テンプレート・メタ・プログラミングによるFFTの適応的最適化

神 戸 隆 行†

現在，数多くの様々な最適化技術が研究されているが，そのすべての最適化技術，特に問題依存度の高い最適化を組み込むことは処理系の肥大化を招く．そこでこのような問題依存度の高い最適化は問題の解法とともに処理系ではなくプログラム部品としてライブラリ化することが考えられる．その手段の 1 つとしてメタ・プログラミングがある．そしてこのように最適化をライブラリに組み込むにあたっては実行環境の違いをどう反映するかという問題がある．これについては最適化に適当なパラメータを導入し，実行環境で試行・計測してパラメータを求める実行時プロファイリングという方法がある．本発表では FFT を例にとり，メモリ階層を意識した最適化を C++テンプレートの機能を用いたメタ・プログラミングで行うとともに，実行時プロファイリングに基づいて適応的な最適化を行ったので報告する．これは 3 つの段階からなる．1) 核となる小さなサイズのデータに対する FFT コードをサイズごとに複数生成（ループの展開，三角関数値の静的計算）．2) 前段で生成したサイズごとの核コードの実行時間計測．3) 計測結果に基づく核コードの選択・合成による最終的な FFT コードの生成．特にこのうち 1) と 3) で C++テンプレート・メタ・プログラミング技法を利用した．以上の実装と評価について報告を行う．

An Adaptive Optimization of FFT with Template Meta-programming

Takayuki Kando†

Although much various optimization technologies are studied now, including all these optimization technologies, especially highly problem dependent optimizations bloats code size of compiler too much. One approach is the following: the optimizations build into a part of component library the solution instead of build into compiler itsself, which is able to realize with meta-programming technique. But including the optimizations in a library in this way, there is a problem how to reflect the difference in execution environment. This problem is dealt with introducing parameters for optimization, profiling trial execution in an execution environment, and looking for a suitable parameter value. In this presentation, FFT is taken for the example, and we describe memory hierarchy conscious optimization for FFT, which is implemented in meta-programming technique, and adaptive optimization based on execution time profiling. Our method consists of three steps. Step1: They are some FFT code generation (ex. unrolling of a loop, static evaluation of a trigonometric-functions value) for each small constant size data used as a kernel. Step2: Execution time measurement of the kernel for every size generated in the preceding step. Step3: Generation of the final FFT code by selection and composition of the kernel based on the measurement results. C++ template meta-programming technique was used in Step 1 and Step3. The above method and its evaluation are reported.

1. はじめに

現在，数多くの様々な最適化技術が研究されているが，どの最適化が効果を上げるか，あるいはどういう順序で最適化を適用するかということは問題依存の傾向があるし，問題の数学的な性質を利用した高レベルの最適化は自動化が難しく高度に問題依存である．このように問題に依存する度合いの高い最適化は特に問題に特化した最適化（Domain Specific Optimization）と呼ばれる．これら問題依存度の高い最適化までもすべてコンパイラに組み込むことはコンパイラの肥大化を際限なく進めることになり望ましくない．そこでこのような問題依存度の高い最適化は問題の解法とともに処理系ではなくプログラム部品としてライブラリ化することが考えられる．このように，単に呼ばれて受動的に動くだけのオブジェクトやサブルーチンの集まりという従来のライブラリ枠を超えて最適化など従来コンパイラなどが担っていた機能の一部をライブラリ化したものを**アクティブ・ライブラリ**[17] と呼ぶ．こ

† フリーランス
freelance

のようなことを実現する 1 つの手段としてメタ・プログラミングがある．特に本発表の中ではメタ・プログラミングの中でも**生成的（generative）プログラミング**[4] に注目する．

一方，最適化をライブラリに組み込むにあたっては実行環境の違いをどうライブラリに反映するかという問題がある．この問題は最適化が実行環境にもおおいに依存するにもかかわらず，計算機システムが高度になり複雑化してきているため，最適化に際して必要なパラメータを理論的に予測することが難しくなってきていることから重要になってきている．これについては最適化に適当なパラメータを導入し，実行環境で試行・計測してパラメータを求める実行時プロファイリングという方法がある．

1.1 本発表の内容

本発表では FFT[3] を例にとり，両者を組み合わせた最適化をテンプレート・メタ・プログラミング，特に生成的プログラミング・スタイルで実装し，アクティブ・ライブラリ化することを目指した実装を行った．そしてその実装の効果を検証した．本発表で採用した最適化の基本的な考え方そのものは **FFTW**[6] として紹介され，すでに実績をあげているものであるが，本発表はこの基本的な考えがテンプレート・メタ・プログラミング，特に生成的プログラミング・スタイルで記述されるアクティブ・ライブラリとしてどこまで実現でき，実効性があるかを検証した．

具体的には，C++テンプレートの機能を用い，メモリ階層を意識した最適化を行う構造と実行時プロファイリングに基づくその適応的なチューニングを実装した．これは以下の 3 つの段階からなる．

(1) あらかじめ定めた特定の短い長さ（サイズ）の列に対する最適化 FFT コードをサイズごとに生成して「核コード」とする．

核コード生成 テンプレート・メタ・プログラミングによる既定サイズ向けの最適化コードの生成，ここで行う最適化は以下のとおり：

- FFT のループの展開
- 三角関数値の静的計算

(2) 前段で生成したサイズごとの核コードの実行時間計測．

計測 次段のテンプレート・メタ・プログラムで利用可能な出力形式としてテンプレートで記述されるコードを生成．

(3) ユーザに与えられたデータの大きさ N について計測結果から最適なコードを生成する．

分割計画 各サイズで生成済みの核コードに対する実行時間データを利用し，動的計画法によって，与えられた N に対する FFT の最適な分割を計算する．

合成 算出した分割計画に基づき適切な核コード呼び出す．

以上の実装とについてその実行時間を計測して評価した．

以下の各節の内容は次のとおりである．2 章では C++テンプレート・メタ・プログラムの概要と性質，生成的プログラミングとの関係について述べる．次いで 3 章では対象となっている FFT の概要と性質と最適化のアイディアについて述べ，4 章では FFT を対象に今回行った最適化手法とそれを実現するライブラリの構成の詳細について述べる．続く 5 章で実行性能の計測結果を示すとともにその評価を行う．6 章で関連研究について検討した後，7 章でまとめと今後の計画について述べる．

2. テンプレート・メタ・プログラミング

C++言語には**ジェネリック・プログラミング**を可能にするために**テンプレート**という機能が組み込まれている．ジェネリック・プログラミングとはプログラム内の型をパラメータ化（**型パラメータ**）して同じ構造を持つ一群のプログラムを 1 つのジェネリックなソースで記述することである．このようなジェネリックなプログラムを実行するには型パラメータに具体的な型を割り当てて**インスタンス化**する必要がある．インスタンス化によってパラメータ型が定まると，その具体的な型でさらに内部のジェネリックなデータ構造，関数や演算を再帰的にインスタンス化したり，具体的な個々の型について多重定義された関数，演算を呼び出したりすることで実際に実行できるようになる．C++では**図 1** ように通常の宣言の先頭に型パラメータ（図 1 の例では様々な精度，実装の実数型を表す Float）の導入を宣言するテンプレート構文でクラスや関数についてジェネリックなコードを記述できる☆．

C++のテンプレート機能は以下のような特徴を持つ：

- 静的（コンパイル時）インスタンス化
- パターン・マッチング機能（特別バージョン，部

☆ C++では struct はすべてのメンバが公開されているような class と同義．テンプレート・メタ・プログラミングでは公開メンバを通じて情報を受け渡すため公開メンバが多用される．公開/非公開を細かく制御すればコンパイル時計算において「関数」ローカルな「変数」を表現できるが，簡単のため本発表では統一的にクラスの表現に struct キーワードを用いている．

```
template<typename Float>
struct complex
{
  //Float is type parameter
  Float re;
  Float im;

  complex(Float real, Float imaginary)
    : re(real), im(imaginary) {;}
};
template<typename Float>
complex<Float> operator+(complex<Float> a
                         , complex<Float> b)
  {return(a.re + b.re, a.im + b.im);}
```

図 1 ジェネリックな複素数クラスの例
Fig. 1 Generic complex number class sample.

分特別バージョン）

- 型パラメータに整数パラメータを含むことが可能

コンパイラが行う静的インスタンス化によって実行時のインスタンス化などの柔軟性は若干犠牲となる代わりにオーバヘッドのほとんどない効率の良い実行コードが生成される．加えてパターン・マッチング機能と整数パラメータを組み合わせることで単なるジェネリック・プログラミングではなく，静的（コンパイル時）に値を評価して特化済みの実行効率の良い実行コードを生成するなどといった**生成的プログラミング**が可能になる．

たとえば**図 2** は再帰的なクラス・テンプレート定義を利用して階乗をコンパイル時に計算する例である．`template<int N> struct Factorial` の result に見られるように値はクラスの static な定数に保持され（すなわちクラス定数の定義はメタレベルでの整数変数の定義となる），Recursion に見られるように「関数」の閉包への参照はクラスの typedef で行われる（すなわち typedef の定義はメタレベルでの関数閉包変数の定義となる）．終了条件はパターン・マッチング機能によって `N == 1` 向けの特別バージョンが呼び出されることで検査される．特定の整数定数 N についてインスタンス化終了後の `Factorial<N>::result`（図 2 の例では `Factorial<3>::result`）は単なる整数定数となる．

また**図 3** は再帰的なクラス・テンプレート定義を利用してベキ乗のコードを生成する例である．インライン・キーワードが指定されたクラス・メンバ関数の呼び出しがインスタンス化後にインライン展開されることでコードが生成される．終了条件はパターン・マッチング機能によって `N == 1` 向けの特別バージョンが呼び出されることで検査される．特定の型と整数定数 N についてインスタンス化終了後のインライン・クラス・メンバ関数の呼び出し `Power<T, N>::exec(x)`（図 3 の例では `Power<double, 3>::exec(x)`）は呼び出し側で展開され `x*x*x` と等価なコードとなる．

以上のようにクラスに対するテンプレートをあたかも再帰呼び出しとパターン・マッチング機能を関数型

```
//Generic version
template<int N>
struct Factorial
{
  typedef Factorial<N-1> Recursion;
  static const int result
    = n * Recursion::result;
};
//Special version for N == 1
template<>
struct Factorial<1>
  {static const int result = 1;};
//Equivalent to constant 6
Factorial<3>::result;
```

図 2 階乗のコンパイル時計算
Fig. 2 Factorial computation in compile time.

```
//Generic version
template<typename T, unsigned N>
struct Power
{
  typedef Power<T, (N-1)> Recursion;

  static inline T exec(T x)
    {return (x * Recursion::exec(x));}
};
//Special version for N == 1.
template<typename T>
struct Power<T,1>
{
  static inline T exec(T x)
    {return (x);}
};
//Equivalent to code "x*x*x"
Power<double, 3>::exec(x);
```

図 3 ベキ乗コードの生成
Fig. 3 Code generation for power.

言語における関数のように見なして様々な計算とそれを利用したコード生成をコンパイル時に行うようプログラムすることが可能になる．ここではクラス・テンプレートをコンパイル時に処理される「関数」と見なし，クラス static な整数定数の定義と参照を「整数変数」の定義と参照，クラス内の typedef の定義と参照を「関数閉包への参照を保持する変数」の定義と参照，クラス・テンプレートのインスタンス化を「関数」の評価，定数伝播をコンパイル時の「整数計算」というようにそれぞれ見なすことができる．以上のように見なすことができると同時に，元々テンプレートをはじめとするそれらの機構は型として静的なデータ構造を組み立てる道具でもある．つまりテンプレート・メタ・プログラミングはプログラムをデータ構造へ写像する手段と見ることができる．組み立てられたテンプレートの呼び出し関係は非循環有向グラフ状のデータ構造を表現できる．このデータ構造を処理してその結果からたとえば図 3 のように生成されるバイナリを制御できると考えれば，テンプレート・メタ・プログラミングはコード変換であると見なせる．表記の煩雑さなど細かな使い勝手はともかく「関数」としてのテンプレートには計算の記述能力は最低限備わっているので，理論上は実に多様なコード変換が記述できることになるが，実際的であるかどうかということは各事例について個別に検討が必要となる．

「関数」としてのテンプレートは再帰とパターンマッチング以外にも以下のような特長を持っている．

副作用なし 変数はクラス・メンバ整数定数の初期化子や typedef による型の別名といった構文で定義できるだけで代入などの副作用はない．

値のメモ機能 同じ引数でインスタンス化されたテンプレートは一度だけインスタンス化されて以降のはその結果が参照される．

遅延評価 テンプレートは定義が実際に必要となる（クラス・メンバの参照やクラス・メンバ関数の呼び出し）までインスタンス化されない．

高階関数 テンプレートはテンプレートを引数として受け取ってテンプレートを結果として返すことができる．

多値 複数の値（整数定数や型）の組を結果とすることができる☆．

これらの特長は元来クラス・テンプレートがジェネリックなクラスの定義を目的としていたことから必然的に備わったものである．

一方クラス・テンプレートを「関数」として利用することは C++開発時に想定されていた利用法ではない☆☆ため欠点も多数ある．

- 整数しかデータ型がなく,「関数」内では浮動小数点演算も配列もオブジェクトも利用できない．
- 計算の中間結果が不要になっても記憶領域は開放されない．
- 今のところ標準的な処理系ではすべてのパラメータが具体化された特別バージョン以外の分割コンパイルはできない．
- C/C++本来の関数と構文・挙動があまりに異なり，煩雑で可読性がきわめて悪い．
- テンプレートのインスタンス化中は入出力できない．
- テンプレートのインスタンス化中の動作はデバッガでデバッグできない．
- テンプレートのインスタンス化中の動作はプロファイリングできない．

これらの欠点のため実際的なコンパイル時計算を実行するためには様々な工夫が必要となる（これらの欠点と対する工夫のいくつかについては 4.1 節，4.2 節，4.5 節で後述する）が，現状ではコンパイル時計算や生成的プログラミングが可能なプログラミング言語の中で：

- 安定したフリー/商用のコンパイラが存在している，
- 多様な環境に対して移植されている，
- 実用レベルの標準的な最適化は大概組み込まれている，
- 標準規格が制定されている．

このように安定して利用できる条件が整っているものはほかにない．

3. FFT の構造と最適化

長さ n の複素数の列 $X_n(n=0,1,\cdots,N-1)$ から式 (1) で定義される複素数の列 $X^k(k=0,1,\cdots,N-1)$ を求める変換を**離散 Fourier 変換**（DFT）という．

$$X^k=\sum_{n=0}^{N-1} e(\frac{kn}{N})X_n \ (k=0,1,\cdots,N-1) \quad (1)$$

$$e(\xi)=\exp(-2\pi i\xi) \quad (2)$$

この定義に従って素朴に計算すると計算量は $O(N^2)$

☆ より正確には関数に返り値がないかわりに関数の閉包にアクセスできる．

☆☆ クラス・テンプレートのコンパイル時関数としての利用は再帰的な型定義を試みた際に出力されたエラーメッセージによって「発見」された[14]．

になる．しかし $N = 2^n$（n は 1 以上の自然数）である場合は，文献 3) によって紹介され現在広く普及している**高速 Fourier 変換**（FFT）アルゴリズムを利用すれば，$O(N\log_2(N))$ になることが知られている．その基本となるアイディアは DFT の持つ数学的性質を利用して長さ N の $DFT(N)$ を，N の因数 N_i を長さとする $DFT(N_i)$ から求めることである．

列の長さ N が 2 つの自然数の積 $N = N_1N_2$ であるとき式 (3)，(4) と (5) によって $DFT(N)$ を計算することができる．

$$X_{n_1}{}^{k_2} = \sum_{n_2=0}^{N_2-1} e(\frac{k_2n_2}{N_2})X_{n_1n_2}; \quad (3)$$

$$(n_1 = 0,\cdots,N_1-1; k_2 = 0,\cdots,N_2-1;)$$

$$\widetilde{X_{n_1}{}^{k_2}} = e(\frac{k_2n_1}{N})X_{n_1}{}^{k_2} \quad (4)$$

$$X^{k_1k_2} = \sum_{n_1=0}^{N_1-1} e(\frac{k_1n_1}{N_1})\widetilde{X_{n_1}{}^{k_2}}; \quad (5)$$

$$(k_2 = 0,\cdots,N_2-1; k_1 = 0,\cdots,N_1-1;)$$

これらの式は，式 (3) のように長さ N_2 の DFTN_1 個で変換後に式 (4) で定義される**回転因子**（twiddle factor）を列の各項に掛け，その結果を式 (5) のように長さ N_1 の DFTN_2 個で変換するということを意味している．この際，$DFT(N_2)$ の計算量を T_2，$DFT(N_1)$ の計算量を T_1 とすると $DFT(N_1N_2)$ の計算量は $N_1T_2 + N_2T_1$ となる．$N = N_1N_2 = 2\cdot4$ についての合成の例を図 5 に示す（ただし図 5 では $DFT(2)$ および $DFT(4)$ としてそれぞれ $FFT(2)$ と $FFT(4)$ があるものとして利用している）．

$N = 2^n$ の場合に $N = 2(2(2\cdots(2\cdot2)\cdots))$ と考えて☆再帰的に式 (3)，(4) と (5) の関係を利用したアルゴリズムが FFT であり，その計算量は $N\log_2(N)$ となる．計算の最初の段階でビット・リバース・オーダと呼ばれる順序に X_n を並べ換えておくと入力から中間結果を配列上で上書きしながら計算して最終結果 X^k が $k = 0, 1, \cdots N-1$ の順で得られる☆☆．$N = 2^8$ としたときの計算の様子を**図 4** に示す．

3.1 FFT の問題点

FFT アルゴリズムは計算量的にはよく工夫されているが，図 4 の配列アクセス・パターンを見ても分かるとおり様々な幅で何度も配列を走査するため参照の局所性に欠ける．現代の多くの計算機システムではレジスタ〜1 次キャッシュ〜2 次キャッシュ〜メイン・メモリといったように少容量の高価な高速メモリと大容量の安価な低速メモリを組み合わせて**記憶階層**を構成することによってメモリ・アクセスの速さとハードウェアのコストのトレードオフを緩和している．このため，あるところまで N が大きくなると段階的に計算の時間効率が悪化するという問題がある．

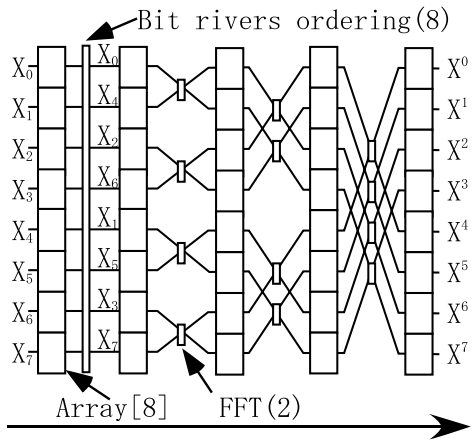

図 4 FFT の配列アクセス・パターン（$N = 8$）
Fig. 4 Array access pattern of FFT ($N = 8$).

3.2 FFT の最適化

3.1 節の問題を解決する方法の 1 つは FFTW で提案されたもので式 (3)，(4) と (5) の性質を利用して短い列に対する FFT の組合せで必要な長さの FFT を計算することである．ここで短い列に対する FFT は自前のバッファに値をコピーしてから計算を行うので参照の局所性が高まる．**図 5** は $N = N_1N_2 = 2\cdot4$ についてこのような組合せを行った場合の計算の様子を示す．

図 5 の 1 段目の FFT はこの組合せによる「合成 FFT」に置き換えてもよく，そうして入れ子構造にすることで任意の $N = 2^n$ の形式の任意の長さの列を処理する FFT を組み合わせて必要な長さの「合成 FFT」を作り出すことができる．「合成 FFT」は純粋な FFT アルゴリズムに比べて合成の基本単位となる FFT コンポーネント（以降核コードと呼び，核コードが処理する列の長さを核コードのサイズと呼ぶ）の合成に関わる処理（たとえば回転因子の計算と掛け合わせ）の分だけ計算量（浮動小数点乗算回数）が増加し，コピーや並べ換えなどのオーバヘッドも増えるが，核コードが記憶階層の性能を十分に生かした高速なコードであればそれを挽回できると期待でき，実際 FFTW ではそれが達成されている．FFTW の場合はさらに核コード内部でもキャッシュの階層を生かす工夫として計算順序のスケジューリングなどが行われている．

☆ $N = ((\cdots(2\cdot2)\cdots2)2)2$ と分解する方法もある．
☆☆ 後で並べ換える方法もある．

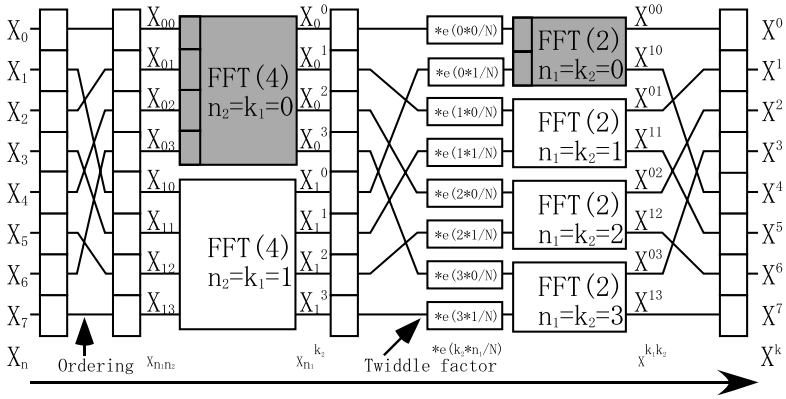

図 5 FFT の合成（$N = N_1 N_2 = 2 \cdot 4$）
Fig. 5 Composition of FFT ($N = N_1 N_2 = 2 \cdot 4$).

4. 実　　装

3.2 節のような最適化を実現するため本ライブラリは FFTW を参考にしてテンプレート・メタ・プログラミングでインストール時にコード生成と計測，コンパイル時に合成を行う．以下この章ではその実装の詳細について述べる．

まず参考にした FFTW との比較をしながら行っている最適化について概観する．まず全体の構成として，一般に FFTW ではインストール時には各 N について生成済みの核コード（FFTW では Codelet と呼ばれる）の C コードが配布されており，配布されていない構成のコードを追加するには別に ML 言語で書かれた codelet ソース生成用パッケージを取り寄せて生成する．codelet の C ソース・コードは genfft [5] という ML で記述された特殊なコンパイラに核コードのサイズ N を与えて生成させる．genfft は前もって ML 言語で記述された基本的ないくつかのアルゴリズム（Cooley-Tukey($N = 2^n$) [3]，Rader($N =$ 素数) [12] など）についてデータ・フローを表す DAG（非循環有向グラフ）を生成し，数学的知識（たとえば一定の基準より小さい実数は 0 に置き換えるなど）に基づく書き換え規則で書き換えて不要な演算を削除することと，演算のスケジューリングでキャッシュを有効利用することで実現している☆．この FFTW の専用コンパイラ genfft は以上の処理によって新たなアルゴリズムを発見したとされている．インストール時には以上のようにしてすでに生成されている C で書かれた codelet がコンパイルされる．そうして作られた codelet は実行時に計測され，その結果に基づいて実行時にユーザが求める最終的な変換コードが合成される☆☆．

一方本ライブラリはインストール時に C++テンプレートで記述されたソースから C++で直接各サイズ N ごとの実行コード生成し，それらの計測まで行っておき，ユーザのアプリケーションのコンパイル時に計測結果を取り込んでユーザが求める最終的な変換コードを合成する．実行時には実際の変換だけが行われる．核コードのソース・コードは C++のテンプレートを利用して直接記述されている．現在，核コードのコードは Cooley-Tukey アルゴリズムのみであり，それに対する最適化はループの展開と配列内データへのオフセットと回転因子のコンパイル時計算のみである．現在はどの環境でどの手法がどこまで有効であるかが明らかでないため，核コードは FullStatic 版核コード（ループを展開したうえで可能な限り浮動小数点数をコンパイル時に計算した FFT），LoopStatic 版核コード（ループを展開し添え字操作だけを静的に計算した FFT），Dynamic 版核コード（ローカルなバッファにコピーをして計算する以外まったく教科書[10] どおりの単なる Cooley-Tukey 法で計算される FFT）の 3 種類作成し，それらを各サイズ N（N は 2 のべき乗で，FullStatic 版と LoopStatic 版は $N \leq 64 = 2^6$，Dynamic 版は $N \leq 32768 = 2^{15}$）について展開し，計測結果から各 N について最も速い版を選んで利用している．$N > 64$ のサイズの Dynamic 版核コードについては，最適化への寄与は核コードが丸ごとキャッシュ・ブロックになっていることだけである．核コー

☆ このスケジューリングは cash oblivious algolithm [7] を実現しており，キャッシュ・サイズなどのパラメータなしにほとんど最適なコードを生成する．

☆☆ FFTW では実行時に行う計測結果に関する情報と codelet の組合せに関する情報を特別なデータ構造で管理し，キャッシュすることで FFT の反復実行の際の実行時オーバヘッドを削減する工夫がなされている．

表 1 FFTW との比較
Table 1 The comparison of this library and FFTW.

	FFTW	本ライブラリ
核コード生成のタイミング	各 N に対する C ソースは配布時に生成済み，ユーザが生成して追加インストールすることも可能．バイナリへのコンパイルはインストール時	インストール時
計測のタイミング	実行時	インストール時
合成のタイミング	実行時	コンパイル時
核コードで利用するアルゴリズム	Cooley-Tukey（$N = 2^n$），Rader(N = 素数)	Cooley-Tukey（$N = 2^n$）のみ
核コードの最適化	アルゴリズムから自動抽出したデータ・フロー・グラフを利用した最適化（グラフ書き換え規則による不要な演算の省略，共通部分式の削除，スケジューリング）	ループの展開，定数のコンパイル時計算

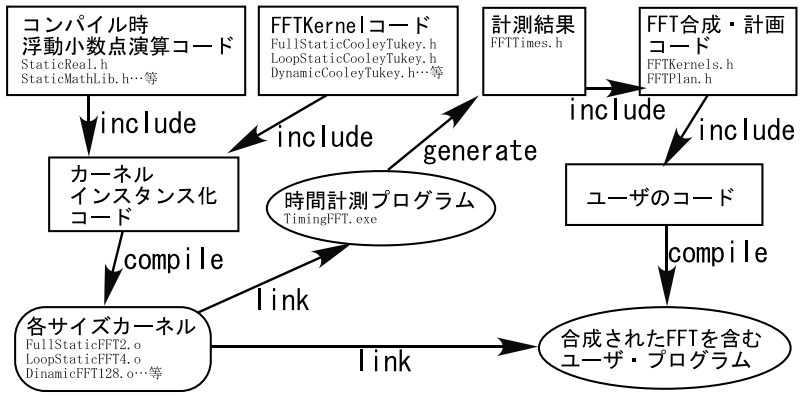

図 6 ライブラリ概要
Fig. 6 Overview of the library.

ドの生成に関して現時点では限られた最適化しか実装していないが，今後より大きなサイズの核コードに対する不要な演算や共通部分式の削除，計算順序のスケジューリングなどを実現することを計画している．

以上を表にまとめたものが**表 1** である．

4.1 ライブラリの構成

ライブラリの構成の概要を**図 6** に示す．テンプレート・メタ・プログラミングされたヘッダファイルはインクルードされたソース・コードがコンパイルされる際にインスタンス化されコンパイル時計算とコード生成が行われる．

素朴にユーザ・プログラムに FFT 核コードコードをインスタンス化するとコンパイル時間が大変長くなるため，あらかじめ特化したコードを分割コンパイルして作っておき，ユーザ・プログラムではそれらをリンクするだけにする．これによりユーザ・コードのコンパイル時間は短縮できるがあらかじめ作っていないサイズの核コードは利用できなくなる．一方，あらかじめ作成しておいた各サイズの核コードについては時間計測プログラムで核コードの実行時間を計っておき，そのデータはヘッダファイルの形で出力する☆．ここまでの作業はライブラリのインストール時に行う．この作業は make プログラムによって自動的に実行できる．

```
//Generic version
template<int N>
#include"FFTPlan.h"
//Planning
typedef typename Plan<32>::division FFT;
//Transform
FFT::transform(data, data);
```

図 7 ユーザ・コードの例
Fig. 7 User code sample.

図 7 に示すように，ユーザ・プログラムでは分割計画と合成を行うテンプレート・メタ・プログラムを取り込み，計画テンプレートにサイズを指定してその結果となるようインスタンス化された分割仕様のテンプレートの static メンバ関数 transform() を呼び出すように作成するだけで必要な変換が実行できる．このと

☆ クロス開発の場合は最低限，時間計測プログラムをターゲット計算機上で実行できれば必要な計測データが得られる．

き，分割計画・合成のテンプレート・メタ・プログラムを含むヘッダ・ファイルは核コードの実行時間計測データを含むヘッダ・ファイルをインクルードすることで計測結果を利用する．

以上に関して FFTW[6] との違いは核コード生成も分割計画もテンプレート・メタ・プログラムで記述されていることである．FFTW では ML で記述された専用コンパイラで C の核コード・コードを生成している．本発表で紹介しているライブラリではユーザが自身のプログラムをコンパイルするのに用いるのと同じ標準コンパイラだけがあればよく，そのような外部プログラムは不要である．

以下の節ではコード生成（ループの展開と関数値のコンパイル時計算），計測，分割計画と FFT の合成についてこのライブラリの詳細を述べる．

4.2 コード生成

本発表で紹介する最適化手法を実現するには各サイズに特化した効率の良い核コードを自動的に生成する必要がある．たとえば 4 章冒頭でも述べたように，FFTW[6] では ML を用いて記述した専用コンパイラ genfft[5] を利用する．一方，本発表で紹介するライブラリは原状では FFT アルゴリズムについて直接ループの展開とコンパイル時計算をテンプレート・メタ・プログラミングで記述することで特定の N に特化したコードを生成する．

ループの展開は FFT の本体部分とビット・リバース・オーダへの並べ換えとローカルバッファへのコピー，ローカル・バッファから元の配列への書き戻しについて行っている．ただし，コードの見通しを良くするために関数の再帰呼び出しを直接使うのではなく環境を渡しながら再帰呼び出しを行うような通常のプログラミングでの for 文にあたる汎用の制御構造を**図 8** のようにテンプレートで記述し，それを利用している．この環境となるクラスにはクラスの整数定数定義と typedef 定義が含まれ，それぞれがコンパイル時計算（メタレベル）での整数変数の定義と関数閉包変数の定義を表している．FullStatic 版核コードの詳細については FFT の中心となる 3 重ループ部分のソースコードとその解説を A.1 節に付しているが，要約するとテンプレートを関数と見なして環境を表すクラスを For テンプレートを介して渡しながら再帰呼び出しを記述するとパラメータとクラス変数と typedef の定義/参照で連鎖した一連のデータ構造ができる．これをたどるよう設定された static メンバ関数を inline で呼び出せば「ループ」を展開したバイナリが得られるということである☆．

```
template <typename InitialEnv
  , template<typename E> class Cond
  , template<typename E> class Body>
class For
{
  template<typename Env
    , bool cond>
  struct Iterate
  {
    typedef typename Body<Env>::Env
      NextEnv;
    static const bool nextCond
      = Cond<NextEnv>::cond;
    typedef typename Iterate<NextEnv
      , nextCond>::Result Result;
  };
  template<typename Env>
  struct Iterate<Env, false>
  {
    typedef Env Result;
  };
  static const bool initialCond
    = Cond<InitialEnv>::cond;
  public:
  typedef typename Iterate<InitialEnv
    , initialCond>::Result Result;
};
```

図 8 For テンプレートの定義
Fig. 8 Definition of “For” template.

4.3 コンパイル時浮動小数点演算

FullStatic 核コードではループの展開に加えて，回転因子の計算など浮動小数点演算をコンパイル時計算する必要がある．しかし，テンプレート・パラメータとしては浮動小数点型が利用できないし，C/C++コンパイラの仕様では浮動小数点数について定数畳み込みの実行を保証していない☆☆．また FFT の回転因子の計算は三角関数の呼び出し（あるいはそのための表となる配列の参照）を含むので一般には畳み込みの対象にならないことも多い．さらにいえば FFT の回転

☆ 現時点ではできあがったデータ構造から直接的にコード生成を行っているが，静的データ構造を別のテンプレートにパラメータとして渡してさらに処理させることも理論上は可能であり，複雑な変換を行うためにそのような方法が利用できる可能性はある．

☆☆ C/C++の仕様には浮動小数点リテラルはあるが浮動小数点定数式は存在しない．

```
template<typename prec, int s, int e
  , typename f>
struct StaticReal
{
  typedef prec precision;
  typedef RealSpec<prec> Spec;
  typedef typename Spec::Digit Digit;
  static const int nDigitsEX
    = Spec::nDigitsEX;
  static const bool isz
    = StaticIsZero<Digit
    , nDigitsEX,f>::result;
  static const int sign = isz? 0
    : (s > 0? 1: (s < 0? -1: 0));
  static const int exp = isz? 0: e;
  typedef f frac; //64bit unsigned integer
};
```

図 9 浮動小数点数の定義
Fig. 9 Definition of floating point number.

因子の計算はループの添え字に依存するので，コンパイラが仮に浮動小数点数値の畳み込みを行うとしても必ずループの展開後に畳み込みが行われるようにしなければうまくいかない．以上の理由からテンプレート・メタ・プログラミングで確実に狙いどおりの最適化を実現するためには浮動小数点演算が必要となる．そのため組み込みの double と同等の精度でコンパイル時に計算が可能な浮動小数点演算（およびそれに必要な倍長整数演算）を実装した．**図 9** に浮動小数点数を表現するテンプレートの定義を示す．

この定義では浮動小数点型のコンストラクタとなる関数をクラス・テンプレートで表したコンパイル時「関数」として表現し，その具体的な浮動小数点数値をクラス・テンプレートのインスタンスである整数定数の組からなるクラスで「関数」の閉包として表現する．この「値」は typedef で定義された型名に結び付けることができる．つまり typedef をコンパイル時の浮動小数点変数の定義に利用して，その typedef で定義された型名をコンパイル時には変数名として，実行時には定数名として参照することができる．コンパイル時浮動小数点数の計算をするプログラムはテンプレートのパラメータを介した参照関係のデータ構造に写像され，その参照関係に沿ってコンパイラが整数定数の畳み込みを行うことで整数定数の組で表現された浮動小数点「定数」が得られる．このようにして最終的なバイナリにはまったく中間結果は残らず，参照された「値」だけが残る．また，この定義ではコンパイル時の計算で新たな浮動小数点数値が現れるたびにテンプレートのインスタンスが増えることになる．一見すると型が爆発しそうであるが，結局のところ計算量に比例する数の型しかできないのでそれほど爆発はしない．ただ，計算ではループ展開できるものはそれを手で展開して極力再帰を使わないようにしてコンパイル時のコストを下げる工夫をしている．ちなみに，FFT では整数でループが制御されており浮動小数点数は条件分岐に利用されないが，この定義のコンパイル時浮動小数点数はコンパイル時の条件分岐で利用することも可能である．

具体的な演算の実装の例として**図 10** に乗算の定義をあげる．`StaticMulReal<a,b>` で浮動小数点数を表すテンプレート特別バージョンを a と b に渡すと 0 か否かが調べられ，一方でも 0 ならば `StaticMulRealZeroCheck<a,b,true>` が呼び出されて 0.0 相当のテンプレート特別バージョンが result 返る．そうでない場合は少数部を計算する整数演算のテンプレート，次いで正規化を行うテンプレートが「呼び出される」．

このほか，浮動小数点数については π などいくつかの定数が定義してあるほか，static const int といった整数定数によってのみ初期化できる．現在この浮動小数点数について実装済みの演算は**表 2** のとおりである．Int と prec は精度を示す型引数で前者は int や long などをとり，後者は double や float などをとる．x，a，b には図 9 で定義される浮動小数点数を渡し，結果を得るには static メンバ result を参照する．実行時に利用できる浮動小数点型の値を得るには `StaticFromReal<a>::get()` を利用する．`StaticFromReal<a>::get()` は a と同等の double（あるいは float，初期化時に prec で指定されているもの）のビットパターンをコンパイル時に整数演算で組み立てて実行時に union{ } を介して double に変換して渡す．

このように型をテンプレートに，その型に属するオブジェクトをそのテンプレートの特別バージョンにマップし，「関数」としてのテンプレートの型引数に値として渡し，typedef で変数宣言するというテクニックはテンプレート・メタ・プログラミングで数値のようなデータ型を定義するために広く利用できると考えられる．

現在は採用していないが，コードの可読性を向上させるために Expression Template 技法[15] を利用して演算子で計算式を定義できるようにすることを計画中

```
template<typename a, typename b
  , bool a_is_zero>
struct StaticMulRealZeroCheck
{
  typedef typename a::precision prec;
  typedef RealSpec<prec> Spec;
  typedef typename Spec::Digit Digit;
  static const int nDigitsEX
    = Spec::nDigitsEX;
  static const int nFracBits
    = Spec::nFracBits;
  static const int exp = a::exp + b::exp;
  typedef StaticMulEXFraction<Digit
        , nDigitsEX,typename a::frac
        , typename b::frac> mul;
  typedef StaticNormalizeEXFraction<Digit
        , nFracBits, nDigitsEX
        ,false, typename mul::result> n;
  typedef StaticReal<prec
        , a::sign * b::sign
        , exp + n::shifts
        , typename n::result> result;
};
template<typename a, typename b>
struct StaticMulRealZeroCheck<a,b,true>
{
  typedef typename a::precision prec;
  typedef RealSpec<prec> Spec;
  typedef typename Spec::Digit Digit;
  static const int nDigitsEX
    = Spec::nDigitsEX;
  typedef StaticReal<prec,0,0
    ,StaticFraction<Digit
      ,nDigitsEX,0> > result;
};
template<typename a, typename b>
struct StaticMulReal
{
  typedef
    typename StaticMulRealZeroCheck
    <a,b,(a::sign == 0 || b::sign == 0)>
      ::result result;
};
```

図 10 浮動小数点乗算の定義
Fig. 10 Definition of floating point multiplication.

表 2 StaticReal<> に対する演算
Table 2 Operations of StaticReal<>.

種別	構文
定数 0.0	StaticRealConst<prec>::zero
定数 1.0	StaticRealConst<prec>::one
定数 2.0	StaticRealConst<prec>::two
定数 $\frac{1}{2}$	StaticRealConst<prec>::half
定数 ϵ	StaticRealConst<prec>::eps
定数 π	StaticRealConst<prec>::pi
定数 $\frac{\pi}{2}$	StaticRealConst<prec>::half_pi
実行時表現への変換	StaticFromReal<a>::get()
整数定数による初期化	StaticIntToReal<Int,a>::result
整数部取り出し	StaticModReal<a>::result
大小比較	StaticGtReal<a,b>::result
符号反転	StaticNegateReal<a>::result
絶対値	StaticAbsReal<a>::result
加算	StaticAddReal<a,b>::result
減算	StaticSubReal<a,b>::result
乗算	StaticMulReal<a,b>::result
除算	StaticDivReal<a,b>::result
モジュロ演算	StaticModuloReal<a,m>::result
関数 $sin(\frac{\pi}{4}x)$	StaticSinQ<x>::result
変数定義/参照	typedef で行い、型名が変数名となる

である．

4.4 計　　測

様々な最適化ポリシ，サイズについて生成された複数の核コードを組み合わせる際に個々の核コードの性能と合成時のオーバヘッドを予測する必要がある．しかしレジスタの数，キャッシュのサイズや方式をはじめ様々な要因が絡むので予測を行うことは困難である．そこで実際に走らせて時間を計ることで性能を知るという手段をとる必要がある．

4.2 節のように処理する列の長さ別に特化して生成した核コードおよび合成時の回転因子を計算するコードを時間計測用メイン・プログラムにリンクする．時間計測プログラムは擬似乱数で生成した列 X_n について規定の回数 FFT を繰り返し計算した合計時間（clock() で計られるクロック数）を計測する．テンプレート・メタ・プログラミングでは 2 章でも述べたように入力もできなければ配列でテーブルを表現することもできないため，計測結果は**図 11** のような形式で核コードの処理する列の長さパラメータとするテンプレートの特別バージョンとして出力する☆．こうすることによってパターン・マッチング機能を利用して列の長さをキーにして計算時間を検索できるようになる．分割プランを計算するテンプレート・メタ・プログラムは

☆ ここでは簡単のため省略したが実際のテンプレートには核コードで利用する浮動小数点数型や整数型を指定するパラメータもある．以下 Plan<> や SearchPlan<>, Divide<>, Solve<> も同様．

```
// max size of kernel size
static const int MaxN = 32768;
template<int N>
struct FullStaticCooleyTukeyKernelTime
{
  /* general template(place holder)
  for specialized templates. */
  static const clock_t Ticks = -1;
};
template<>
struct FullStaticCooleyTukeyKernelTime<2>
{static const clock_t Ticks = 812;};
template<>
struct FullStaticCooleyTukeyKernelTime<4>
{static const clock_t Ticks = 2312;};
template<>
struct FullStaticCooleyTukeyKernelTime<8>
{static const clock_t Ticks = 7000;};
template<>
struct FullStaticCooleyTukeyKernelTime<16>
{static const clock_t Ticks = 17744;};
```

図 11 テンプレート特別バージョンによる計測結果の表現
Fig. 11 Timing result expressed as template special version.

このヘッダファイルを `#include"FFTTimes.h"` のように取り込んで利用する．

FFTW では核コード速度の計測の際，ハードウェアが備える Cycle Counter の機能が利用できるときはそれを利用して高精度のタイマの代わりにする．本ライブラリではそのようなハードウェア依存の方法は用いないことにしていたので，タイマの分解能が低く，特に小さな核コードの実行時間が直接には計測できない．これをカバーするため適宜反復回数を増やしてその合計時間を計り，反復回数を加味した計測結果として利用しており，特に小さな核コードほど反復回数を大きくしている．このことにより計測にかかる時間は延びるが，本ライブラリでは時間計測はインストール時に行うのでコンパイル時にも実行時にもそのオーバヘッドはかかわらない．

4.5 計画と合成

実際に最適化された計算を実現するためには計測結果から与えられた N について最適な核コードの組合せを求め，実際にその組合せを実現する必要がある．

分割計画は `Divide<32,Divide<8,solve<2>>>` のように入れ子になったテンプレート型で表現する．この型は**図 12** のような分割を表現しており，このクラス・テンプレートのクラス・メンバ関数 `transform(in, out)` を呼び出せば実際にこの分割計画に従って変換を実行する☆．

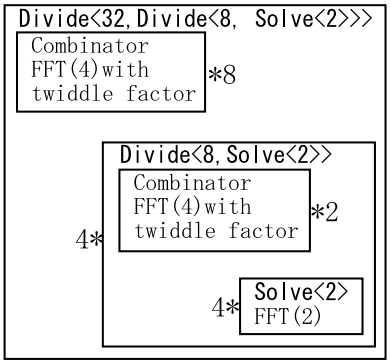

図 12 FFT の再帰的合成（$N = 4 \cdot 4 \cdot 2$）
Fig. 12 Recursive composition of FFT ($N = 4 \cdot 4 \cdot 2$).

`Solve<>::transform()` は FFT を単に実行し，`Divide<>::transform()` は図 5 のようにして変換を合成する．この際，1 段目の変換はさらに子（分割計画の型パラメータで内側に入れ子になっている）の transform を呼び出しその結果に回転因子を掛けてから 2 段目の FFT を実行する．

図 14 の `Plan<N>` は図 11 の計測結果に従って与えられた N について最適な分割をダイナミック・プログラミングで求める．すなわちサイズ N についての分割を決めるためには N より小さいサイズについての分割を実際に求めてそれを利用するのである．これを実現するためテンプレートのパターン・マッチング機能を探索と計算済みデータの検索の両方に用いる．具体的には**図 13** にあげた `SearchPlan<int N, int SubN>` というテンプレート・クラスによって以下のように探索を行う．`SearchPlan<N, SubN>` は N = SubN · CombN(=N/SubN)（SubN は再帰的にさらに分解されているかもしれない）についての最適な分割計画とその予想所要時間 Ticks を typedef および static メンバとして持つ．

`Plan<N>` が必要とする分割を求めるには以上のように作成した `SearchPlan<N,i>` を参照すればよい（ここで i は N/MaxN と N/2 のうち小さい方）．分割に必要な部分計画については `SearchPlan<N,SubN/2>` を参照すればよい．SearchPlan は SubN の小さいもの（最小は 2）へ向かって順にすべての組合せを探索する．この際 2 章で述べたメモ機能によりいったん調

☆ ここで in と out はそれぞれ入力と出力先へのポインタ．同一の場合は結果を上書きする．

```
template<typename FloatType
  , typename IntType, IntType SizeN
  , int SubN>
struct SearchPlan
{
  typedef typename SearchPlan<FloatType
    , IntType, SubN
    , SubN/2>::Plan MySubPlan;
  typedef Divide<FloatType, IntType
    , SizeN, MySubPlan> MyPlan;
  static const clock_t MyPlanTicks
    = MyPlan::Ticks;

  typedef typename SearchPlan<FloatType
    , IntType, SizeN
    , SubN/2>::Plan SearchedPlan;
  static const clock_t SearchedPlanTicks
    = SearchedPlan::Ticks;

  static const bool cond
      = ((MyPlanTicks >= 0)
      && (SearchedPlanTicks > MyPlanTicks))
      || (SearchedPlanTicks < 0);
  typedef typename SelectPlan<cond, MyPlan
    , SearchedPlan>::Plan Plan;
  static const clock_t Ticks = Plan::Ticks;
};

template<typename FloatType
    , typename IntType
    , IntType SizeN>
struct SearchPlan<FloatType
  , IntType, SizeN, 1>
{
  typedef Solve<FloatType
    , IntType, SizeN> Plan;
  const static clock_t Ticks = Plan::Ticks;
};
```

図 13 SearchPlan<> の定義
Fig. 13 Definition of SearchPlan<>.

べた組合せは新たに探索せず単に参照できる．その分割に必要なサイズの核コードが用意されていない場合 Ticks として −1 が返る．

分割を評価する実行時間予想は `Plan<N>` から得る．Plan が Solve であった場合は核コードの実行時間が

```
template<typename FloatType
  , typename IntType, int N>
struct Plan
{
  static const IntType SubN
    = (N/2 > MaxN)? N/MaxN: N/2;
  typedef typename SearchPlan<FloatType
    , IntType, N, SubN>::Plan division;
};
```

図 14 Plan<> の定義
Fig. 14 Definition of Plan<>.

そのまま予想時間になり，`Divide<N,SubN>` の場合は N/SubN 個の大きさ SubN のサブ Plan の合計予想時間と SubN 個の大きさ N/SubN の核コードの合計予想時間，それに必要な回転因子の計算予想時間の和で表される．FFT の合成に関わるコピーのオーバヘッドは考慮していない．また分割計画の探索は生の実行時間（核コードを規定の回数を実行した際に計測されたクロック数）で行っている．これはテンプレートでは整数データの方が扱いやすいからであるが，これだと大規模な問題の計画で桁あふれが発生する恐れがある．将来的には何らかの改良が必要だと考えられる．

現状では `Plan<N>` の N が 2 のべき乗であるという制約があるが，ダイナミック・プログラミングで計画を計算するうえでこれは本質的ではない．しかし現時点で実装している核コードは Cooley-Tukey アルゴリズムによるものだけなので，合成結果も 2 のべき乗となる．また，一般の因数分解は計算コストもそれなりにかかるのに対して 2 のべき乗という仮定の下では因数分解は自明であり探索空間を容易にテンプレートで表現できるが，2 のべき乗以外も扱うとなるとテンプレートで表現して探索すべき空間も広くなるので現時点ではこの制約は妥当であると考えている．

5. 実験結果

現在，ようやくライブラリのコードが動き始めたばかりで，現在準備できているハードウェアが x86 系の CPU を搭載した WindowsXP マシンのみなので適応性についても十分なデータはとれているとはいえない．以上のような状況であるが，現時点での基本的な計測データおよびそれと FFTW との比較データだけを示す．

以下のデータは Pentium4 1.90 GHz（外部クロック 400 MHz）L1 キャシュ8 KB，L2 キャッシュ256 KB，主記憶 1 GByte の計算機上で計測した．コンパイラ

表 3 核コードの実行時間
Table 3 Timing data of each kernel.

N	FullStatic		LoopStatic		Dynamic	
	Ticks	MFlops	Ticks	MFlops	Ticks	MFlops
2	126	83	64	164	1,968	5
4	188	223	1,312	32	3,252	13
8	248	507	2,496	50	4,744	27
16	1,248	269	5,248	64	7,264	46
32	3,008	279	8,000	105	12,000	70
64	8,960	225	15,040	134	19,968	101
128					34,048	138
256					71,936	149
512					159,744	151
1,024					336,896	159
2,048					735,232	161
4,096					1,662,976	155
8,192					3,588,096	156
16,384					12,795,904	94

は g++（GCC）3.4.1（mingw special）最適化オプションは-O3 である．以下で使われる単位 Flops は繰返し回数に $5\log_2(N)$ を掛けて実行時間（秒）で割ったもので 1 秒あたりに実行された浮動小数点演算の数を表し，数字が大きいほど効率良く計算が実行されたことを示す．サイズの違う核コードの速度を比較することができる指標である．

表 3 は FullStatic，LoopStatic，Dynamic の各サイズの核コードを 2^{20} 回繰り返したときの実行時間（クロック数）と MFlops である．この結果ではループの展開だけでも若干の効果はあるものの，コンパイル時計算による効果が主であることが分かる．特にサイズの小さなコードではコピーのオーバヘッドの割合が重くなるはずなのにもかかわらず☆この結果になったことは大変興味深い．現在 FullStatic と LoopStatic 核コードは 64 より上のサイズでは展開していないが，コンパイル時間の増加の割に速度が向上しない傾向があるのでこのあたりがループの展開とコンパイル時計算による核コード最適化の限界であると考えられる．一方 Dynamic 核コードは $N = 2,048$ をピークに下り坂になっている．これはおそらくデータが 2 次キャッシュからはみ出しつつあることを示しており，この計算機上ではこれ以上大きなサイズでは通常の FFT は徐々に性能が落ちることを示していると考えられる．

合成された FFT コードの計測結果を**図 15** に示す．単位と反復回数は核コードについての表と同じである．Comp は本ライブラリの MFlops 値である．FFTW は実行時に計測と合成を行うので 3 種類のテストケースを用意した．（1st.）は計測をフルに行ってから 1 回だけ計算するという FFTW にとって最悪の場合の「計画時間＋変換 1 回」の MFlops 値である．核コード自体の速度は非常に速いが初回の計測と合成にかかる時間が同サイズの FFT の計算より桁違いに多いためこのような結果になる．（repeated）はフルに計測した結果を利用して別の配列で計算する場合の「計画時間＋変換 1 回」の MFlops 値である．（pure transform）は計画時間を含まない，純粋な変換時間だけから計算した MFlops 値である．plain は教科書そのままの Cooley-Tukey アルゴリズムの MFlops 値である．

Comp では $N \leq 64$ の核コードしか最適化していないが，N=16 で追いつかれるまでは FFTW を上回る速度を出している．その一方で，サイズが大きくなるにつれて性能が低下している．**表 4** と合わせて考えると最適化していないサイズの大きな Cooley-Tukey アルゴリズムの核コードの性能に引きずられていると推測できる．それでも細かく見ると未最適化核コードを使っていてさえも生の FFT より一応は速度はおおむね上回るが，FFTW の核コードの速度と比べるとあまり芳しくない結果といえる．以上から $N = 32$〜64 程度では複雑なスケジューリングより単純にループを展開し，コンパイル時に定数を計算する方が効果的であることが分かり，それ以上では本ライブラリでは実現できていない計算順序のスケジューリングなどの効果が高いことが分かる．また計画を実行時に立てる初期化オーバヘッドは案外大きいことが分かった．FFTW では計画を計算する際の情報を保管することで FFT が大量に反復計算される際にはそのオーバヘッドの重みが軽くなるように工夫がなされている．本ライブラリでは配列が変わっても N が一定なら同じ計

☆ 実際テンプレート・メタ・プログラミングによる最適化抜きの Dynamic 版は $N = 2,048$ までは N の増加にともなって速度が向上している．

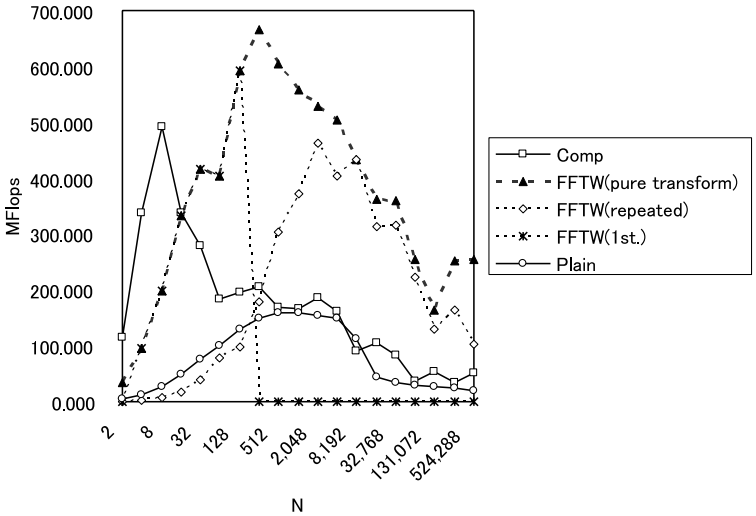

図 15 計時結果
Fig. 15 Timing result.

表 4 各 N に対する分割計画
Table 4 Division plan for each N.

N	Plan< N >::division
2	Solve<2>
4	Solve<4>
8	Solve<8>
16	Solve<16>
32	Solve<32>
64	Division<64, Solve<8>>
128	Division<128, Solve<8>>
256	Division<256, Solve<8>>
512	Division<512, Division<64 , Solve<8>>>
1,024	Division<1024, Division<64 , Solve<8>>>
2,048	Division<2048, Division<64 , Solve<8>>>
4,096	Division<4096, Division<512 , Division<64, Solve<8>>>>
8,192	Division<8192, Division<512 , Division<64, Solve<8>>>>
16,384	Division<16384, Division<512 , Division<64, Solve<8>>>>
32,768	Division<32768, Division<4096 , Division<512, Division<64 , Solve<8>>>>>
65,536	Division<65536, Solve<4>>
131,072	Division<131072, Solve<8>>
262,144	Division<262144, Solve<16>>
524,288	Division<524288, Solve<32>>

画を採用でき，実行時にはいっさいオーバヘッドはなく，実行時に計算に必要な領域以外のメモリ管理も必要ないが，実行時に動的に配列のサイズを決められるという柔軟さは失っている．

5.1 コンパイラの挙動について

現状では原因は不明であるが，ループの再帰的な展開を記述することが一番コンパイラにかける負担が大きい様子である．当初（g++ version 3.2.3），正弦関数の計算でテーラー展開で収束判定を行って計算を打ち切るアルゴリズムを利用していたころは関数値を 10 個も計算するとコンパイルに 5 分かかっていたのに対し，項数の定まった最良近似式を再帰せずに展開した式で計算するようになってからはそのようなことはなくなった．その一方で 4.3 節のように浮動小数点数値とテンプレートのインスタンスを対応させるようなプログラムは案外負担がかからないようである．なお g++の Version を 3.2.3 から 3.4.1 に上げたところ，コンパイルできる FullStatic 版核コードのサイズが N=16 から N=128 まで増加した．ただし N=128 はコンパイルに 30 分もかかる割に性能が芳しくなかったので利用はしていない．

6. 関 連 研 究

FFTW [6] は専用プログラムによってではあるが生成的技法で FFT の適応的最適化を行っているもので．本発表で述べた内容の最適化に関する部分はこの手法を下敷きにしている．このほか同種の最適化方針で信号処理分野における FFT と似た変換の実装に取り組むライブラリとしては Spiral [11] がある．

テンプレート・メタ・プログラミング [4] を利用した最適化の例としては他に MTL [13]，GMCL [4]，Blitz++ [15] がある．前 2 者は記憶階層を意識して行列のブロック化をプログラミングしているが，パラメー

タの設定はユーザが行う．Blitz++はエクスプレッション・テンプレートという技法でユーザプログラムに記述された行列の加減算とスカラー乗算などを検出して複数の演算でループの融合を行うなど，より高レベルの最適化を施している．

1 章で述べたようなコンパイラの肥大化を避ける目的で，**拡張可能コンパイラ**や**アクティブ・ライブラリ**（Active Library）[17] といった概念が提唱されてきている．前者が分野別に限らずコンパイラの機能にアクセスする一般的なプリミティブなどを提供することを主眼にしているのに対して，後者は分野ごとの知識として最適化，デバッグ支援，オブジェクトの表示などをまとめて表現することに主眼を置いている．本発表は最適化に主眼を置いたアクティブ・ライブラリの一例である．

アクティブ・ライブラリと関連の深い技術としては**生成的プログラミング**（Generative Programming）[4] がある．生成的プログラミングでは特定の問題領域のソフトウェアシステムを記述する**ドメイン言語**（Domain Language）で記述された仕様に従ってプログラムを生成するプログラム生成器（generator）を多数の小さなプログラム生成器の組合せとして編成するプログラム技法である．最終的な生成器の部品となるプログラム生成器は理論的にはドメイン言語の要素と出力されるプログラム部品の写像であり，ドメインの部分問題に関する知識をカプセル化したものとなる．そしてこのようなプログラム生成器はさらに部品となる別のプログラム生成器を組み合わせて作られるというように最終的に実行可能な表現にたどり着くまで再帰的にこの関係が繰り返される．このようなプログラム生成器はプログラミング環境やツール，言語処理系などによってサポートされる．C++のテンプレートはドメイン言語というほど洗練されてはいないが，一応コード生成器と見なすことができる．元々ジェネリック・プログラミングで実行効率を求めると静的インスタンス化（コンパイル時の展開）という解に落ち着くことが多く，コード生成という点で生成的プログラミングとは密接な関係がある．

コード生成に関しては Tick-C [9] などのテンプレート型コード生成といわれる手法がある．これは同じ 2 レベル計算でもテンプレート・メタ・プログラミングのようにコンパイル時へ計算を持ち込む手法とは逆に実行時へとコード生成機能を遅延する（コンパイル時には型検査などは済ませてテンプレートという実行時にコード生成を行うための穴あきバイナリを用意しておいて実行時に穴埋めを行うことでコード生成を行う）ものである．このためテンプレート・メタ・プログラミングのようにコード生成段階が遅いという問題は軽減され，実行時に定まるパラメータを取り込める柔軟性が獲得できる．しかし生成されたコードをコンパイラが通常どおり全力で最適化するテンプレート・メタ・プログラミングとは異なり，生成されたコードに対して十分な最適化が施せないため生成されたコードの実効効率は落ちる傾向がある．

C++は通常の C++プログラムでの実行時の計算を記述するためのベース・レベルとテンプレートのインスタンス化によるコンパイル時の計算を記述するメタ・レベルの 2 レベルある 2 レベル言語であるといわれることがある．これを一般化したマルチ・レベル・プログラミング言語という概念が存在する．このような言語が実際にあればメタレベルの実行自体を最適化するため利用できるかもしれない．実際テンプレート・メタ・プログラム自体の実行を高速化が可能ならば利用したくなる局面は本発表で紹介したライブラリの開発過程で何度かあった．たとえば，浮動小数点ライブラリが利用する整数演算ライブラリは多重精度整数演算のアルゴリズムを実装したものであるが，コンパイル時計算の効率を高めるため手でループを展開して実装している．これはマルチ・レベル記述が可能であれば自動で展開できたであろう．MetaML [8] は ML 言語から派生した言語であり ML 言語の式をマルチ・レベルで実行するようマークアップできる．ただ残念ながら MetaML はマルチ・レベル言語の型推論アルゴリズムの実験のために開発されたものであり，実用的な速度で動く実行コードを生成することはできない．

特化およびコード生成は部分評価の技術とも深い関係がある（C++のテンプレートを部分評価と見る見方も存在する[16]）．C 言語を対象とした部分評価に基づく特化器も Tempo [2] や C-Mix [1] などいくつか存在する．実際，今回核コード生成などをテンプレート・メタ・プログラミングで記述したことは理論上では人力で束縛時解析（部分評価で特化の前に行う解析で，プログラム中の静的に計算できる部分を求める）を行ったことに等しい．これは結構煩雑な作業であるので，もしこれを自動化できればプログラムの開発が大変楽になると期待できる．しかし単に FFT のプログラムを特化器にかけて完全自動化では必ずしもうまくいかない．任意の N について無制限に特化を行ってループを展開しても効率は向上しないし，今回のような FFT の合成アルゴリズムが自動で生成されることもない．おそらく両者を組合せ基本構造をメタ・プログラミングで記述し，細部を自動で特化するような枠組みが将

来的には必要となると考えられる．

7. ま と め

本発表では FFT[3] を例にとり，両者を組み合わせた最適化をテンプレート・メタ・プログラミング，特に生成的プログラミング・スタイルで実装し，アクティブ・ライブラリ化することを目指した実装を行った．そしてその実装の効果を検証した．本発表で採用した最適化の基本的な考え方そのものは **FFTW**[6] として紹介され，すでに実績をあげているものであるが，本発表はこの基本的な考えがテンプレート・メタ・プログラミング，特に生成的プログラミング・スタイルで記述されるアクティブ・ライブラリとしてどこまで実現でき，実効性があるかを検証した．

その結果，ループの展開や関数値の静的計算による核コードの最適化は高速な核コードの生成に効果があることが分かる一方で，その手法により生成できる核コードが有効であるサイズの上限が分かった．また合成のために計測データから分割計画を動的計画法でコンパイル時に求めるメカニズムも機能することも分かった．

テンプレート・メタ・プログラミング技法はデータ型の不備や可読性，デバッグやチューニングの困難さなど欠点は多々あるものの実際に意図どおり動作すること，そしてこの技法をを用いて行う FFT の適応的最適化は現状では生成できる最適化核コードのサイズや合成部の性能についてまだ改良の余地ががあるものの十分可能性のある選択肢であると考えられる．

今後は合成部の性能向上，より大きなサイズの核コード・サイズについてスケジューリングなどの最適化実現，マルチ・スレッドや SIMD 命令のサポートなど核コードの性能向上に努めていきたい．また同時に Experession Template 技法などを導入してコードの可読性，保守性の向上にも努めたい．

謝辞 静的な浮動小数点数の実装で三角関数を実装するにあたって NetNUMPAC（http://netnumpac.fuis.fukui-u.ac.jp/）のソース公開サービスを利用し，ソースを参考にさせていただきました．ありがとうございます．また，無所属の不安定な生活を心配して有形無形の支援をしてくれている父母に感謝します．

参 考 文 献

1) Andersen, L.O.: Program Analysis and Specialization for the C Programming Language, *University of Copenhagen* (1994).
2) Consel, C., et al.: Tempo: Specializing Systems Applications and Beyond C, *ACM Computing Surveys* (1996).
3) Cooley, J.W. and Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series, *Mathematics of Computation*, Vol.19 (1965).
4) Czarnecki, K. and Eisenecker, U.W.: *Generative Programming*, Addison-Wesley (2000).
5) Frigo, M.: A fast Fourier transform compiler, *Proc. 1999 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Vol.34, No.5, pp.169–180, ACM (1999).
6) Frigo, M. and Johnson, S.G.: FFTW: An adaptive software architecture for the FFT, *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, Vol.3, pp.1381–1384, IEEE (1998).
7) Frigo, M., Leiserson, C.E., Prokop, H. and Ramachandran, S.: Cache-oblivious algorithms, *Proc.40th Ann.Symp.on Foundations of Comp. Sci.* (*FOCS*), pp.285–297 (1999).
8) Martel, M. and Sheard, T.: Introduction to Multi-Stage Programming Using MetaML, Technical report, OGI, Portland, OR (1997).
9) Poletto, M., Hsieh, W.C., Engler, D.R. and Kaashoek, M.F.: 'C and tcc: A language and compiler for dynamic code generation, *ACM Trans. Prog. Lang. Syst.*, Vol.21, No.2, pp.324–369 (1999).
10) Press, W.H., Teukolsky, S.A., Vetterling, W.T. and Flannery, B.P.: *NUMERICAL RECIPES in C*, Cambridge University Press (1988).
11) Püschel, M., Singer, B., Xiong, J., Moura, J., Johnson, J., Padua, D., Veloso, M. and Johnson, R.W.: SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms, Vol.18, No.1, pp.21–45, IEEE (2004).
12) Rader, C.M.: Discrete Fourier transforms when the number of data samples is prime, *Proc. IEEE*, Vol.56, pp.1107–1108, IEEE (1968).
13) Siek, J.G. and Lumsdaine, A.: The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra, *ISCOPE* (1998).
14) Unruh, E.: Prime number computation, Technical report (1994). X3J16-94-0075/SO WG21-462.
15) Veldhuizen, T.L.: Linear algebra with C++ template metaprograms, Dr. Dobb's Journal (1996).
16) Veldhuizen, T.L.: C++ Templates as Partial Evaluation, *ACM SIGPLAN Workshop on*

Partial Evaluation and Semantics-Based Program Manipulation (1999).
17) Veldhuizen, T.L. and Gannon, D.: Active Libraries: Rethinking the roles of compilers and libraries, *SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing* (1998).

```
template<typename Env>
struct FSCalcLoop3Cond
//all variannts of lower bits of indices
{
  static const bool cond
    = (Env::i < Env::count);
};
```

図 16 FFT の最内ループの条件部
Fig. 16 FFT most inner loop condition.

```
template<typename Enviroment>
struct FSCalcLoop3Body
{
  struct Env
  {
    typedef typename Enviroment::Conf Conf;
    typedef typename Conf::Complex Complex;
    typedef typename Conf::Int Int;
    typedef typename Conf::Float Float;
    static const Int stride
      = Enviroment::stride;
    static const Int step
      = Enviroment::step;
    static const Int curOfst0
      = Enviroment::offset0;
    static const Int curOfst
      = Enviroment::offset1;
    static const Int offset0
      = Enviroment::offset0 + step;
    static const Int offset1
      = offset0 + stride;
    static const Int count
      = Enviroment::count;
    static const Int i = Enviroment::i + 1;
```

図 17 FFT の最内ループの本体（1）
Fig. 17 FFT most inner loop body (1).

付　　録

A.1 FFT のソース・コード

以下のコードは図 8 の For テンプレートを利用して Cooley-Tukey アルゴリズムの中心となる 3 重ループを記述したものである．ループは内側にあたるものを外側が参照することになるので，内側のループから順に宣言される．

図 16 は最内ループの条件判定に利用される．For テンプレートを介して渡された環境 Env を参照して bool 型定数 cond に値を設定する．値が偽であれば再帰は打ち切られる．

図 17，**図 18** の最内ループ本体ではデータへの最終的なオフセットを現すクラス定数，と実際に配列から値を取り出して計算を行うコードとして static メンバ関数 exec() が置かれている．「ループ」で直前の「周回」のデータの集まりを表す環境引数 Env がテンプレートのパラメータとして For テンプレート経由で渡されるので，それを参照し static メンバ関数で参照したり次の「周回」のための環境を設定に利用することができる．

```
    static void exec(Float wr, Float wi)
    {
      Enviroment::exec(wr, wi);
      Complex x0 = Conf::buffer[curOfst0];
      Complex x1 = Conf::buffer[curOfst1];
      x1 = Complex(
          x1.real()*wr - x1.imag()*wi
          , x1.imag()*wr + x1.real()*wi);
      Conf::buffer[curOfst0] = x0 + x1;
      Conf::buffer[curOfst1] = x0 - x1;
    }
  };
};
```

図 18 FFT の最内ループの本体（2）
Fig. 18 FFT most inner loop body (2).

```
template<typename Env>
struct FSCalcLoop2Cond
//all variannts of upper bits of indices
{
  static const bool cond
    = (Env::m < Env::mmax);
};
```

図 19 FFT の 2 番目のループの条件部
Fig. 19 FFT 2nd. inner loop condition part.

```
template<typename Enviroment>
struct FSCalcLoop2Body
{
  struct Env
  {
    typedef typename Enviroment::Conf Conf;
    typedef typename Conf::Int Int;
    typedef typename Conf::Float Float;
    static const Int mmax
      = Enviroment::mmax;
    static const Int count
      = Enviroment::count;
    static const Int stride
        = Enviroment::stride;
    static const Int step
        = Enviroment::step;
```

図 20 FFT の 2 番目のループの本体 (1)
Fig. 20 FFT 2nd. inner loop body (1).

```
    struct InitEnv
    {
      typedef typename
        Enviroment::Conf Conf;
      static const Int stride
        = Enviroment::stride;
      static const Int step =
        Enviroment::step;
      static const Int count
        = Enviroment::count;
      static const Int offset0
        = Enviroment::m;
      static const Int offset1
        = offset0 + stride;
      static const Int i = 0;
      static void exec(Float wr, Float wi)
      {;}
    };
```

図 21 FFT の 2 番目のループの本体 (2)
Fig. 21 FFT 2nd. inner loop body (2).

図 19 は 2 番目のループの条件判定部である．

図 20，**図 21**，**図 22** のループ本体では 2 番目のループのために初期環境を設定し For を介して最内ループを呼び出している．回転因子の値はここで定まるので回転因子を計算するテンプレートを呼び出している．後で計算を実行するために最内ループの static メソッド InnerLoopResult::exec() に回転因子を渡して呼び出すために実行時用の double 型の値を取り出す処理を行う static メンバ関数 exec() が置かれている．環境の受け渡しなどは最内ループと同様である．

```
    typedef typename For<InitEnv
      , FSCalcLoop3Cond
      , FSCalcLoop3Body>::Result
      InnerLoopResult;
    static const Int m = Enviroment::m + 1;
    typedef TwiddleFactor<Float
      , Conf::direction, Enviroment::mmax
      , Enviroment::m>  twf;
    static void exec(void)
    {
      Float vwr = twf::getRe();
      Float vwi = twf::getIm();
      Enviroment::exec();
      InnerLoopResult::exec(vwr, vwi);
    }
  };
};
```

図 22 FFT の 2 番目のループの本体 (3)
Fig. 22 FFT 2nd. inner loop body (3).

```
template<typename Env>
struct FSCalcLoop1Cond
{
  static const bool cond
    = (Env::p > 0);
};
```

図 23 FFT の最外ループの条件部
Fig. 23 FFT most outer loop condition part.

図 23 は最外ループの条件判定部である．

図 24，**図 25** のループ本体では最外のループのために初期環境を設定し For を介して 2 番目のループを呼び出している．後で計算を実行するために 2 番目のループの static メソッド InnerLoopResult::exec() を呼び出すための static メンバ関数 exec() が置かれている．環境の受け渡しなどは最内ループと同様である．

図 26，**図 27** は FFT の本体では最外ループのために初期環境を設定し For を介して最内ループを呼び出している．後で計算を実行するために最外ループの static メソッド InnerLoopResult::exec() を呼び出すための static メンバ関数 exec() が置かれている．

(平成 16 年 7 月 2 日受付)
(平成 16 年 10 月 12 日採録)

```
template<typename Enviroment>
struct FSCalcLoop1Body
{
  struct Env
  {
    typedef typename Enviroment::Conf Conf;
    typedef typename Conf::Int Int;
    typedef typename Conf::Float Float;
    static const Int mmax
      = Enviroment::mmax * 2;
    static const Int count
      = Enviroment::count / 2;
    static const Int p = Enviroment::p - 1;
```

図 24 FFT の最外ループの本体 (1)
Fig. 24 FFT most outer loop body (1).

```
    struct InitEnv
    {
      typedef typename
        Enviroment::Conf Conf;
      static const Int stride
        = Enviroment::mmax;
      static const Int step = stride * 2;
      static const Int mmax
        = Enviroment::mmax;
      static const Int count
      = Enviroment::count;
      static const Int m = 0;
      static void exec(void)
      {;}
    };
    typedef typename For<InitEnv
      , FSCalcLoop2Cond
      , FSCalcLoop2Body
      >::Result InnerLoopResult;
    static void exec(void)
    {
      Enviroment::exec();
      InnerLoopResult::exec();
    }
  };
};
```

図 25 FFT の最外ループの本体 (2)
Fig. 25 FFT most outer loop body (2).

```
template<typename Config>
struct FullStaticFFTCalc
{
  typedef Config Conf;
  typedef typename Conf::Int Int;
  typedef typename Conf::UInt UInt;
  static const Int num = Conf::size;
  static const Int nbits
    = Int(BitsForStaticN<UInt
        , UInt(num - 1)>::result);
```

図 26 FFT の本体 (1)
Fig. 26 FFT body (1).

```
  struct InitEnv
  {
    typedef Config Conf;
    static const Int p = nbits;
    static const Int mmax = 1;// ==2^0
    static const Int count
      = num / 2;// ==2^(bits-1)
    static void exec()
    {;}
  };
  typedef typename For<InitEnv
    , FSCalcLoop1Cond
    , FSCalcLoop1Body>::Result Code;
  static void exec(void)
  {
    Code::exec();
  }
};
```

図 27 FFT の本体 (2)
Fig. 27 FFT body (2).

神戸　隆行（正会員）

1967 年生まれ．1991 年名古屋大学工学部情報工学科卒業．1993 年同大学院工学研究科情報工学専攻修了．1993〜1998 年株式会社富士通（富士通研究所）勤務．数式処理システムの開発に携わる．2000〜2003 年博士後期課程在籍の後中退．現在は無所属，求職中．関心のあるテーマは生成的プログラミング，プログラム特化，数学ソフトウェア向けプログラミング言語．