## 1.7 Inverse FFT for free

Suppose you programmed some FFT algorithm just for one value of `is`, the sign in the exponent. There is a nice trick that gives the inverse transform for free, if your implementation uses seperate arrays for

real and imaginary part of the complex sequences to be transformed. If your procedure is something like

```
procedure my_fft(ar[], ai[], ldn)  // only for is==+1 !
// real ar[0..2**ldn-1] input, result, real part
// real ai[0..2**ldn-1] input, result, imaginary part
{
    // incredibly complicated code
    // that you can't see how to modify
    // for is==-1
}
```

Then you *don't* need to modify this procedure at all in order to get the inverse transform. If you want the inverse transform somewhere then just, instead of

```
    my_fft(ar[], ai[], ldn)  // forward fft
```

type

```
    my_fft(ai[], ar[], ldn)  // backward fft
```

Note the swapped real- and imaginary parts ! The same trick works if your procedure coded for fixed `is`$= -1$.

To see, why this works, we first note that

$$\begin{aligned}\mathcal{F}[a+ib] &= \mathcal{F}[a_S] + i\sigma\mathcal{F}[a_A] + i\mathcal{F}[b_S] + \sigma\mathcal{F}[b_A] && (1.67)\\ &= \mathcal{F}[a_S] + i\mathcal{F}[b_S] + i\sigma\left(\mathcal{F}[a_A] - i\mathcal{F}[b_A]\right) && (1.68)\end{aligned}$$

and the computation with swapped real- and imaginary parts gives

$$\mathcal{F}[b+ia] = \mathcal{F}[b_S] + i\mathcal{F}[a_S] + i\sigma\left(\mathcal{F}[b_A] - i\mathcal{F}[a_A]\right) \qquad (1.69)$$

... but these are implicitely swapped at the end of the computation, giving

$$\mathcal{F}[a_S] + i\mathcal{F}[b_S] - i\sigma\left(\mathcal{F}[a_A] - i\mathcal{F}[b_A]\right) = \mathcal{F}^{-1}[a+ib] \qquad (1.70)$$

When the type `Complex` is used then the best way to achieve the inverse transform may be to reverse the sequence according to the symmetry of the FT ([FXT: `reverse_nh` in `aux/copy.h`], reordering by $k \mapsto k^{-1} \bmod n$). While not really 'free' the additional work shouldn't matter in most cases.

With real-to-complex FTs (R2CFT) the trick is to reverse the imaginary part after the transform. Obviously for the complex-to-real FTs (R2CFT) one has to reverse the imaginary part before the transform. Note that in the latter two cases the modification does not yield the inverse transform but the one with the 'other' sign in the exponent. Sometimes it may be advantageous to reverse the input of the R2CFT before transform, especially if the operation can be fused with other computations (e.g. with copying in or with the revbin-permutation).

Jörg Arndt
arndt@jjj.de

This document[1] was LaTeX'd at September 26, 2002

**Algorithm** : Classical radix-2 IFFT

```
Input: The evaluations $f(\omega^{\sigma(j\cdot 2m+0)}), f(\omega^{\sigma(j\cdot 2m+1)}), \ldots, f(\omega^{\sigma(j\cdot 2m+2m-1)})$
    of some polynomial with coefficients in a ring $R$ with primitive $n$th root
    of unity $\omega$. Here, $m$ is a power of two where $2m \leq n$.
Output: $(2m) \cdot f \bmod (x^{2m} - \omega^{\sigma(j)})$.
0. If $(2m) = 1$ then return $f \bmod (x - \omega^{\sigma(j)}) = f(\omega^{\sigma(j)})$.
1. Compute the IFFT of $f(\omega^{\sigma(j\cdot 2m+0)}), f(\omega^{\sigma(j\cdot 2m+1)}), \ldots, f(\omega^{\sigma(j\cdot 2m+m-1)})$
       to obtain $f_Y = m \cdot f \bmod (x^m - \omega^{\sigma(2j)})$.
2. Compute the IFFT of $f(\omega^{\sigma(j\cdot 2m+m)}), f(\omega^{\sigma(j\cdot 2m+m+1)}), \ldots, f(\omega^{\sigma(j\cdot 2m+2m-1)})$
       to obtain $f_Z = m \cdot f \bmod (x^m - \omega^{\sigma(2j+1)})$.
3. Compute $f_A = (\omega^{\sigma(2j)})^{-1} \cdot (f_Y - f_Z)$.
4. Compute $f_B = f_Y + f_Z$.
5. Return $(2m) \cdot f \bmod (x^{2m} - \omega^{\sigma(j)}) = f_A \cdot x^m + f_B$.
```

Figure 4.1 Pseudocode for classical radix-2 IFFT

```
Algorithm : Twisted radix-2 IFFT
Input: The evaluations $f(\omega^{\sigma(j\cdot 2m+0)}), f(\omega^{\sigma(j\cdot 2m+1)}), \ldots, f(\omega^{\sigma(j\cdot 2m+2m-1)})$
    of some polynomial with coefficients in a ring $R$.
    Here $R$ has a $n$th root of unity $\omega$, and $m$ is a power of two where $2m \leq n$.
Output: $2m \cdot f(\omega^{\sigma(j)/(2m)} \cdot x) \bmod (x^{2m} - 1)$.
0.  If $(2m) = 1$ then return $f(\omega^{\sigma(j)} \cdot x) \bmod (x - 1) = f(\omega^{\sigma(j)})$.
1.  Compute the IFFT of $f(\omega^{\sigma(j\cdot 2m+0)}), f(\omega^{\sigma(j\cdot 2m+1)}), \ldots, f(\omega^{\sigma(j\cdot 2m+m-1)})$
        to obtain $f_Y = m \cdot f(\omega^{\sigma(2j)/m} \cdot x) \bmod (x^m - 1)$.
2.  Compute the IFFT of $f(\omega^{\sigma(j\cdot 2m+m)}), f(\omega^{\sigma(j\cdot 2m+m+1)}), \ldots, f(\omega^{\sigma(j\cdot 2m+2m-1)})$
        to obtain $m \cdot f(\omega^{\sigma(2j+1)/m} \cdot x) \bmod (x^m - 1)$.
3.  Twist $m \cdot f(\omega^{\sigma(2j+1)/m} \cdot x) \bmod (x^m - 1)$ by $\omega^{-\sigma(1)/m}$
        to obtain $f_Z = m \cdot f(\omega^{\sigma(2j)/m} \cdot x) \bmod (x^m + 1)$.
4.  Compute $f_A = f_Y - f_Z$.
5.  Compute $f_B = f_Y + f_Z$.
6.  Return $(2m) \cdot f(\omega^{\sigma(j)/(2m)} \cdot x) \bmod (x^{2m} - 1) = f_A \cdot x^m + f_B$.
```

Figure 4.2   Pseudocode for twisted radix-2 IFFT

**Algorithm** : Split-radix IFFT (conjugate-pair version)

Input: The evaluations $f(\omega^{\sigma'(j\cdot 4m+0)}), f(\omega^{\sigma'(j\cdot 4m+1)}), \ldots, f(\omega^{\sigma'(j\cdot 4m+4m-1)})$
of some polynomial with coefficients in a ring $R$.
Here $R$ has a $n$th root of unity $\omega$, and $m$ is a power of two where $4m \leq n$.

Output: $(4m) \cdot f(\omega^{\sigma'(j)/(4m)} \cdot x) \bmod (x^{4m} - 1)$.

```
0A. If $(4m) = 1$, then return $f(\omega^{\sigma'(j)} \cdot x) \bmod (x-1) = f(\omega^{\sigma'(j)})$.
0B. If $(4m) = 2$, then call a radix-2 IFFT algorithm to compute the result.
1.  Compute the IFFT of $f(\omega^{\sigma'(j\cdot 4m)}), f(\omega^{\sigma'(j\cdot 4m+1)}), \ldots, f(\omega^{\sigma'(j\cdot 4m+2m-1)})$
        to obtain $f_W \cdot x^m + f_X = (2m) \cdot f(\omega^{\sigma'(2j)/(2m)} \cdot x) \bmod (x^{2m} - 1)$ .
2.  Compute the IFFT of $f(\omega^{\sigma'(j\cdot 4m+2m)}), f(\omega^{\sigma'(j\cdot 4m+2m+1)}), \ldots, f(\omega^{\sigma'(j\cdot 4m+3m-1)})$
        to obtain $m \cdot f(\omega^{\sigma'(4j+2)/m} \cdot x) \bmod (x^m - 1)$.
3.  Compute the IFFT of $f(\omega^{\sigma'(j\cdot 4m+3m)}), f(\omega^{\sigma'(j\cdot 4m+3m+1)}), \ldots, f(\omega^{\sigma'(j\cdot 4m+4m-1)})$
        to obtain $m \cdot f(\omega^{\sigma'(4j+3)/m} \cdot x) \bmod (x^m - 1)$.
4.  Compute $f_Y = m \cdot f(\omega^{\sigma'(4j)/m} \cdot x) \bmod (x^m - \mathrm{I})$
        by twisting $m \cdot f(\omega^{\sigma'(4j+2)/m} \cdot x) \bmod (x^m - 1)$ by $\omega^{-\sigma'(2)/m}$.
5.  Compute $f_Z = m \cdot f(\omega^{\sigma'(4j)/m} \cdot x) \bmod (x^m + \mathrm{I})$
        by twisting $m \cdot f(\omega^{\sigma'(4j+3)/m} \cdot x) \bmod (x^m - 1)$ by $\omega^{\sigma'(2)/m}$.
6.  Compute $f_\alpha = \mathrm{I} \cdot (f_Y - f_Z)$.
7.  Compute $f_\beta = f_Y + f_Z$.
8.  Compute $f_A = f_W + f_\alpha$.
9.  Compute $f_B = f_X - f_\beta$.
10. Compute $f_C = f_W - f_\alpha$.
11. Compute $f_D = f_X + f_\beta$.
12. Return $(4m) \cdot f(\omega^{\sigma'(j)/(4m)} \cdot x) \bmod (x^{4m} - 1) = f_A \cdot x^{3m} + f_B \cdot x^{2m} + f_C \cdot x^m + f_D$.
```

Figure 4.3 Pseudocode for split-radix IFFT (conjugate-pair version)

```
Algorithm : New twisted radix-3 IFFT

Input: The evaluations $f(\omega^{\Delta'(j\cdot 3m+0)}), f(\omega^{\Delta'(j\cdot 3m+1)}), \ldots, f(\omega^{\Delta'(j\cdot 3m+3m-1)})$
    of some polynomial with coefficients in a ring $R$.
    Here $R$ has a $n$th root of unity $\omega$, and $m$ is a power of three where $3m \leq n$.

Output: $(3m) \cdot f(\omega^{\Delta'(j)/(3m)} \cdot x) \bmod (x^{3m} - 1)$.

0.  If $(3m) = 1$ then return $f(\omega^{\Delta'(j)} \cdot x) \bmod (x - 1) = f(\omega^{\Delta'(j)})$.
1.  Compute the IFFT of $f(\omega^{\Delta'(j\cdot 3m+0)}), f(\omega^{\Delta'(j\cdot 3m+1)}), \ldots, f(\omega^{\Delta'(j\cdot 3m+m-1)})$
        to obtain $f_X = m \cdot f(\omega^{\Delta'(3j)/m} \cdot x) \bmod (x^m - 1)$.
2.  Compute the IFFT of $f(\omega^{\Delta'(j\cdot 3m+m)}), f(\omega^{\Delta'(j\cdot 3m+m+1)}), \ldots, f(\omega^{\Delta'(j\cdot 3m+2m-1)})$
        to obtain $\widetilde{f_Y} = m \cdot f(\omega^{\Delta'(3j+1)/m} \cdot x) \bmod (x^m - 1)$.
3.  Compute the IFFT of $f(\omega^{\Delta'(j\cdot 3m+2m)}), f(\omega^{\Delta'(j\cdot 3m+2m+1)}), \ldots, f(\omega^{\Delta'(j\cdot 3m+3m-1)})$
        to obtain $\widetilde{f_Z} = m \cdot f(\omega^{\Delta'(3j+2)/m} \cdot x) \bmod (x^m - 1)$.
4.  Let $\zeta = \omega^{\Delta'(1)/m}$.
5.  Compute $(f_\gamma)_d = \zeta^{-d} \cdot (\widetilde{f_Y})_d + \overline{\zeta^{-d}} \cdot (\widetilde{f_Z})_d$ for all $d$ in $0 \leq d < m$.
        Combine the $(f_\gamma)_d$'s to obtain $f_\gamma = f_Y + f_Z$.
6.  Compute $(f_\beta)_d = \Omega^2 \cdot \zeta^{-d} \cdot (\widetilde{f_Y})_d + \overline{\Omega^2 \cdot \zeta^{-d}} \cdot (\widetilde{f_Z})_d$ for all $d$ in $0 \leq d < m$.
        Combine the $(f_\beta)_d$'s to obtain $f_\beta = \Omega^2 \cdot f_Y + \Omega \cdot f_Z$.
7.  Compute $f_\alpha = f_\beta + f_\gamma = -\Omega \cdot f_Y - \Omega^2 \cdot f_Z$.
8.  Compute $f_A = f_X - f_\alpha = f_X + \Omega \cdot f_Y + \Omega^2 \cdot f_Z$.
9.  Compute $f_B = f_X + f_\beta = f_X + \Omega^2 \cdot f_Y + \Omega \cdot f_Z$.
10. Compute $f_C = f_X + f_\gamma = f_X + f_Y + f_Z$.
11. Return $(3m) \cdot f(\omega^{\Delta'(j)/(3m)} \cdot x) \bmod (x^{3m} - 1) = f_A \cdot x^{2m} + f_B \cdot x^m + f_C$.
```

Figure 4.4 Pseudocode for new twisted radix-3 IFFT

```
Algorithm : Wang-Zhu-Cantor additive IFFT

Input: The evaluations $f(\varpi_{j\cdot 2m+0}), f(\varpi_{j\cdot 2m+1}), \ldots, f(\varpi_{j\cdot 2m+2m-1})$
    of some polynomial $f$ with coefficients in a finite field $\mathbb{F}$
    with $n = 2^k$ elements. Here, $m = 2^i$ and $2m \leq n$.

Output: $f \bmod (s_{i+1} - \varpi_j)$.

0.  If $(2m) = 1$, then return $f \bmod (x - \varpi_j) = f(\varpi_j)$.
1.  Compute the IFFT of $f(\varpi_{j\cdot 2m+0}), f(\varpi_{j\cdot 2m+1}), \ldots, f(\varpi_{j\cdot 2m+m-1})$
        to obtain $r = f \bmod (s_i - \varpi_{2j})$.
2.  Compute the IFFT of $f(\varpi_{j\cdot 2m+m}), f(\varpi_{j\cdot 2m+m+1}), \ldots, f(\varpi_{j\cdot 2m+2m-1})$
        to obtain $f \bmod (s_i - \varpi_{2j+1})$.
3.  Compute $q = f \bmod (s_i - \varpi_{2j+1}) - f \bmod (s_i - \varpi_{2j})$.
4.  Return $f \bmod (s_{i+1} - \varpi_j) = q \cdot (s_i - \varpi_{2j}) + r$.
```

Figure 4.5 Pseudocode for Wang-Zhu-Cantor additive IFFT

```
Algorithm : New additive IFFT
Input: The evaluations $f(\varpi_{j\cdot\eta+0}), f(\varpi_{j\cdot\eta+1}), \ldots, f(\varpi_{j\cdot\eta+\eta-1})$
    of some polynomial $f$ with coefficients in a finite field $\mathbb{F}$
    with $n = 2^k$ elements. Here, $\eta$ is a power of two.
Output: $f$ mod $(x^\eta - x - \varpi_j)$.
0.  If $\eta = 2$, then return $(f(\varpi_{2j+1}) - f(\varpi_{2j})) \cdot (x - \varpi_{2j}) + f(\varpi_{2j})$.
1.  Set $\tau = \sqrt{\eta}$.
2.  for $\phi = j \cdot \tau$ to $j \cdot \tau + \tau - 1$ do
3.      Recursively call the IFFT algorithm with input
            $f(\varpi_{\phi\cdot\tau}), f(\varpi_{\phi\cdot\tau+1}), \cdots, f(\varpi_{\phi\cdot\tau+\tau-1})$
            to obtain $f$ mod $(x^\tau - x - \varpi_\phi)$.
4.  end for (Loop $\phi$)
5.  Assign the coefficient of $x^\lambda$ from $f$ mod $(x^\tau - x - \varpi_\phi)$ to $h_\lambda(\varpi_\phi)$
        for each $\phi$ in $j \cdot \tau \leq \phi < (j+1) \cdot \tau$ and each $\lambda$ in $0 \leq \lambda \leq \tau - 1$.
6.  for $\lambda = 0$ to $\tau - 1$ do
7.      Recursively call the new IFFT algorithm with input
            $h_\lambda(\varpi_{j\cdot\tau}), h_\lambda(\varpi_{j\cdot\tau+1}), \cdots, h_\lambda(\varpi_{j\cdot\tau+\tau-1})$
            to obtain $h_\lambda(x)$ mod $(x^\tau - x - \varpi_j) = h_\lambda(x)$.
8.  end for (Loop $\lambda$)
9.  Construct $g(y)$ using the coefficients of $h_\lambda(x)$ for each $\lambda$ in $0 \leq \lambda < \tau$.
10. Recover $f$ mod $(x^\eta - x - \varpi_j)$ by computing the inverse Taylor expansion
        of $g(y)$ at $x^\tau$.
11. Return $f$ mod $(x^\eta - x - \varpi_j)$.
```

Figure 4.6 Pseudocode for the new additive IFFT

# FAST FOURIER TRANSFORM ALGORITHMS WITH APPLICATIONS

---

A Dissertation
Presented to
the Graduate School of
Clemson University

---

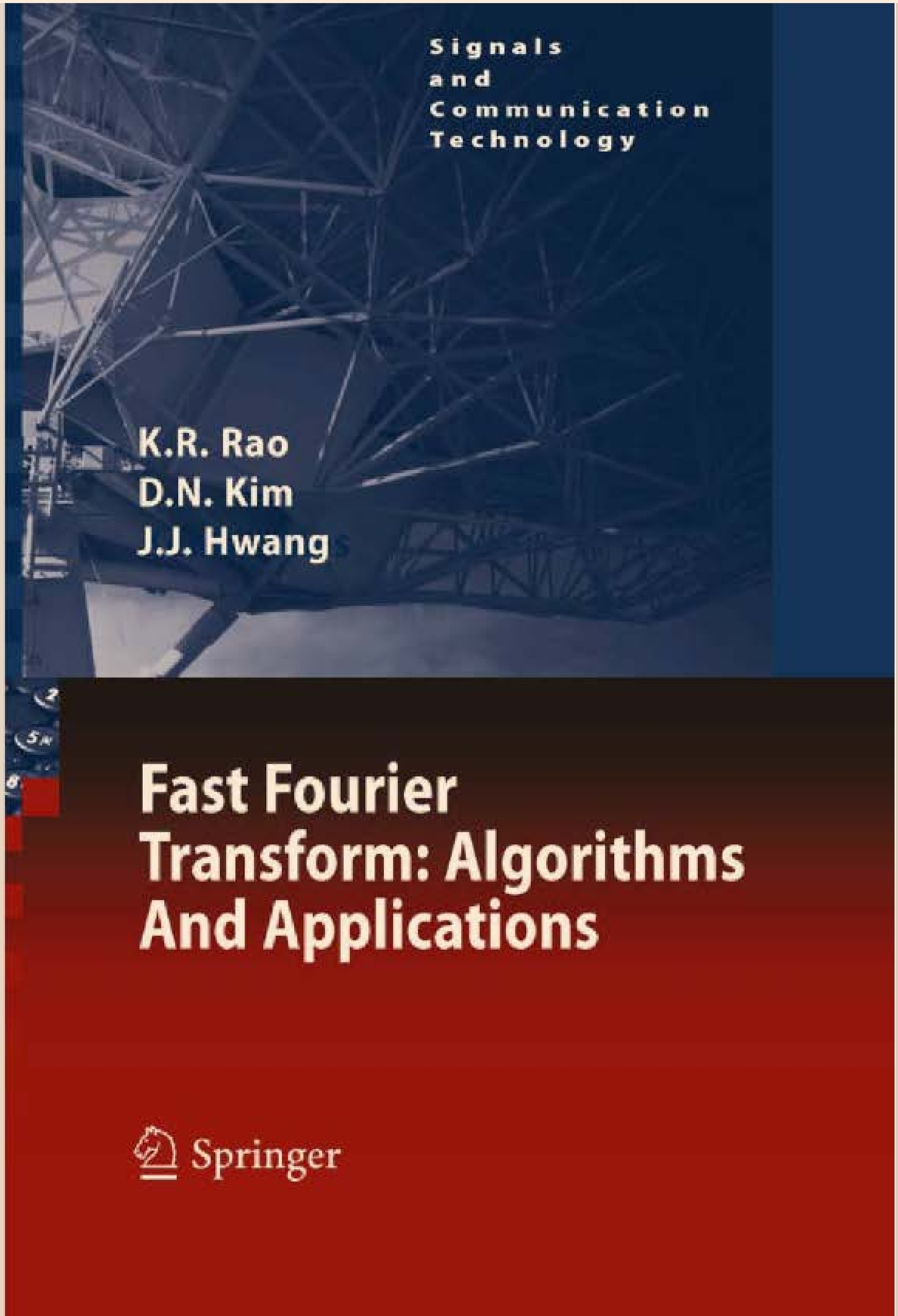
In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Mathematical Sciences

---

by
Todd Mateer
August 2008

---

Accepted by:
Dr. Shuhong Gao, Committee Chair
Dr. Joel Brawley
Dr. Neil Calkin
Dr. Kevin James


Signals and Communication Technology

K.R. Rao
D.N. Kim
J.J. Hwang

Fast Fourier Transform: Algorithms And Applications

Springer

Computationally efficient sparsity-inducing coherence spectrum estimation of complete and non-complete data sets
By Andreas Jakobsson

The n-order iterative schemes for a nonlinear Kirchhoff-Carrier wave equation associated with the mixed inhomogeneous conditions
By Thành Lê Xuân

Accurate inter-laminar recovery for plates and doubly-curved shells with variable radii of curvature using layer-wise theories
By Francesco Tornabene

A statistical-thermodynamic analysis of stably ordered substitutional structures in graphene
By V Tatarenko and Taras Radchenko

Lines 6-8 set up the input data to be a ramp that varies from 0 to N.

```
6.     double x1[] = new double[N];
7.    for (int j=0; j<N; j++)
8.           x1[j] = j;
```

Now the housekeeping. The programmer, interested in keeping copies of the original data, the result of the forward FFT and the result of the inverse FFT, must allocate four arrays! This is an unusual case, as it requires that all intermediate results be kept for checking purposes. Normally, production code would not have to keep all intermediate results.

```
9.     double[] in_r = new double[N];
10.     double[] in_i = new double[N];
```

The in_r and in_i arrays are copies of the input data, with the imaginary component equal to zero. Real data (like audio data) often has a zero imaginary component. There are algorithms that can save significant time by taking advantage of the zero imaginary part of the input data. This requires a different FFT implementation.

```
11.     double[] fftResult_r = new double[N];
12.     double[] fftResult_i = new double[N];

13.     // copy test signal.
14.     in_r = arrayCopy(x1);
```

Line 14 copies the input data into in_r.

```
15.     f.forwardFFT(in_r, in_i);
```

Line 15 replaces in_r and in_i with the forward FFT results.

```
16.     // Copy to new array because IFFT will
17.     // destroy the FFT results.
18.     fftResult_r = arrayCopy(in_r);
19.     fftResult_i = arrayCopy(in_i);
20.     f.reverseFFT(in_r, in_i);
21.     System.out.println("j\tx1[j]\tre[j]\tim[j]\tv[j]");
22.     for(int i=0; i<N; i++) {
23.          System.out.println(
24.          i + "\t" +
25.          x1[i] + "\t" +
26.          fftResult_r[i] + "\t" +
27.             fftResult_i[i] + "\t" +
28.             in_r[i]);
29.     }

30.   }
```

## About the author

**Douglas A. Lyon** (M'89-SM'00) received the Ph.D., M.S. and B.S. degrees in computer and systems engineering from Rensselaer Polytechnic Institute (1991, 1985 and 1983). Dr. Lyon has worked at AT&T Bell Laboratories at Murray Hill, NJ and the Jet Propulsion Laboratory at the California Institute of Technology, Pasadena, CA. He is currently the Chairman of the Computer Engineering Department at Fairfield University, in Fairfield CT, a senior member of the IEEE and President of DocJava, Inc., a consulting firm in Connecticut. Dr. Lyon has authored or co-authored three books (Java, Digital Signal Processing, Image Processing in Java and Java for Programmers). He has authored over 40 journal publications. Email: lyon@docjava.com. Web: http://www.DocJava.com.

JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©JOT, 2009

Vol. 8, No. 5, July-August 2009

# The Discrete Fourier Transform, Part 2: Radix 2 FFT

**By Douglas Lyon**

**Abstract**

This paper is part 2 in a series of papers about the Discrete Fourier Transform (DFT) and the Inverse Discrete Fourier Transform (IDFT). The focus of this paper is on a fast implementation of the DFT, called the FFT (Fast Fourier Transform) and the IFFT (Inverse Fast Fourier Transform). The implementation is based on a well-known algorithm, called the Radix 2 FFT, and requires that its' input data be an integral power of two in length.

Part 3 of this series of papers, demonstrates the computation of the PSD (Power Spectral Density) and applications of the DFT and IDFT. The applications include filtering, windowing, pitch shifting and the spectral analysis of re-sampling.

## 1 THE FFT

Given a sampled waveform

$$v_j, j \in [0 \ldots N-1] \tag{1}$$

The Continuous Time Fourier Transform (CTFT) is defined by:

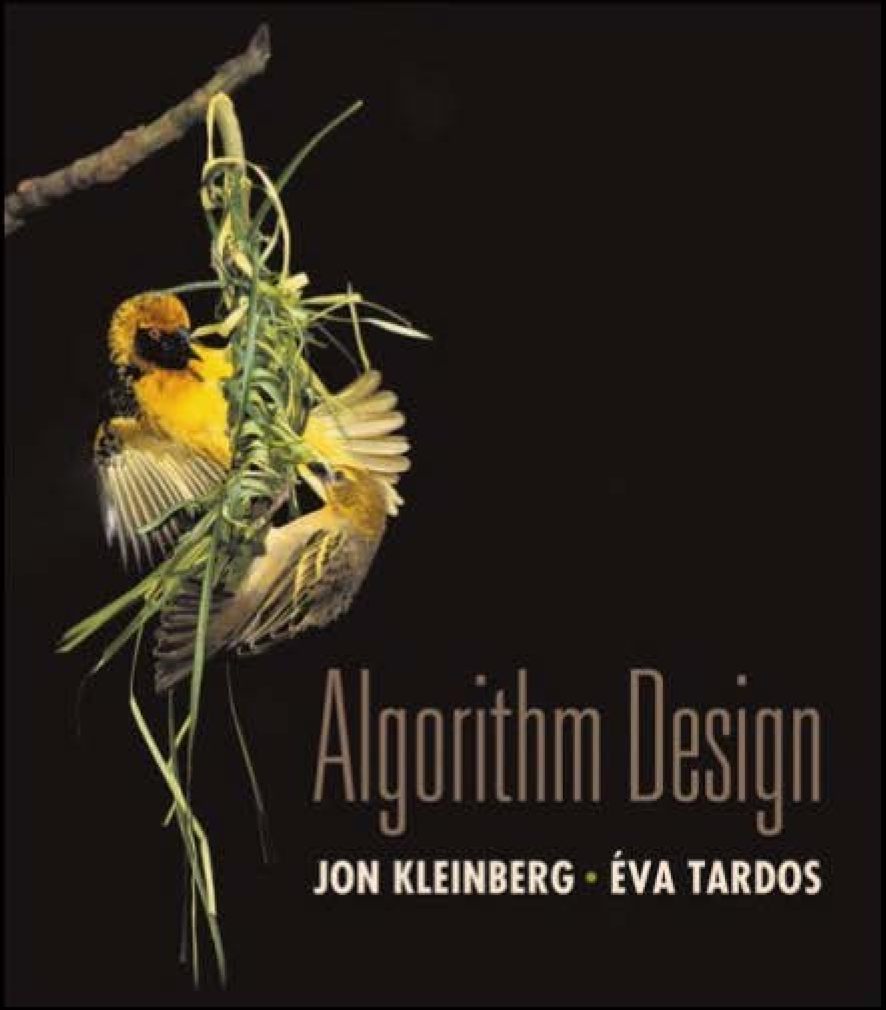
## Inverse FFT: Algorithm

```
ifft(n, $a_0, a_1, \ldots, a_{n-1}$) {
    if (n == 1) return $a_0$

    ($e_0, e_1, \ldots, e_{n/2-1}$) ← FFT(n/2, $a_0, a_2, a_4, \ldots, a_{n-2}$)
    ($d_0, d_1, \ldots, d_{n/2-1}$) ← FFT(n/2, $a_1, a_3, a_5, \ldots, a_{n-1}$)

    for k = 0 to n/2 - 1 {
        $\omega^k$ ← $e^{-2\pi ik/n}$
        $y_{k+n/2}$ ← ($e_k + \omega^k d_k$) / n
        $y_{k+n/2}$ ← ($e_k - \omega^k d_k$) / n
    }

    return ($y_0, y_1, \ldots, y_{n-1}$)
}
```



# How to Multiply

integers, matrices, and polynomials

Slides by Kevin Wayne.
Copyright © 2005 Pearson-Addison Wesley.
All rights reserved.

```
1  template<int R> void
2  FftShMemCol(int sign, int N, int strideO,
3                   float2* dataI, float2* dataO ){
4    float2 v[R];
5    int strideI = B.x*T.x;
6    int idxI = (((b.y*N+t.y)*B.x+b.x)*T.x)+t.x;
7    int incI = T.y*strideI;
8    for( int r=0; r<R; r++ )
9      v[r] = data[idxI + r*incI];
10   DoFft( v, R, N, t.y, T.x );
11   if( strideO < strideI ) {
12     int i = t.y,  j = (idxI%strideI)/strideO;
13     angle = sign*2*M_PI*j/(N*strideI/strideO);
14     for( int r=0; r<R; r++ ) {
15       v[r] *= (cos(i*angle),sin(i*angle));
16       i += T.y;
17     }
18   }
19   int incO = T.y*strideO;
20   int idxO = b.y*R*incI+expand(idxI%incI,incO,R);
21   if( strideO == 1 ) {
22     int idxD = t.x*N + t.y;
23     int idxS = t.y*T.x + t.x;
24     incO = T.y*T.x;
25     idxD = (b.y*R.x+b.x)*N + idxS;
26     exchange( v,R,1, idxD,T.y, idxS,incO );
27   }
28   float s = (sign < 1) ? 1 : 1/N;
29   for( int r=0; r<R; r++ )
30     data[idxO + r*incO] = s*v[r];
31 }
```

Fig. 5. Pseudo-code for shared memory radix-$R$ FFT along columns used with the hierarchical FFT. `strideI` and `strideO` are the strides of sequence elements on input and output. The kernel is invoked with $T_x$ set to a multiple of $R$ not smaller than $CW$, $T_y = N/R$, and $B = (\texttt{strideI}/T_x, M/\texttt{strideI})$. The twiddle stage (lines 11–18) and the transposes (lines 19–27) of the hierarchical algorithm are also included in the kernel.

global memory FFT, we set the number of threads $T$ to $T = \max(\lceil 64 \rceil_{R^i}, N/R)$. These lower bounds on the thread count also ensure that when the data is read from global memory (lines 4–6), it will be read in contiguous segments greater than $CW$ in length. However, when $T > N/R$, the data must first be exchanged between threads. In this case, the kernel computes more than one FFT at a time and the number of thread blocks used are reduced accordingly. The data is then restored to its original order to produce large contiguous segments when written back to global memory. When $T = N/R$, no data exchange is required after reading from global memory. Because the data is always written back to the same location from which it was read, the FFT can be performed in-place. As mentioned previously, bank conflicts that occur when $R$ is a power of two can be handled with appropriate padding.

The large number of registers available on NVIDIA GPUs relative to the size of shared memory can be exploited to increase performance. Because the data held by each thread can be stored entirely in registers (in the array v), the FFT in each iteration (line 26) can be computed without reading or writing data to memory, and is therefore faster. Shared memory is used only to exchange data between registers of different threads. If the number of registers were smaller, then the data would have to reside primarily in shared memory. Additional memory might be required for intermediate results. In particular, the Stockham formulation would require at least twice the amount of shared memory due to the fact that it is performed out-of-place. Larger memory requirements reduce the maximum $N$ that can be handled.

## C. Hierarchical FFT

The shared memory FFT is fast but limited in the sizes it can handle. The hierarchical FFT computes the FFT of a large sequence by combining FFTs of subsequences that are small enough to be handled in shared memory. This is analogous to how the shared memory algorithm computes an FFT of length $N$ by utilizing multiple FFTs of length $R$ that are performed entirely in registers. Suppose we have an array $A$ of length $N = N_\alpha N_\beta$. We first consider a variation of the standard "four-step" hierarchical FFT algorithm [23]:

1) Treating $A$ as $N_\alpha \times N_\beta$ array (row-major), perform $N_\alpha$ FFTs of size $N_\beta$ along the columns.
2) Multiply each element $A_{ij}$ in the array with twiddle factors $\omega = e^{\pm 2\pi ij/N}$ ($-$ for a forward transform, $+$ for the inverse).
3) Perform $N_\beta$ FFTs of size $N_\alpha$ along the rows.
4) Transpose $A$ from $N_\alpha \times N_\beta$ to $N_\beta \times N_\alpha$

$N_\beta$ is chosen to be small enough that the FFT can be performed in shared memory. If $N_\alpha$ is too large to fit into shared memory, then the algorithm recurses, treating each row of length $N_\alpha$ as an $N_{\alpha\alpha} \times N_{\alpha\beta}$ array, etc. One way to think about this algorithm is that it wraps the original one dimensional array of length $N$ into multiple dimensions, each small enough that the FFT can be performed in shared memory along that dimension. The dimensions are then transformed from highest to lowest. The effect of the multiple transposes that occur when coming out of the recursion is to reverse the order of the dimensions, which is analogous to bit-reversal. The original "four-step" algorithm swaps steps 3 and 4. The end result is the same, except that FFTs are always performed along columns. For example, suppose $A$ is partitioned wrapped into a 3D array with dimensions $(N_1, N_2, N_3)$. The execution of the original and the modified algorithms can be depicted as follows:

$$\begin{array}{ll}
(\underline{N_1, N_2}, N_3') & (\underline{N_1, N_2}, N_3') \\
(\underline{N_1}, N_2', N_3) & (N_3, \underline{N_1}, N_2') \\
(N_1', N_2, N_3) & (N_3, N_2, N_1') \\
(N_3, N_2, N_1) &
\end{array}$$

where $'$ indicates an FFT transformation along the specified dimension. The $i$ index in step 2 corresponds to an element's index in the transformed dimension ($N_\alpha$) and the $j$ index corresponds to the concatenation of the indices in the underlined dimensions ($N_\beta$). The original algorithm (left) performs all of the FFTs in-place and uses a series of transposes at the end to reverse the order of the dimensions. The entire algorithm can be performed in-place if the transposes are performed in-place. In-place algorithms can be important for large data sizes. In the modified algorithm, the FFT computation always takes

```
void exchange( float2* v, int R, int stride,
                  int idxD, int incD,
                  int idxS, int incS ){
  float* sr = shared, *si = shared+T*R;
  __syncthreads();
  for( int r=0, ; r<R; r++ ) {
    int i = (idxD + r*incD)*stride;
    (sr[i], si[i]) = v[r];
  }
  __syncthreads();
  for( r=0; r<R; r++ ) {
    int i = (incS + r*incS)*stride;
    v[r] = (sr[i], si[i]);
  }
}
```

Fig. 3. Function for exchanging the $R$ values in v between $T$ threads. The real and imaginary components of v are stored in separate arrays to avoid bank-conflicts. The second synchronization avoids read-after-write data hazards. The first synchronization is necessary to avoid data hazards only when `exchange()` is invoked multiple times.

satisfy the requests. For both sets of coalescing requirements, the greatest bandwidth is achieved when the accesses are contiguous and properly aligned.

Assuming that the number of threads per block $T = N/R$ is no less than $CW$, our mapping of threads to elements in the Stockham formulation ensures that the reads from global memory are in contiguous segments of at least $CW$ in length (line 23 in Fig. 2). If the radix $R$ is a power of two, the reads are also properly aligned. Writes are not contiguous for the first $\lceil \log_R CW \rceil$ iterations where $N_s < CW$ (line 29), although under the assumption that $T \geq CW$, when all the writes have completed, the memory areas touched do contain contiguous segments of sufficient length. Therefore, we handle this problem by first exchanging data between threads using shared memory so that it can then be written out in larger contiguous segments to global memory. We do this by replacing lines 28–29 with the following:

```
int idxD = (t/Ns)*R + (t%Ns);
exchange( v, R, 1, idxD,Ns,  t,T );
idxD = b*T*R + t;
for( int r=0; r<R; r++ )
  data1[idxD+r*T] = v[r];
```

The pseudo-code for `exchange()` can be found in Fig. 3.

To maximize the reuse of data read from global memory and to reduce the total number of iterations, it is best to use a radix $R$ that is as large as possible. However, the size of $R$ is limited by the number of registers and the size of the shared memory on the multiprocessors. Reducing the number of threads reduces the total number of registers and the amount of shared memory used, but with too few threads there are not enough warps to hide memory latency. We have found that using $T = \max(\lceil 64 \rceil_{R^i}, N/R)$ produces good results, where $\lceil x \rceil_{R^i}$ represents the smallest power of $R$ not less than $x$.

*Bank conflicts:* Shared memory on current GPUs is organized into 16 banks with 32-bit words distributed round-robin between them. Accesses to shared memory are serviced for groups of 16 threads at a time (half-warps). If any of the threads in a half-warp access the same memory bank

```
1  template<int R> void
2  FftShMem(int sign, int N, float2* data){
3    float2 v[R];
4    int idxG = b*N + t;
5    for( int r=0; r<R; r++ )
6      v[r] = data[idxG + r*T];
7    if( T == N/R )
8      DoFft( v, R, N, t );
9    else {
10     int idx = expand(t,v,N/R,R);
11     exchange(v,R,1, idx,N/R, t,T );
12     DoFft( v, R, N, t );
13     exchange(v,R,1, t,T, idx,N/R );
14   }
15   float s = (sign < 1) ? 1 : 1/N;
16   for( int r=0; r<R; r++ )
17     data[idxG + r*T] = s*v[r];
18 }
19
20 void DoFft(float2* v, int R, int N,
21              int j, int stride=1) {
22   for( int Ns=1; Ns<N; Ns*=R ){
23     float angle = sign*2*M_PI*(j%Ns)/(Ns*R);
24     for( int r=0; r<R; r++ )
25       v[r] *= (cos(r*angle), sin(r*angle));
26     FFT<R>( v );
27     int idxD = expand(j,Ns,R);
28     int idxS = expand(j,N/R,R);
29     exchange( v,R,stride, idxD,Ns, idxS,N/R );
30   }
31 }
```

Fig. 4. Pseudo-code for shared memory radix-$R$ FFT. This kernel is used when $N$ is small enough that the entire FFT can be performed using just shared memory and registers.

at the same time, a conflict occurs, and the simultaneous accesses must be serialized, which degrades performance. In order to avoid bank conflicts, `exchange()` writes the real and imaginary components to separate arrays with stride 1 instead of a single array of `float2`. When a `float2` is written to shared memory, the two components are written separately with stride 2, resulting in bank conflicts. The call to `exchange()` still results in bank conflicts when $R$ is a power of two and $N_s < 16$. The solution is to pad with $N_s$ empty values between every 16 values. For $R = 2$ the extra cost of computing the padded indexes actually outweighs the benefit of avoiding bank conflicts, but for radix-4 and radix-8, the net gain is significant. Padding requires extra shared memory. To reduce the amount of shared memory by a factor of 2, it is possible to exchange only one component at a time. This requires 3 synchronizations instead of 1, but can result in a net gain in performance because it allows more in-flight threads. When $R$ is odd, padding is not necessary because $R$ is relatively prime w.r.t. the number of banks.

## B. Shared Memory FFT

For small $N$, we can perform the entire FFT using only shared memory and registers without writing intermediate results back to global memory. This can result in substantial performance improvements. The pseudo-code for our shared memory kernel is shown in Fig. 4. As with the that support writing multiple values to the same location in multiple buffers can save the redundant reads, but must either use more complex indexing when accessing the values written in a preceding iteration, or after each iteration, they must copy the values to their proper location in a separate pass [15], which consumes bandwidth. Thus scatter is important for conserving memory bandwidth.

Fig. 2 also shows pseudo-code for an implementation of the FFT on a GPU which supports scatter. The main difference between `GPU_FFT()` and `CPU_FFT()` is that the index $j$ into the data is generated as a function of the thread number $t$, the block index $b$, and the number of threads per block $T$ (line 13). Also, the iteration over values of $N_s$ are generated by multiple invocations of `GPU_FFT()` rather than in a loop (line 3) because a global synchronization between threads is needed between the iterations, and for many GPUs the only global synchronization is kernel termination.

For each invocation of `GPU_FFT()`, $T$ is set to $N/R$ and the number of thread blocks $B$ is set to $M$, where $M$ is the number of FFTs to process simultaneously. Processing multiple FFTs at the same time is important because the number of warps used for small-sized FFTs may not be sufficient to achieve full utilization of the multiprocessor or to hide memory latency while accessing global memory. Processing more than one FFT results in more warps and alleviates these problems.

Despite the fact that `GPU_FFT()` uses scatter, it still has a number of performance issues. First, the writes to memory have coalescing issues. The memory subsystem tries to coalesce memory accesses from multiple threads into a smaller number of accesses to larger blocks of memory. But the space between consecutive accesses generated during first few iterations (small $N_s$) is too large for coalescing to be effective (line 29). Second, the algorithm does not exploit low-latency shared memory to improve data reuse. This is also a problem for traditional GPGPU implementations as well, because the graphics APIs do not provide access to shared memory. Finally, to handle arbitrary lengths, we would need to write a separate specialization for all possible radices $R$. This is impractical, especially for large $R$. In the next section we will discuss how we address each of these issues.

Because GPUs vary in shared memory sizes, memory, and processor configurations, the FFT algorithms should ideally be parametrized and auto-tuned across different algorithm variants and architectures.

## IV. FFT Algorithms

In this section, we present several FFT algorithms — a global memory algorithm that works well for larger FFTs with higher radices on architectures with high memory bandwidth, a shared memory algorithm for smaller FFTs, a hierarchical FFT that exploits shared memory by decomposing large FFTs into a sequence of smaller ones, mixed-radix FFTs that handle sizes that are multiples of small prime factors, and an implementation of Bluestein's algorithm for handling larger prime factors. We also discuss extensions to handle multi-dimensional FFTs, real FFTs, and discrete cosine transforms (DCTs).

```
1   float2* CPU_FFT(int N, int R,
2                   float2* data0, float2* data1) {
3     for( int Ns=1; Ns<N; Ns*=R ) {
4       for( int j=0; j<N/R; j++ )
5         FftIteration( j, N, R, Ns, data0, data1 );
6       swap( data0, data1 );
7     }
8     return data0;
9   }
10
11  void GPU_FFT(int N, int R, int Ns,
12               float2* dataI, float2* dataO) {
13    int j = b*N + t;
14    FftIteration( j, N, R, Ns, dataI, dataO );
15  }
16
17  void FftIteration(int j, int N, int R, int Ns,
18                    float2* data0, float2*data1){
19    float2 v[R];
20    int idxS = j;
21    float angle = -2*M_PI*(j%Ns)/(Ns*R);
22    for( int r=0; r<R; r++ ) {
23      v[r] = data0[idxS+r*N/R];
24      v[r] *= (cos(r*angle), sin(r*angle));
25    }
26    FFT<R>( v );
27    int idxD = expand(j,Ns,R);
28    for( int r=0; r<R; r++ )
29      data1[idxD+r*Ns] = v[r];
30  }
31
32  void FFT<2>( float2* v ) {
33    float2 v0 = v[0];
34    v[0] = v0 + v[1];
35    v[1] = v0 - v[1];
36  }
37
38  int expand(int idxL, int N1, int N2 ){
39    return (idxL/N1)*N1*N2 + (idxL%N1);
40  }
```

Fig. 2. Reference implementation of the radix-$R$ Stockham algorithm. Each iteration over the data combines $R$ subarray of length $N_s$ into arrays of length $RN_s$. The iterations stop when the entire array of length $N$ is obtained. The data is read from memory and scaled by so-called *twiddle factors* (lines 20–25), combined using an $R$-point FFT (line 26), and written back out to memory (lines 27–29). The number of threads used for `GPU_FFT()`, $T$, is $N/R$. The `expand()` function can be thought of as inserting a dimension of length $N_2$ after the first dimension of length $N_1$ in a linearized index.

### A. Global Memory FFT

As mentioned in Section III.B, the pseudo-code for `GPU_FFT()` in Fig. 2 can lead to poor memory access coalescing, which reduces performance. On some GPUs the rules for memory access coalescing are quite stringent. Memory accesses to global memory are coalesced for groups of $CW$ threads at a time, where $CW$ is the coalescing width. $CW$ is 16 for recent NVIDIA GPUs. Coalescing is performed when each thread in the group accesses either a 32-bit, 64-bit, or 128 bit word in sequential order and the address of the first thread is aligned to ($CW \times$ word size). Bandwidth for non-coalesced accesses is about an order of magnitude slower. Later GPUs have more relaxed coalescing requirements. Memory accesses can be coalesced even if they are not sequential, so long as all the threads access the same word size. The hardware issues memory transactions in blocks of 32, 64, or 128 bytes while seeking to minimize the number and size of the transactions to

# High Performance Discrete Fourier Transforms on Graphics Processors

Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli
Microsoft Corporation
{nagag,dalloyd,yurido,burtons,jmanfer}@microsoft.com

***Abstract*—We present novel algorithms for computing discrete Fourier transforms with high performance on GPUs. We present hierarchical, mixed radix FFT algorithms for both power-of-two and non-power-of-two sizes. Our hierarchical FFT algorithms efficiently exploit shared memory on GPUs using a Stockham formulation. We reduce the memory transpose overheads in hierarchical algorithms by combining the transposes into a block-based multi-FFT algorithm. For non-power-of-two sizes, we use a combination of mixed radix FFTs of small primes and Bluestein's algorithm. We use modular arithmetic in Bluestein's algorithm to improve the accuracy. We implemented our algorithms using the NVIDIA CUDA API and compared their performance with NVIDIA's CUFFT library and an optimized CPU-implementation (Intel's MKL) on a high-end quad-core CPU. On an NVIDIA GPU, we obtained performance of up to 300 GFlops, with typical performance improvements of 2–4× over CUFFT and 8–40× improvement over MKL for large sizes.**

## I. Introduction

The Fast Fourier Transform (FFT) refers to a class of algorithms for efficiently computing the Discrete Fourier Transform (DFT). The FFT is used in many different fields such as physics, astronomy, engineering, applied mathematics, cryptography, and computational finance. Some of its many and varied applications include solving PDEs in computational fluid dynamics, digital signal processing, and multiplying large polynomials. Because of its importance, the FFT is used in several benchmarks for parallel computers such as the HPC challenge [1] and NAS parallel benchmarks [2]. In this paper we present algorithms for computing FFTs with high performance on graphics processing units (GPUs).

The GPU is an attractive target for computation because of its high performance and low cost. For example, a $300 GPU can deliver peak theoretical performance of over 1 TFlop/s and peak theoretical bandwidth of over 100 GiB/s. Owens et al. [3] provides a survey of algorithms using GPUs for general purpose computing. Typically, general purpose algorithms for the GPU had to be mapped to the programming model provided by graphics APIs. Recently, however, alternative APIs have been provided that expose low-level hardware features and the performance characteristics of the GPU. We support non-power-of-two sizes using a mixed radix FFT for small primes and Bluestein's algorithm for large primes. We address important performance issues such as memory bank conflicts and memory access coalescing. We also address an accuracy issue in Bluestein's algorithm that arises when using single-precision arithmetic. We perform comparisons with NVIDIA's CUFFT library and Intel's Math Kernel Library (MKL) on a high end PC. On data residing in GPU memory, our library achieves up to 300 GFlops at factory core clock settings, and overclocking we achieve 340 GFlops. We obtain typical performance improvements of 2–4× over CUFFT and 8–40× over MKL for large sizes. We also obtain significant improvements in numerical accuracy over CUFFT.

The rest of the paper is organized as follows. After discussing related work in Section II we present an overview of mapping FFT computation to the GPU in Section III. We then present our algorithms in Section IV and implementation details in Section V. We compare results with other FFT implementation in Section VI and then conclude with some ideas for future work.

## II. Related Work

A large body of research exists on FFT algorithms and their implementations on various architectures. Sorensen and Burrus compiled a database of over 3400 entries on efficient algorithms for the FFT [8]. We refer the reader to the book by Van Loan [9] which provides a matrix framework for understanding many of the algorithmic variations of the FFT. The book also touches on many important implementation issues.

The research most related to our work involves accelerating FFT computation by using commodity hardware such as GPUs or Cell processors. Most implementations of the FFTs on the GPU use graphics APIs such as current versions of OpenGL or DirectX [10], [11], [12], [13], [14], [15]. However, these APIs do not directly support scatters, access to shared memory,

**IFFT Algorithm with Pseudo-Code**

Distortion is a really simple algorithm. Each "distorted" sample can be computed with only the value of the original sample. The original sample changes only if it is on the wrong side of the "threshold."

*FFT Frequency Analysis Pseudo-Code*

In this object-based pseudo-code for frequency analysis, it is assumed that the FFT conversion is already implemented.

```
Variables
      input_pointer;
      total_points;
      real_in[MAX], Ar[MAX];
      imag_in[MAX], Ai[MAX];

Constructor()

set_size(int size)
      total_points=size;
      input_pointer=0;

add(x sample)
      real_in[input_pointer]=x;
      imag_in[input_pointer]=0;
      input_pointer=input_pointer+1;


perform_FFT ()
      // Perform the FFT algorithm previously
      // described, using real_in and imag_in
      // as the complex inputs and using
      // Ar and Ai to store the outputs
      FFT()

      input_pointer=0

get_out_power(x integer)
      return sqrt(Ar[x]*Ar[x]+Ai[x]*Ai[x])
```