高速フーリエ変換におけるキャッシュ最適化
Cache Optimization on Fast Fourier Transform

筑波大学システム情報系（計算科学研究センター）
高橋　大介

高速フーリエ変換（fast Fourier transform、FFT）は、科学技術計算において今日広く用いられているアルゴリズムである。本稿ではFFTにおけるキャッシュ最適化について述べる。まず、離散フーリエ変換（discrete Fourier transform、DFT）の定義を示した後にFFTの基本的な考え方について説明する。さらに、six-step FFTアルゴリズムについて説明した後、キャッシュブロッキングがsix-step FFTに対して適用できることを示す。

1．はじめに

高速フーリエ変換（fast Fourier transform、FFT）は、離散フーリエ変換（discrete Fourier transform、DFT）を高速に計算するアルゴリズムとして知られている。

FFTは1965年のCooleyとTukey［1］の論文によって一般に広く知られるようになったが、1805年頃にGaussが同様のアルゴリズムを独自に発見していた［2］。

多くのFFTアルゴリズムは処理するデータがキャッシュメモリに載っている場合には高い性能を示す。しかし、問題サイズがキャッシュメモリのサイズより大きくなった場合においては著しい性能の低下をきたす。FFTアルゴリズムにおける1つの目標は、いかにしてキャッシュミスの回数を減らすかということにある。

近年のプロセッサの演算速度に対する主記憶のアクセス速度は相対的に遅くなってきており、主記憶のアクセス回数を減らすことは、より重要になっている。したがって、キャッシュメモリを搭載したプロセッサにおけるFFTアルゴリズムでは、演算回数だけではなく、主記憶のアクセス回数も減らすことが重要になる。ここで、キャッシュミスの回数を減らすことができれば、主記憶のアクセス回数を減らすうえで非常に効果があるといえる。

本稿では、FFTにおけるキャッシュ最適化について述べる。

2．離散フーリエ変換

n点のデータに対するDFTは次式で定義される。

$$y(k) = \sum_{j=0}^{n-1} x(j)\,\omega_n^{jk}, \quad 0 \le k \le n-1 \qquad (1)$$

また、n点のデータに対する離散フーリエ変換（discrete inverse Fourier transform、逆DFT）は次式で定義される。

$$x(j) = \frac{1}{n}\sum_{k=0}^{n-1} y(k)\omega_n^{-jk}, \quad 0 \le j \le n-1 \qquad (2)$$

ここで、$x(j)$および$y(k)$は複素数の値であり、$\omega_n = e^{-2\pi i/n}$、$i = \sqrt{-1}$である。

式（1）において、例えばn=4のとき、DFT

は以下のように計算できる。

$$\begin{aligned} y(0) &= x(0)\omega^0 + x(1)\omega^0 + x(2)\omega^0 + x(3)\omega^0 \\ y(1) &= x(0)\omega^0 + x(1)\omega^1 + x(2)\omega^2 + x(3)\omega^3 \\ y(2) &= x(0)\omega^0 + x(1)\omega^2 + x(2)\omega^4 + x(3)\omega^6 \\ y(3) &= x(0)\omega^0 + x(1)\omega^3 + x(2)\omega^6 + x(3)\omega^9 \end{aligned} \tag{3}$$

式（3）は行列を用いれば、より簡単に

$$\begin{bmatrix} y(0) \\ y(1) \\ y(2) \\ y(3) \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{bmatrix} \tag{4}$$

と行列ベクトル積の形で表すことができる。

式（1）において、$x(j)$、$y(k)$およびωは複素数の値であることから、この式の行列ベクトル積を計算するためには、n^2回の複素数乗算と$n(n\text{-}1)$回の複素数加算が必要となることが分かる。

3．高速フーリエ変換

3.1　基本的な考え方

式（1）においてnが2で割り切れる場合には、n点のデータを$n/2$点の前半部分と$n/2$点の後半部分に分解することにより、n点DFTは以下のように表すことができる。

$$\begin{aligned} y(k) &= \sum_{j=0}^{n/2-1} \left\{ x(j)\omega_n^{jk} + x(j+n/2)\omega_n^{(j+n/2)k} \right\} \\ &= \sum_{j=0}^{n/2-1} \left\{ x(j) + x(j+n/2)\omega_n^{(n/2)k} \right\} \omega_n^{jk}, \\ & 0 \le k \le n-1 \end{aligned} \tag{5}$$

式（5）において、$x(j+n/2)$に掛かっている$\omega_n^{(n/2)k}$は、$\omega_n = e^{-2\pi i/n}$より、

$$\omega_n^{(n/2)k} = e^{-\pi i k} = \begin{cases} 1 & k\text{が偶数} \\ -1 & k\text{が奇数} \end{cases} \tag{6}$$

となる。

したがって、kが偶数の場合と奇数の場合に分けると、n点DFTは以下のように2つの$n/2$点DFTに分解できる。

$$\begin{aligned} y(2k) &= \sum_{j=0}^{n/2-1} \{x(j) + x(j+n/2)\} \omega_n^{2jk} \\ &= \sum_{j=0}^{n/2-1} \{x(j) + x(j+n/2)\} \omega_{n/2}^{jk}, \\ & 0 \le k \le n/2-1 \end{aligned} \tag{7}$$

$$\begin{aligned} y(2k+1) &= \sum_{j=0}^{n/2-1} \{x(j) - x(j+n/2)\} \omega_n^{j(2k+1)} \\ &= \sum_{j=0}^{n/2-1} \left\{ \{x(j) - x(j+n/2)\}\omega_n^j \right\} \omega_{n/2}^{jk}, \\ & 0 \le k \le n/2-1 \end{aligned} \tag{8}$$

式（7）、（8）における$n/2$点DFTは、直接計算すると$n^2/4$回の複素数乗算と（$n/2$）（$n/2$-1）回の複素数加算で計算できることから、この分解により計算量が約1/2になることが分かる。さらに、nが2のべきである場合には、この分解を再帰的に行うことで、最終的には2点DFTに帰着させることができ、その結果演算量を$O(n\log n)$に削減することが可能になる。

FFTの分解には、式（5）のようにn点のデータを$n/2$点の前半部分と$n/2$点の後半部分に分解する**周波数間引き**（decimation in frequency）と、n点のデータを偶数番目と奇数番目に分解する**時間間引き**（decimation in time）の方法があるが、どちらの方法でも演算量は同じになる。

式（7）、（8）の考え方を素直にコーディングすると、図1のような再帰呼び出しによる周波数間引きFFTルーチンが書ける。

```
    recursive subroutine
   &           fft(x,temp,n)
    implicit real*8 (a-h,o-z)
    complex*16 x(*),temp(*)
!
    if (n .le. 1) return
!
    pi=4.0d0*atan(1.0d0)
    px=-2.0d0*pi/dble(n)
!
    do j=1,n/2
      w=px*dble(j-1)
      temp(j)=x(j)+x(j+n/2)
      temp(j+n/2)=(x(j)-x(j+n/2))
   &  *dcmplx(cos(w),sin(w))
!
    call fft(temp,x,n/2)
    call fft(temp(n/2+1),x,n/2)
!
    do j=1,n/2
      x(2*j-1)=temp(j)
      x(2*j)=temp(j+n/2)
    end do
    return
    end
```

図1　再帰呼び出しによる周波数間引きFFTルーチン

このFFTルーチンでは、nが2のべきである場合$\log_2 n$回の再帰呼び出しが行われるが、1回の呼び出し毎に$n/2$回の複素数乗算とn回の複素数加減算が行われるので、結局$O(n \log n)$回の演算量になることが分かる。また、このFFTルーチンでは最内側ループにおいて毎回sin、cos関数を呼び出して三角関数の値を計算しているが、高速化のためにはあらかじめこれらの値を計算しておきテーブルにするなどの工夫を行うことが多い。

なお、逆FFTを求めるにはw=px*dble(j-1)と計算している部分の符号を反転させw=-px*dble(j-1)として計算を行い、計算結果を$1/n$倍すればよい。

3.2　Cooley-Tukeyアルゴリズムと Stockhamアルゴリズム

```
    subroutine fft(x,n,m)
! データ数 n=2**m
    implicit real*8 (a-h,o-z)
    complex*16 x(*),temp
!
    pi=4.0d0*atan(1.0d0)
!
    l=n
    do k=1,m
      px=-2.0d0*pi/dble(l)
      l=l/2
      do j=1,l
        l=l/2
        w=px*dble(j-1)
        do i=j,n,l*2
          temp=x(i)-x(i+l)
          x(i)=x(i)+x(i+l)
          x(i+l)=temp
   &       *dcmplx(cos(w),sin(w))
        end do
      end do
    end do
! 出力データをビット反転順に並べ替え
    j=1
    do i=1,n-1
      if (i .lt. j) then
        temp=x(i)
        x(i)=x(j)
        x(j)=temp
      end if
      k=n/2
 10   if (k .lt. j) then
        j=j-k
        k=k/2
        go to 10
      end if
      j=j+k
    end do
    return
    end
```

図2　周波数間引きCooley-Tukey FFTルーチン

3.1節で示したFFTルーチンにおける再帰呼び出しをループに書き換えることにより、Cooley-Tukeyアルゴリズム［1］としてよく知られているFFTアルゴリズムが導かれる。図2に周波数間引きCooley-Tukey FFTルーチンを示す。Cooley-Tukeyアルゴリズムでは、入力データは出力データに上書きされる

（in-place）という特徴がある。

その一方でCooley-Tukeyアルゴリズムでは、出力データの順序が**ビット反転**（bit reversal）して得られた順序になる。出力データの順序を入力データと同じにするためには、ビット反転して得られた順序を元通りに並べ替えることが必要になるが、この並び替えはメモリアクセスの局所性が低いことからキャッシュミスが多発する。さらに、Cooley-Tukeyアルゴリズムの最内側ループでは2のべき乗飛びのストライドアクセスになっていることから、キャッシュラインコンフリクトも多発する。したがって、階層型メモリを意識した場合、Cooley-Tukeyアルゴリズムは必ずしも好ましいアルゴリズムではない。

Cooley-Tukeyのアルゴリズムは、入力データは出力データに上書きされる、いわゆるin-placeアルゴリズムであったが、入力データと出力データを別の配列にするout-of-placeアルゴリズムを構成することもできる。out-of-placeアルゴリズムとしてよく知られているのは、Stockhamアルゴリズム［3,7］である。

Stockhamアルゴリズムでは、入力データが出力データに上書きできないため、Cookey-Tukeyアルゴリズムに比べて必要となるメモリ容量が2倍になる。しかし、最内側ループが連続アクセスになっており、またビット反転順に並べ替える処理が不要になっている。したがって、階層型メモリへの適合性という観点からは、StockhamアルゴリズムがCooley-Tukeyアルゴリズムに比べて有利であるといえる。

4．Six-Step FFTアルゴリズム

キャッシュメモリを有効に活用することのできる、six-step FFTアルゴリズム［4, 5］について説明する。six-step FFTアルゴリズムでは、一次元FFTを二次元表現で表して計算することにより、キャッシュミスを少なくできるのが特徴である。

n点DFTを計算する際に$n=n_1\times n_2$と分解できるものとすると、式（1）におけるjおよびkは、

$$j = j_1 + j_2, \quad k = k_2 + k_1 n_2 \tag{9}$$

と書くことができる。そのとき、式（1）のxとyは次のような二次元配列（columnwise）で表すことができる。

$$\begin{aligned} x_j &= x(j_1, j_2), \quad 0 \le j_1 \le n_1 - 1, \\ & \qquad\qquad 0 \le j_2 \le n_2 - 1 \\ y_k &= y(k_2, k_1), \quad 0 \le k_1 \le n_1 - 1, \\ & \qquad\qquad 0 \le k_2 \le n_2 - 1 \end{aligned} \tag{10}$$

したがって式（1）は式（11）のように変形できる。

$$\begin{aligned} & y(k_2, k_1) \\ &= \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} x(j_1, j_2) \omega_{n_2}^{j_2 k_2} \omega_{n_1 n_2}^{j_1 k_2} \omega_{n_1}^{j_1 k_1} \end{aligned} \tag{11}$$

式（11）から次に示されるような、six-step FFTアルゴリズム［4, 5］が導かれる。

Step 1 ： 転置

$$x_1(j_2, j_1) = x(j_1, j_2)$$

Step 2 ： n_1組のn_2点multicolumn FFT

$$x_2(k_2, j_1) = \sum_{j_2=0}^{n_2-1} x_1(j_2, j_1) \omega_{n_2}^{j_2 k_2}$$

Step 3 ： ひねり係数の乗算

$$x_3(k_2, j_1) = x_2(k_2, j_1) \omega_{n_1 n_2}^{j_1 k_2}$$

Step 4 ： 転置

$$x_4(j_1, k_2) = x_3(k_2, j_1)$$

Step 5：n_2組のn_1点multicolumn FFT

$$x_5(k_1,k_2)=\sum_{j_1=0}^{n_1-1}x_4(j_1,k_2)\omega_{n_1}^{j_1k_1}$$

Step 6：転置

$$y(k_2,k_1)=x_5(k_1,k_2)$$

Step 3における$\omega_{n_1}^{j_1k_1}$は、ひねり係数[6]と呼ばれる1の原始根であり、複素数である。図3にsix-step FFTアルゴリズムを示す。Step 1および4において行列の転置を行っているが、これはStep 2および5におけるmulticolumn FFTのメモリアクセスを連続にするためである。さらに、Step 6の行列の転置は入力データと出力データの順序を同じにするために必要になる。

six-step FFTにおいて、Step 1、4、6の行列の転置および、Step 3のひねり係数の乗算をブロック化したsix-step FFTアルゴリズムが文献［5］に示されている。しかし、このFFTアルゴリズムではmulticolumn FFTの部分と行列の転置の部分が分離されているため、multicolumn FFTにおいてキャッシュメモリに載っていたデータが行列の転置の際に有効に再利用されないという問題点がある。ここで、3回の行列の転置がキャッシュメモリを搭載した計算機においてボトルネックとなる。

5．ブロックSix-Step FFTアルゴリズム

six-step FFTにおいて、さらにキャッシュ内のデータを有効に再利用し、キャッシュミスの回数を少なくするため、six-step FFTでは分離されていたmulticolumn FFTと行列の転置を統合した、ブロックsix-step FFTアルゴリズムを構成することができる。4章で説明したsix-step FFTにおいて、$n=n_1\times n_2$とし、n_bをブロックサイズとする。ここで、プロセッサはmulti-levelキャッシュメモリを搭載しているものと仮定する。ブロックsix-step FFTアルゴリズムは以下のようになる。

```
    complex*16 x(n1,n2),y(n2,n1)
    complex*16 w(n2,n1)
!  Step 1: 転置
    do i=1,n1
      do j=1,n2
        y(j,i)=x(i,j)
      end do
    end do
! Step 2: n1 組の n2 点 multicolumn FFT
    do i=1,n1
      call fft(y(1,i),n2)
    end do
!  Step 3: ひねり係数の乗算
    do i=1,n1
      do j=1,n2
        y(j,i)=y(j,i)*w(j,i)
      end do
    end do
!  Step 4: 転置
    do j=1,n2
      do i=1,n1
        x(i,j)=y(j,i)
      end do
    end do
! Step 5: n2 組の n1 点 multicolumn FFT
    do j=1,n2
      call fft(x(1,j),n1)
    end do
!  Step 6: 転置
    do i=1,n1
      do j=1,n2
        y(j,i)=x(i,j)
      end do
    end do
```

図3　six-step FFTアルゴリズム

Step 1：$n=n_1\times n_2$の大きさの複素数配列xに入力データが入っているとする。このとき、$n_1\times n_2$の大きさの配列xからn_b列ずつデータを転置しながら、$n_2\times n_b$の大きさの作業用配列workに転送する。ここでブロックサイズn_bは配列workがL2キャッシュに載るように定める。

Step 2：n_b組のn_2点multicolumn FFTをL2キャッシュに載っている$n_2\times n_b$配列workの上で行う。ここで各column FFTは、ほぼL1キャッシュ内で行えるものとする。

Step 3：multicolumn FFTを行った後L2

```
    complex*16 x(n1,n2),y(n2,n1)
    complex*16 w(n1,n2)
    complex*16 work(n2+np,nb)
!
    do ii=1,n1,nb
! Step 1: ブロック化転置
      do jj=1,n2,nb
        do i=ii,min(ii+nb-1,n1)
          do j=jj,min(jj+nb-1,n2)
            work(j,i-ii+1)=x(i,j)
          end do
        end do
      end do
! Step 2: n1 組の n2 点 multicolumn FFT
      do i=ii,min(ii+nb-1,n1)
        call fft(work(1,i-ii+1),n2)
      end do
! Step 3: ブロック化したひねり係数の
            乗算および転置
       do j=1,n2
         do i=ii,min(ii+nb-1,n1)
           x(i,j)=
     &        work(j,i-ii+1)*w(i,j)
         end do
       end do
     end do
     do jj=1,n2,nb
! Step 4: n2 組の n1 点 multicolumn FFT
       do j=jj,min(jj+nb-1,n2)
         call fft(x(1,j),n1)
       end do
! Step 5: ブロック化転置
       do i=1,n1
         do j=jj,min(jj+nb-1,n2)
           y(j,i)=x(i,j)
         end do
       end do
     end do
```

図4　ブロックsix-step FFTアルゴリズム

キャッシュに残っている$n_2 \times n_b$配列workの各要素にひねり係数wの乗算を行う。そしてこの$n_2 \times n_b$配列workのデータをn_b列ずつ転置しながら元の$n_1 \times n_2$配列xの同じ場所に再び格納する。

Step 4：n_2組のn_1点multicolumn FFTを$n_1 \times n_2$配列xの上で行う。ここでも各column FFTは、ほぼL1キャッシュ内で行える。

Step 5：最後にこの$n_1 \times n_2$配列xをn_b列ずつ転置して、$n_2 \times n_1$配列yに格納する。

ブロックsix-step FFTアルゴリズムのプログラムは図4のようになる。

このプログラムにおいて、nbはブロックサイズ、npはパディングサイズ、workは作業用の配列である。図5にブロックsix-step FFTアルゴリズムのメモリ配置を示す。なお、図5において、配列x、work、y の中の1から16の数字は、配列のアクセス順序を示している。

また、作業用の配列workに**パディング**を施すことにより、配列workから配列xにデータを転送する際や、配列work上でmulticolumn FFTを行う際にキャッシュラインコンフリクトの発生を極力防ぐことができる。このパディングは、配列の先導次元（leading dimension）が2のべき乗となっている場合、これが2のべき乗にならないようにすることで、行列の転置において2のべき乗飛びのストライドアクセスになることを防ぐために用いられる。

ブロックsix-step FFTアルゴリズムは、いわゆるtwo-passアルゴリズム［4］となる。つまり、ブロックsix-step FFTアルゴリズムではn点FFTの演算回数は$O(n \log n)$であるのに対し、主記憶のアクセス回数は理想的には$O(n)$で済む。

なお、本稿ではStep 2およびStep 4の各column FFTはL1キャッシュに載ると想定しているが、問題サイズnが非常に大きい場合には各column FFTがL1キャッシュに載らないことも十分予想される。このような場合は二次元表現ではなく、多次元表現［8, 9］を用いて、各column FFTの問題サイズを小さくすることにより、L1キャッシュ内で各column FFTを計算することができる。ただし、三次元以上の多次元表現を用いた場合にはtwo-passアルゴリズムとすることはできず、例えば三次元表現を用いた場合にはthree-passアルゴリズムになる。このように、多次元表現の次元数を大きくするに従っ

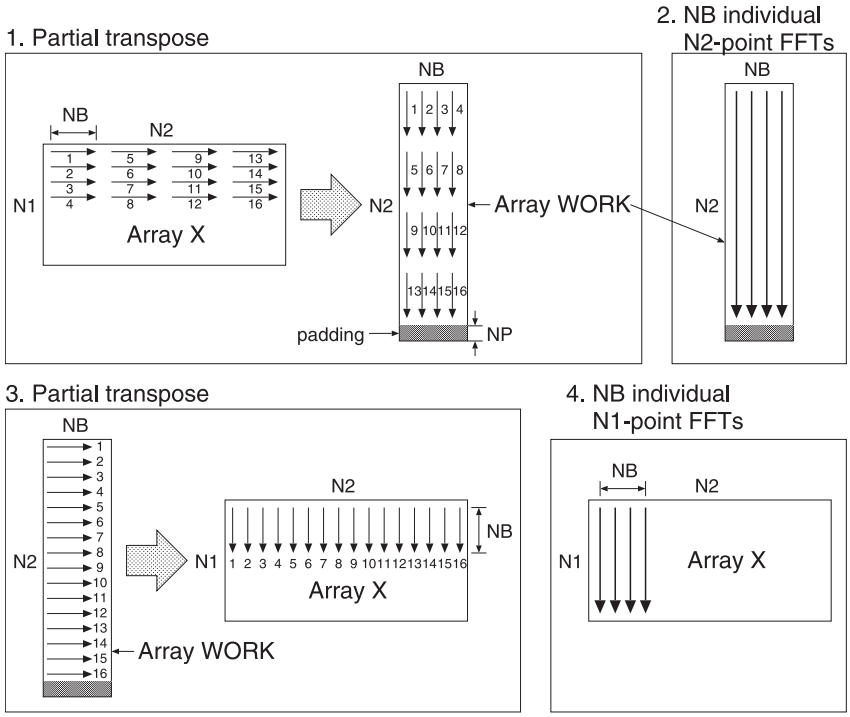

図5　ブロックsix-step FFTアルゴリズムのメモリ配置

て、より大きな問題サイズのFFTに対応することが可能になるが、その一方で主記憶のアクセス回数が増加する。これは、ブロックsix-step FFTにおいても性能はキャッシュメモリの容量に依存することを示している。

なお、out-of-placeアルゴリズム（例えばStockhamアルゴリズム）をStep 2、4のmulticolumn FFTに用いたとしても、余分に必要となる配列の大きさは$O(\sqrt{n})$で済む。

また、一次元FFTの結果が転置された出力で構わなければ、Step 5の行列の転置は省略することができる。この場合、作業用の配列は$O(\sqrt{n})$の大きさの配列workだけで済むことが分かる。

6．まとめ

本稿では、FFTにおけるキャッシュ最適化について述べた。ブロックsix-step FFTでは、キャッシュメモリの再利用率を高くすることにより、キャッシュミスを少なくし、その結果主記憶のアクセス回数も少なくすることができる。データがキャッシュに入り切らないような大きな問題サイズのFFTでは、ブロックsix-step FFTは効果的である。

本稿で述べたブロック化の手法は、他のアプリケーションの高速化にも適用できると考えられる。

参考文献

［1］ J. W. Cooley and J. W. Tukey: An algorithm for the machine calculation of complex Fourier series, Math. Comput. Vol. 19, pp. 297-301 (1965).

［2］ M. T. Heideman, D. H.Johnson, and C. S. Burrus: Gauss and the history of the fast Fourier transform, IEEE ASSP Magazine, Vol. 1, pp. 14-21 (1984).

［3］ P. N. Swarztrauber: FFT algorithms for vector computers, Parallel Comput-

ing, Vol. 1, pp. 45-63 (1984).

[4] D. H. Bailey: FFTs in external or hierarchical memory, The Journal of Supercomputing, Vol. 4, pp. 23-35 (1990).

[5] C. Van Loan: Computational Frameworks for the Fast Fourier Transform, SIAM Press, Philadelphia, PA (1992).

[6] E. O. Brigham: The Fast Fourier Transform and its Applications, Prentice-Hall, Englewood Cliffs, NJ (1988).

[7] W. T. Cochrane, J. W. Cooley, D. L. Favin, H. D. Helms, R. A. Kaenel, W. W. Lang, G. C. Maling, Jr., D. E. Nelson, C. M. Rader and P. D. Welch: What is the fast Fourier transform?, IEEE Trans. Audio Electroacoust, Vol. 15, pp. 45-55 (1967).

[8] R. C. Agarwal: An efficient formulation of the mixed-radix FFT algorithm, Proc. International Conference on Computers, Systems and Signal Processing, pp. 769-772 (1984).

[9] R. C. Agarwal and J. W. Cooley: Vectorized mixed radix discrete Fourier transform algorithms, Proc. IEEE, Vol. 75, pp. 1283-1292 (1987).