complexity using the idiom “divide and conquer.”

The discrete Fourier transform $f_n$ of the $N$ points signal $x_n$ is defined as a sum:

$$f_k = \sum_{j=0}^{N-1} g^{jk} x_j, \quad g = e^{2\pi i/N},$$

Example 1.

where $i$ is the complex unity. Put simply, the formula says that an algorithm for the computing of the transform will require $O(N^2)$ operations. But the Danielson-Lanczos Lemma (1942), using properties of the complex roots of unity $g$, gave a wonderful idea to construct the Fourier transform recursively (Example 1).

$$f_k = f_k^e + g^k f_k^o, \quad k = 0...N - 1,$$

Example 2.

where $f_k^e$ denotes the $k$-th component of the Fourier transform of length $N/2$ formed from the even components of the original $x_n$, while $f_k^o$ is the corresponding transform formed from the odd components. Although $k$ in the last line of Example 2 varies from 0 to $N-1$, the transforms $f_k^e$ and $f_k^o$ are periodic in $k$ with length $N/2$. The same formula applied to the transforms $f_k^e$ and $f_k^o$ reduces the problem to the transforms of length $N/4$ and so on. It means, if $N$ is a power of 2, the transform will be computed with a linear complexity $O(N\log(N))$. More information on the mathematical background of the FFT and advanced algorithms, which are not limited to $N=2^p$, can be found in many books, e.g. [3,4].
I would like to start with the simplest recursive form of the algorithm, that follows directly from the relation in Example 2:

```
FFT(x) {
  n=length(x);
  if (n==1) return x;
  m = n/2;
  X = \{x_{2j}\}_{j=0}^{m-1};
  Y = \{x_{2j+1}\}_{j=0}^{m-1};
  X = FFT(X);
  Y = FFT(Y);
  U = \{X_{k mod m}\}_{k=0}^{n-1};
  V = \{g^{-k}Y_{k mod m}\}_{k=0}^{n-1};
  return U+V;
}
```

This algorithm should give only a first impression of the FFT construction. The \( FFT(x) \) function is called twice recursively on the even and odd elements of the source data. After that some transformation on the data is performed. The recursion ends if the data length becomes 1. This recursion form is instructive, but the overwhelming majority of FFT implementations use a loop structure first achieved by Cooley and Tukey [2] in 1965. The Cooley-Tukey algorithm uses the fact that if the elements of the original length \( N \) signal \( x \) are given a certain "bit-scrambling" permutation, then the FFT can be carried out with convenient nested loops. The scrambling intended is reverse-binary reindexing, meaning that \( x_j \) gets replaced by \( x_k \), where \( k \) is the reverse-binary representation of \( j \). For example, for data length \( N=2^5 \), the element \( x_5 \) must be exchanged with \( x_{(20)} \), because the binary reversal of \( 5=00101_2 \) is \( 10100_2=20 \). The implementation of this data permutation will be considered later, since it has been a minor part of the whole FFT. A most important observation is that the Cooley-Tukey scheme actually allows the FFT to be performed in place, that is, the original data \( x \) is replaced, element by element, with the FFT values. This is an extremely memory-efficient way to proceed with large data. Listing One shows the classical implementation of the Cooley-Tukey algorithm from Numerical Recipes in C++ [5], p.513.

```
void four1(double* data, unsigned long nn) {
  // reverse-binary reindexing
  unsigned long n, mmax, m, j, istep, i;
  double wtemp, wr, wpr, wpi, wi, theta;
  double tempr, tempi;
  n = nn<<1;
  j=1;
  for (i=1; i<n; i+=2) {
    if (j>i) {
      swap(data[j-1], data[i-1]);
      swap(data[j], data[i]);
    }
    m = nn;
    while (m>=2 && j>m) {
      j -= m;
      m >>= 1;
    }
    // ..
  }
}
```
The initial signal is stored in the array `data` of length $2^{nn}$, where each even element corresponds to the real part and each odd element to the imaginary part of a complex number. **Recursion Is Not Evil**

Most of the known approaches to the FFT implementation are based on avoiding the natural FFT recursion, replacing it by loops. But a recursion is not expensive anymore, if it is resolved at compile-time, as it happens with template class recursion. Moreover, this kind of recursion can give performance benefits, since the code has been better unrolled than usual loops. This idea seems to be very similar to the approach of Todd Veldhuizen [6], who rewrote the same Cooley-Tukey algorithm (Listing One) completely in template metaprograms. The nested loops became recursive templates with nonlinear complexity, which can be compiled at most for $N=2^{12}$ on modern workstations taking much time and memory. Being quite efficient, this implementation has not been applied to real technical problems, because they often need to handle larger amounts of data. From these two points of view, I try to find a "golden section" using the efficiency of template metaprogramming and reducing compile-time to make the implementation applicable to huge signals limited only by physical memory.

The approach presented here exploits the original recursive nature of the FFT implementing the Danielson-Lanczos relation (Example 2) using template class recursion. The necessary assumption is
that the length of the signal \( N=2^P \) is a static constant and is passed as template parameter \( P \). I start
in the high abstraction level dividing the algorithm from Listing One into two parts: the scrambling
and the Danielson-Lanczos section. Listing Two represents the initial template class \( GFFT \) with
member function \( \text{fft}(T^*\ data) \) including two parts of the transform.

```cpp
template<unsigned P,
    typename T=double>
class GFFT {
enum { N = 1<<P }; //DanielsonLanczos<N,T> recursion;
public:
    void fft(T* data) {
        scramble(data,N);
        recursion.apply(data);
    }
};
```

Listing Two The main point now is the implementation of \( DanielsonLanczos \) template class using
recursive templates, where \( P \) is the power of 2 defining \( N \). Type \( T \) is default type of the data
elements. The implementation of the function \( \text{scramble} \) will be mentioned briefly later and for now
could be taken over from Listing One. \( DanielsonLanczos \) template class in Listing Three depends on
the integer \( N \) defining the current length of the data in the recursion and on the same type \( T \). To
avoid the nonlinear number of instantiated templates, I define only one template class
\( DanielsonLanczos<N/2,T> \) per recursion level. Therefore, the total number of the template classes
to be instantiated is \( P+1 \). The constant \( P \) can not be large because of the physical memory limits. For
instance, if you have a data with complex elements of double precision (2x8 bytes per element), then
\( P \) may vary from 1 to 27 on a 32-bit platform. The case \( P=28 \) corresponds to 4GB of data and there is
no memory for some other program variables. \( P \) can be bigger on 64-bit processors, but it's limited
again by available physical memory. Such a number of the instantiated template classes should not
provide any compilation problems. The recursive idea of the Danielson-Lanczos relation is realized
by two recursive calls of the member function \( \text{apply} \): the first time with the original signal data and
the second time shifted by \( N \). Every next recursion level divides \( N \) by 2. The last one is specialized
for \( N=1 \) and includes empty member function \( \text{apply} \).

```cpp
template<unsigned N, typename T=double>
class DanielsonLanczos {
    DanielsonLanczos<N/2,T> next;
public:
    void apply(T* data) {
        next.apply(data);
        next.apply(data+N);
        T tempr,tempi,c,s;
        for (unsigned i=0; i<N; i+=2) {
            c = cos(i*M_PI/N);
            s = -sin(i*M_PI/N);
            tempr = data[i+N]*c - data[i+N+1]*s;
            tempi = data[i+N]*s + data[i+N+1]*c;
            data[i+N] = data[i] - tempr;
            data[i+N+1] = data[i+1] - tempi;
            data[i] += tempr;
        }
    }
};
```
data[i+1] += tempi;
};

template<typename T>
class DanielsonLanczos<1,T> {
public:
  void apply(T* data) { }
};

Listing Three

After the recursion has been finished, the data is modified in the loop, where cos and sin functions used to compute the complex roots of unity (c,s). The resulting (tempr,tempi) is a temporary complex number to modify (data[i+N],data[i+N+1]) and (data[i],data[i+1]). This simple implementation in Listing Three has poor performance due to many computations of trigonometric functions.

Elimination of Trigonometric Function Calls

The original implementation in Listing One contains a nice property of roots of unity: their recurrence calculation. Listing Four is the next implementation step with the same recurrence formula. Computing only two sine functions and starting with 1, the next roots (wr,wi) are calculated recurrently from (wpr,wpi): (wr,wi)+=(wr,wi)*(wpr,wpi). The end-specialization of DanielsonLanczos template class stays unchanged.

template<unsigned N, typename T=double>
class DanielsonLanczos {
  DanielsonLanczos<N/2,T> next;
public:
  void apply(T* data) {
    next.apply(data);
    next.apply(data+N);
    T wtemp, tempr, tempi, wr, wi, wpr, wpi;
    wtemp = sin(M_PI/N);
    wpr = -2.0*wtemp*wtemp;
    wpi = -sin(2*M_PI/N);
    wr = 1.0;
    wi = 0.0;
    for (unsigned i=0; i<N; i+=2) {
      tempr = data[i+N]*wr - data[i+N+1]*wi;
      tempi = data[i+N]*wi + data[i+N+1]*wr;
      data[i+N] = data[i].tempr;
      data[i+N+1] = data[i+1].tempi;
      data[i] += tempr;
      data[i+1] += tempi;
      wtemp = wr;
      wr += wr*wpr - wi*wpi;
      wi += wi*wpr + wtemp*wpi;
    }
  }
};
Listing Four now represents a C++ version of the C-style algorithm from Listing One. The latter includes three loops with dependent indexes, which have been exchanged with template class recursion and a single loop in the C++ version. Although they are very similar, the tests show different performance on both 32-bit (Figure 1a) and 64-bit (Figure 1b) processors. The dashed line "CT" is the performance of the original C-style implementation. The line "GFFT1" corresponds to the implementation in Listing Three, and "GFFT2" to that in Listing Four. "GFFT" denotes the final generic FFT implementation, which will be reached in the second article in this series, after some improvements. Performance of the original CT-implementation is pretty high until some data length: $P=14$ on P4 Xeon and $P=16$ on AMD Opteron. For bigger $P$, the CPU time grows faster and the C-style implementation becomes very inefficient compared to the recursive templates implementation. What are these "magic" numbers? These powers of two correspond to the data length smaller than the L2 cache memory of the workstations: 512 KB on P4 Xeon and 2 MB on AMD Opteron. It means, for the next power of two, the data does not fit the cache and performance falls. This is not the case in the new implementation because it contains only one loop of constant length and compiler can better unroll the code.

Figure 1a: Performance on 32-bit processors.
The essential reduction of trigonometric function calculations from $2^2P$ to $2P$ made the FFT about four times faster. Moreover, it opens a nice observation, that all $2P$ sine functions receive static constants $M\ PI/N$ and $2*M\ PI/N$, which have been already known at compile-time. Therefore, the corresponding sine function values could be known at compile-time as well. Let the C++ compiler compute them! An implementation of such template metaprogramming is not new—Todd Veldhuizen described it in the articles [6,7]. The sine values are calculated using series expansion in a static member function of the template class SinCosSeries, outlined in Listing Five. All necessary arguments are template parameters. Defining a template metaprogram SinCosSeries to compute the series from member $M$ to $N$, I can write compile-time $\sin(A*M\ PI/B)$ and $\cos(A*M\ PI/B)$ functions for both single and double precision data and substitute their "calls" in two lines of Listing Four containing $\sin()$:

\[
\text{wtemp} = -\text{Sin}<N,1,T>::\text{value}();
\]

and

\[
\text{wpi} = -\text{Sin}<N,2,T>::\text{value}();
\]

Actually, compile-time functions are not called. Their values are evaluated at compile-time and stored in the code as static constants. It means complete elimination of runtime trigonometric functions and 20%-60% additional performance.

```cpp
template<unsigned M, unsigned N, unsigned B, unsigned A>
struct SinCosSeries {
    static double value() {
```
return 1-(A*M_PI/B)*(A*M_PI/B)/M/(M+1)
*SinCosSeries<M+2,N,B,A>::value();
}
};

template<unsigned N, unsigned B, unsigned A>
struct SinCosSeries<N,N,B,A> {
  static double value() { return 1.; }
};

template<unsigned B, unsigned A, typename T=double>
struct Sin;

// In part two of this article, I will discuss the specialization of short FFTs, FFT selection at runtime, and present some comparative benchmarks and conclusions.

References
   http://www.onumerics.org/blitz/examples/fft.html