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

[**Vlodymyr Myrnyy**](http://www.eetimes.com/document.asp?doc_id=1275415&)

5/10/2007 07:00 AM EDT   
[Post a comment](http://www.eetimes.com/document.asp?doc_id=1275415&print=yes#msgs)

[inShare](javascript:void(0);)

*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*](http://www.ddj.com/)*.*

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(Nlog(N))* (where *N* must be power of 2, *N=2P*), 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 *fn* of the *N* points signal *xn* is defined as a sum:

http://m.eet.com/media/1066933/070510vm01_e1.gif  
*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(N2)* 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).

http://m.eet.com/media/1066934/070510vm01_e2.gif  
*Example 2.*

where *fke* denotes the *k*-th component of the Fourier transform of length *N/2* formed from the even components of the original *xj*, while *fko* 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 *fke* and *fko* are periodic in *k* with length *N/2*. The same formula applied to the transforms *fke* and *fke* 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(Nlog(N))*. More information on the mathematical background of the FFT and advanced algorithms, which are not limited to *N=2P*, 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 *xj* gets replaced by *xk*, where *k* is the reverse-binary representation of *j*. For example, for data length *N*=25, the element *x5* must be exchanged with *x{20}*, because the binary reversal of 5=001012 is 101002=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)

{

unsigned long n, mmax, m, j, istep, i;

double wtemp, wr, wpr, wpi, wi, theta;

double tempr, tempi;

// reverse-binary reindexing

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;

}

j += m;

};

// here begins the Danielson-Lanczos section

mmax=2;

while (n>mmax) {

istep = mmax<<1;

theta = -(2\*M\_PI/mmax);

wtemp = sin(0.5\*theta);

wpr = -2.0\*wtemp\*wtemp;

wpi = sin(theta);

wr = 1.0;

wi = 0.0;

for (m=1; m < mmax; m += 2) {

for (i=m; i <= n; i += istep) {

j=i+mmax;

tempr = wr\*data[j-1] - wi\*data[j];

tempi = wr \* data[j] + wi\*data[j-1];

data[j-1] = data[i-1] - tempr;

data[j] = data[i] - tempi;

data[i-1] += tempr;

data[i] += tempi;

}

wtemp=wr;

wr += wr\*wpr - wi\*wpi;

wi += wi\*wpr + wtemp\*wpi;

}

mmax=istep;

}

}

*Listing One*

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  
**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=2P* 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 *fft(T\* data)* including two parts of the transform.

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 *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 *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 *apply*.

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  
**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;

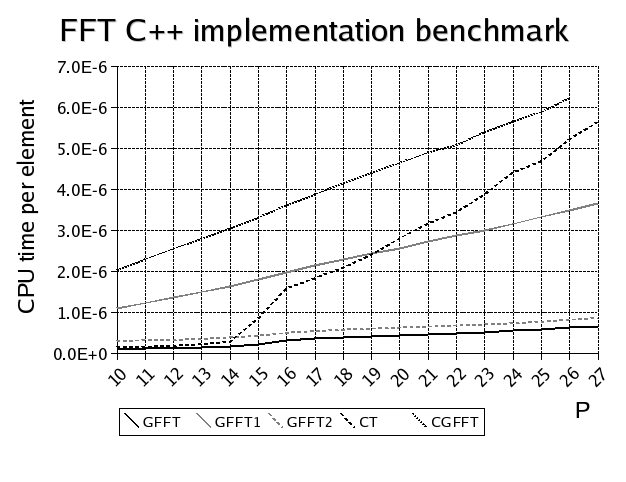
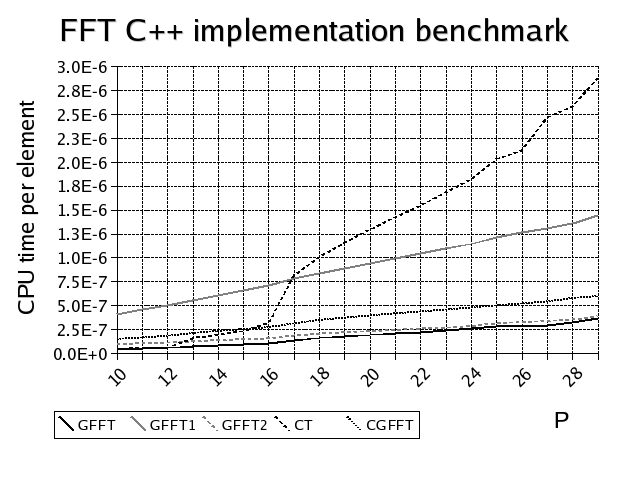
wr += wr\*wpr - wi\*wpi;

wi += wi\*wpr + wtemp\*wpi;

}

}

};

*Listing Four* 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.*   
*Figure 1b: Performance on 64-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()*:

wtemp = -Sin<N,1,T>::value();

and

wpi = -Sin<N,2,T>::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.

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;

template<unsigned B, unsigned A>

struct Sin<B,A,float> {

static float value() {

return (A\*M\_PI/B)\*SinCosSeries<2,24,B,A>::value();

}

};

template<unsigned B, unsigned A>

struct Sin<B,A,double> {

static double value() {

return (A\*M\_PI/B)\*SinCosSeries<2,34,B,A>::value();

}

};

template<unsigned B, unsigned A, typename T=double>

struct Cos;

template<unsigned B, unsigned A>

struct Cos<B,A,float> {

static float value() {

return SinCosSeries<1,23,B,A>::value();

}

};

template<unsigned B, unsigned A>

struct Cos<B,A,double> {

static double value() {

return SinCosSeries<1,33,B,A>::value();

}

};

*Listing Five*

In [part two](http://www.dspdesignline.com/howto/showArticle.jhtml?articleID=199903276) of this article, I will discuss the specialization of short FFTs, FFT selection at runtime, and present some comparative benchmarks and conclusions.

**References**  
1. A. Alexandrescu. *Modern C++ Design*. Addison-Wesley, 2001.  
2. J.W. Cooley, J.W. Tukey. An algorithm for the machine calculation of complex Fourier series. Math. Comp. Vol. 19, 1965, pp. 297-301.  
3. R. Crandall, C. Pomerance. *Prime Numbers: A computational perspective.* Springer, 2005.  
4. H.J. Nussbaumer. *Fast Fourier transform and convolution algorithms.* Springer, Berlin, 1981.  
5. W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P.Flannery. *Numerical Recipes in C++.* Cambridge university press, 2002.  
6. T. Veldhuizen. *Fast Fourier Transform (FFT) implemented using Template Metaprograms.* http://www.oonumerics.org/blitz/examples/fft.html  
7. T. Veldhuizen. *Using C++ template metaprograms*, C++ Report, Vol. 7 No. 4 (May 1995), pp. 36-43. http://osl.iu.edu/~tveldhui/papers/Template-Metaprograms/meta-art.html

# A Simple and Efficient FFT Implementation in C++: Part II

[**Volodymyr Myrnyy**](http://www.eetimes.com/document.asp?doc_id=1275416&)

5/25/2007 07:00 AM EDT   
[Post a comment](http://www.eetimes.com/document.asp?doc_id=1275416&print=yes#msgs)

[inShare](javascript:void(0);)

*Vlodymyr teaches at Brandenburg University of Technology, Cottbus, Germany. He can be reached at myrnyy@math.tu-cottbus.de. This article presented courtesy of Courtesy of* [*Dr. Dobb's Journal*](http://www.ddj.com/)*.*

In [Part One](http://www.eetimes.com/document.asp?doc_id=1275415) of this article, I introduced a new efficient implementation of the Cooley-Tukey Fast Fourier Transform (FFT) algorithm, discussing recursion and the elimination of trigonometric function calls. In Part Two, I discuss the specialization of short FFTs, FFT selection at runtime, and present some comparative benchmarks and conclusions.

**Specialization of Short FFTs**  
The implemented template class recursion has P levels. Every FFT calculation process runs from level P to level 1, while the level 1 is empty (Listing Three, in [part one](http://www.dspdesignline.com/howto/showArticle.jhtml?articleID=199903272) of this article). Some comprehensive books on FFT, for example [4], show that short length FFTs (P=1,2,3,4) could use fewer operations than the general algorithm. Such particular cases can be simply incorporated into the new implementation using partial specialization of the template class *DanielsonLanczos*. Listing Six represents those specializations for N=4 and N=2. Since every FFT goes down to the first specialized one, these additional specializations lead to a small overall performance improvement of about 1-5 percent.

template<typename T>

class DanielsonLanczos<4,T> {

public:

void apply(T\* data) {

T tr = data[2];

T ti = data[3];

data[2] = data[0]-tr;

data[3] = data[1]-ti;

data[0] += tr;

data[1] += ti;

tr = data[6];

ti = data[7];

data[6] = data[5]-ti;

data[7] = tr-data[4];

data[4] += tr;

data[5] += ti;

tr = data[4];

ti = data[5];

data[4] = data[0]-tr;

data[5] = data[1]-ti;

data[0] += tr;

data[1] += ti;

tr = data[6];

ti = data[7];

data[6] = data[2]-tr;

data[7] = data[3]-ti;

data[2] += tr;

data[3] += ti;

}

};

template<typename T>

class DanielsonLanczos<2,T> {

public:

void apply(T\* data) {

T tr = data[2];

T ti = data[3];

data[2] = data[0]-tr;

data[3] = data[1]-ti;

data[0] += tr;

data[1] += ti;

}

};

*Listing Six*

You might ask why, when programming in C++, am I still using a C-style array instead of *std::complex<T>* or even *std::vector<T>*? Because the C++ standard library is not suited to high-performance computing, at least its open source distribution that I have. A simple trace into the sources of the header complex makes clear, that a simple operation on complex numbers like x=a+b written in C++ generates a temporary object of type *std::complex<T>* resulting in poor performance demonstrated by the dotted line 'CGFFT' on Figure 1 (see [Part One](http://www.eetimes.com/document.asp?doc_id=1275415) of this article). This is an example where the expression templates technique in the header {complex} could be very helpful, but was not applied. Application of *std::complex<T>* could result in some shorter code, but the computational performance, which I try to maximize, would be lost.  
FFT Selection at Runtime  
**FFT Selection at Runtime**  
The new generic FFT implementation (GFFT) depends on a constant parameter P, specified before compilation. What should you do, if length of the data will be known at runtime at first? Write a big switch? No. It would not really solve the problem. To overcome compile-runtime parameter definition, I apply object factory, as described in Alexandrescu's *Modern C++ Design* [1], Chapter 8. The source code of the template class *Factory* in header Factory.h is part of the Loki library supplementing the book as well as added to the [full source code](http://m.eet.com/media/1066937/myrnyy_fft_source_code.zip) of this article.

The idea is quite simple: All the template classes with varying template parameters are inherited from a single base class, such as *AbstractFFT* in Listing Seven, where necessary member functions are declared as virtual. The base class is passed to GFFT as an additional template parameter *FactoryPolicy*. It allows you also to substitute an empty base class *EmptyFFT* without virtual function declaration and thus avoid the virtual function call penalty. This is the default case, if GFFT is used without object factory. GFFT receives also a unique identification number id=P and a static object generation function *Create()* to conform to the object factory requirements.

template<typename T>

class AbstractFFT {

public:

virtual void fft(T\*) = 0;

};

class EmptyFFT { };

template<unsigned P, typename T=double,

class FactoryPolicy=EmptyFFT>

class GFFT:public FactoryPolicy {

// ...

public:

enum { id = P };

static FactoryPolicy\* Create() {

return new GFFT<P,T,FactoryPolicy>();

}

// ...

};

*Listing Seven* Each GFFT template class that should be available at runtime is registered in the object factory by its identification number using the *FactoryInit* template metaprogram from Listing Eight. The initializer receives template classes in the form of a *Typelist* ([1], Chapter 3). Template metaprogram *GFFTList* (Listing Eight) constructs such a *Typelist*, receiving an FFT template class, e.g. GFFT, as well as the first and the last value of P. Thus, the creation of the FFT object factory in a program looks like this:

Loki::Factory<AbstractFFT<double>,unsigned int> gfft\_factory;

The new factory bears information about the base class, but is still empty. Metaprograms *GFFTList* and *FactoryInit* help to add the necessary GFFT template classes into the factory writing a single line:

FactoryInit<GFFTList<GFFT,10,27>

::Result>::apply(gfft\_factory);

The factory contains now all the GFFT implementations from P=10 to P=27. To create a needed object instance at runtime on demand and to run the FFT, type:

AbstractFFT<double>\* gfft = gfft\_factory.CreateObject(P);

gfft->fft(data);

template<class TList>

struct FactoryInit;

template<class H, class T>

struct FactoryInit<Loki::Typelist<H,T> > {

template<class Fact>

static void apply(Fact& f) {

f.Register(H::id,H::Create);

FactoryInit<T>::apply(f);

}

};

template<>

struct FactoryInit<Loki::NullType> {

template<class Fact>

static void apply(Fact&) { }

};

template<

template<unsigned,class,class> class FFT,

unsigned Begin, unsigned End,

typename T=double,

class FactoryPolicy=AbstractFFT<T> >

struct GFFTList {

typedef Loki::Typelist<FFT<Begin,T,FactoryPolicy>,

typename GFFTList<FFT,Begin+1,End,T,

FactoryPolicy>::Result> Result;

};

template<

template<unsigned,class,class> class FFT,

unsigned End, typename T, class FactoryPolicy>

struct GFFTList<FFT,End,End,T,FactoryPolicy> {

typedef Loki::NullType Result;

};

*Listing Eight*

As a result, the GFFT obtains a kind of policy-driven design, which makes the implementation very flexible concerning base class. The same procedure can be applied to other independent FFT components, for example, to the element reindexing (function scramble) and the Danielson-Lanczos section.

The element reindexing could also be implemented using recursive template classes. But to preserve only one template class definition per recursion level, the handled indexes (original and bit-reverse) must be non-constant and passed as parameters to some member function. Tests of such implementation showed the same performance and there is no reason to load compiler with another template recursion and to waste the compile-time.  
Comparative Benchmarks and Conclusions  
**Comparative Benchmarks and Conclusions**   
Definition of the GFFT object factory means instantiation of all needed GFFT template classes during compilation. Each of them includes *O(P)* recursive template classes. A reasonable question: how long will the compilation take? GNU C++ 4.x was used to compile performance tests shown in Figures 1 and 2, and took about 10 seconds with optimization keys for a full range of values P in the object factory, because the overall number of different templates to be instantiated stays small. Microsoft Visual C++ 2003 and Intel C++ 8.x and later versions showed similar results, although not all C++ compilers can treat the recursive templates of the GFFT so efficiently. I suppose some older compilers try to instantiate template classes for each member function call even if the template classes are the same. There exist *O(2^P)* such member function calls in the GFFT. As a result, the compilation hangs and finally fails, after entire memory has been exhausted. Those unlucky examples are IBM C++ 6 (Visual Age) and GNU C++ 3.x, but not GNU C++ 2.96! They could compile the GFFT object factory without optimization options quite fast, but hung with optimization turned on.

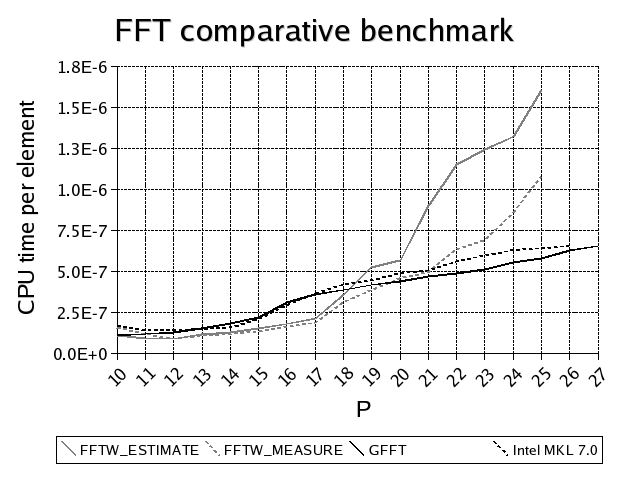
A nice property of the GFFT that distinguishes it from other well known FFT libraries is that a runtime computation of roots of unity using sine and cosine functions is not needed any more. Usual FFT implementations of known libraries like FFTW or Intel MKL proceed in two steps:

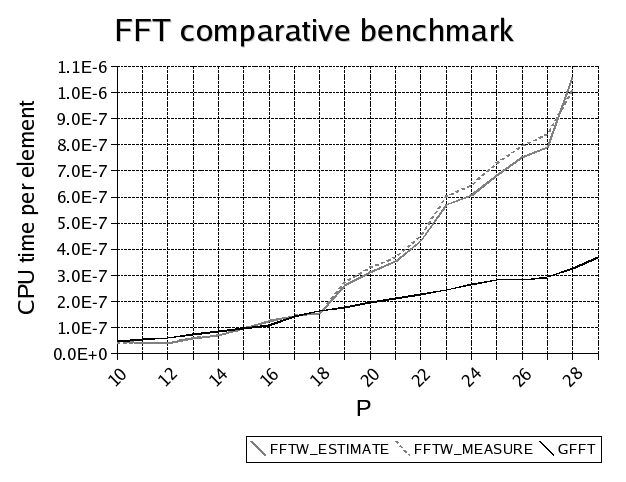
1. "Planning" including calculation of roots of unity
2. Computing of FFT.

The second step can be repeated for different data of the same length. The GFFT omits the first step without any additional penalty. Figure 2 shows benchmarks of the GFFT compared to FFTW with options FFTW\_ESTIMATE and FFTW\_MEASURE as well as routine zfft1dc from Intel MKL 7.0, where total CPU time of both steps was measured. Being very good for small P, the performance of FFTW become poor for large P. The only hardware optimized Intel MKL has a similar high performance comparing to the GFFT. Qualitatively the same benchmark results were obtained on Itanium2 processor with Intel C++ compiler.

Finally, I would like to make a general conclusion as to the differences between the GFFT and traditional implementation. What made the GFFT so efficient? The essential step was the assumption of P to be static constant. The reason was not simply to play with template metaprogramming, but to give additional information to the compiler, so that it could optimize the code better. Really, many numerical algorithms have some integer parameters, which vary in a small range (like P in FFT). It can be assumed static and should be used as far as possible in template metaprogramming, but not to overload the compiler taking care about the total number of template classes to be instantiated. The final implementation can be more efficient, since the compiler has got more information and so an opportunity to optimize the code. You can use such an implementation originally with static parameter or compile it for all possible or needed parameter values within object factory and use as a library. The latter means you can choose highly optimized pieces of code without dynamic bindings at runtime.

This article described a C++ implementation technology on example of simple classical Cooley-Tukey FFT algorithm. Of course, there are many advanced and more complicated FFT algorithms, such as Winograd FFT, parallelized FFT, and the like. Those modern FFT approaches can also take advantages of template recursion and metaprogramming to improve performance significantly.

*Figure 2a: Comparative benchmark, part 1.*

*Figure 2b: Comparative benchmark, part 2.*

**References**  
1. A. Alexandrescu. *Modern C++ Design*. Addison-Wesley, 2001.  
2. J.W. Cooley, J.W. Tukey. An algorithm for the machine calculation of complex Fourier series. Math. Comp. Vol. 19, 1965, pp. 297-301.  
3. R. Crandall, C. Pomerance. *Prime Numbers: A computational perspective.* Springer, 2005.  
4. H.J. Nussbaumer. *Fast Fourier transform and convolution algorithms.* Springer, Berlin, 1981.  
5. W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P.Flannery. *Numerical Recipes in C++.* Cambridge university press, 2002.  
6. T. Veldhuizen. *Fast Fourier Transform (FFT) implemented using Template Metaprograms.* http://www.oonumerics.org/blitz/examples/fft.html  
7. T. Veldhuizen. *Using C++ template metaprograms*, C++ Report, Vol. 7 No. 4 (May 1995), pp. 36-43. http://osl.iu.edu/~tveldhui/papers/Template-Metaprograms/meta-art.html