



Fast Fourier Transform Algorithms

- Swetha Gurram
- Mohammed Yousuf

Introduction

- Discrete Fourier transform (DFT) takes a discrete signal in the time domain and transforms that signal into its discrete frequency domain representation.
- DFT coefficient $X(k)$ is defined by :

$$X(k) = 1/N \sum_{n=0}^{N-1} x(n) e^{-j2\pi kn/N} = 1/N \sum_{n=0}^{N-1} x(n) W^{kn} \quad \text{for } k=0, 1, 2, \dots, N-1$$

(twiddle factor) $W = e^{-\frac{j2\pi}{N}}$

- Algorithm for DFT is given by :

```
for p = 0:n - 1
    ω ← exp(-2πip/n)
    F(p, 0) ← 1
    for q = 1:n - 1
        F(p, q) ← ω F(p, q - 1)
    end
end
```
- The computation involves $O(n^2)$ operations.

- A fast Fourier transform (FFT) is a quick method for forming the matrix-vector product $F_n x$, where F_n is the discrete Fourier transform matrix.
- By fast/quick we mean speed proportional to $n \log n$.
- The FFT uses a greatly reduced number of arithmetic operations as compared to the computation of DFT.
- The basic approach of the FFT is that it enables us to compute quickly an n -point DFT from a pair of $(n/2)$ -point DFTs.

Comparison between DFT and FFT

- DFT involves $8n^2$ flops as compared to FFT that involves $5n \log n$ flops.

n	$\frac{8n^2}{5n \log_2 n}$
32	≈ 10
1024	≈ 160
32768	≈ 3500
1048576	≈ 84000

- Thus, if it takes one second to compute an FFT of size $n = 1048576$, then it would require about one day to compute conventionally.
- The number of complex arithmetic operations required for FFT are less when compared to the DFT.

Different Fast Fourier Transform

- Radix – 2 Fast Fourier Transform
- Radix – 4 Fast Fourier Transform
- Radix – 8 Fast Fourier Transform
- Mixed Radix Fast Fourier Transform
- Split Radix Fast Fourier Transform

Radix – 2 Fast Fourier Transform

- A radix-2 FFT divides a DFT of size N into two interleaved DFTs (hence the name "radix-2") of size N/2 with each recursive stage.
- Radix-2 first computes the Fourier transforms of the even-indexed numbers and of the odd-indexed numbers, and then combines those two results to produce the Fourier transform of the whole sequence.
- This idea can then be performed recursively to reduce the overall runtime to $O(N \log N)$.

$$\begin{aligned} X_k &= \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N}nk} \\ X_k &= \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-\frac{2\pi i}{N}(2m)k} + \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-\frac{2\pi i}{N}(2m+1)k} \\ &= \sum_{m=0}^{M-1} x_{2m} e^{-\frac{2\pi i}{M}mk} + e^{-\frac{2\pi i}{N}k} \sum_{m=0}^{M-1} x_{2m+1} e^{-\frac{2\pi i}{M}mk} \end{aligned}$$

- Consider a DFT matrix of size $n = 4$

$$F_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix},$$

where $\omega = \omega_4 = \exp(-2\pi i/4) = -i$. Since $\omega^4 = 1$, it follows that

$$F_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & 1 & \omega^2 \\ 1 & \omega^3 & \omega^2 & \omega \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}.$$

Let Π_4 be the 4-by-4 permutation

$$\Pi_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and note that

$$F_4 \Pi_4 = \left[\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & -1 & -i & i \\ \hline 1 & 1 & -1 & -1 \\ 1 & -1 & i & -i \end{array} \right]$$

- The key is to regard this permutation of F_4 as a 2 by 2 block matrix.

$$\Omega_2 = \begin{bmatrix} 1 & 0 \\ 0 & -i \end{bmatrix} = \text{diag}(1, \omega_4)$$

$$F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix},$$

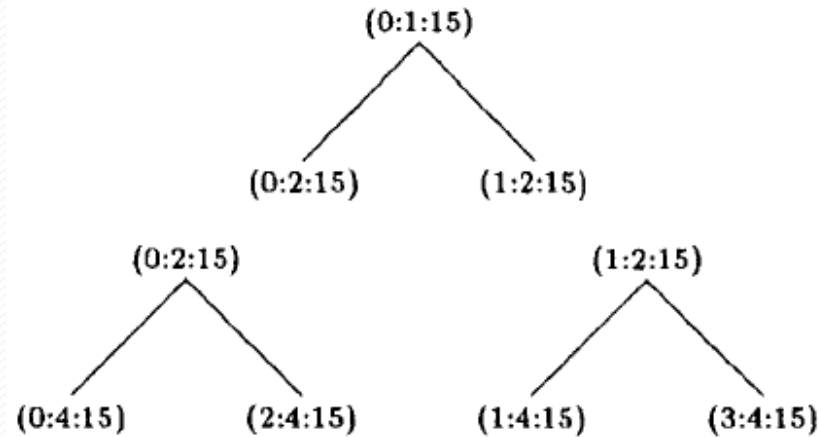
$$F_4 \Pi_4 = \begin{bmatrix} F_2 & \Omega_2 F_2 \\ F_2 & -\Omega_2 F_2 \end{bmatrix}.$$

- Thus, each block is represented in terms of F_2 .
- In general,

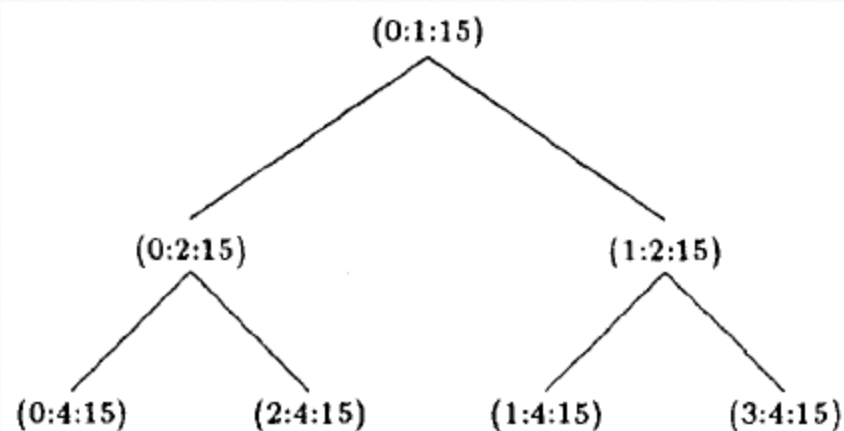
If $n = 2m$ and $x \in \mathbb{C}^n$, then

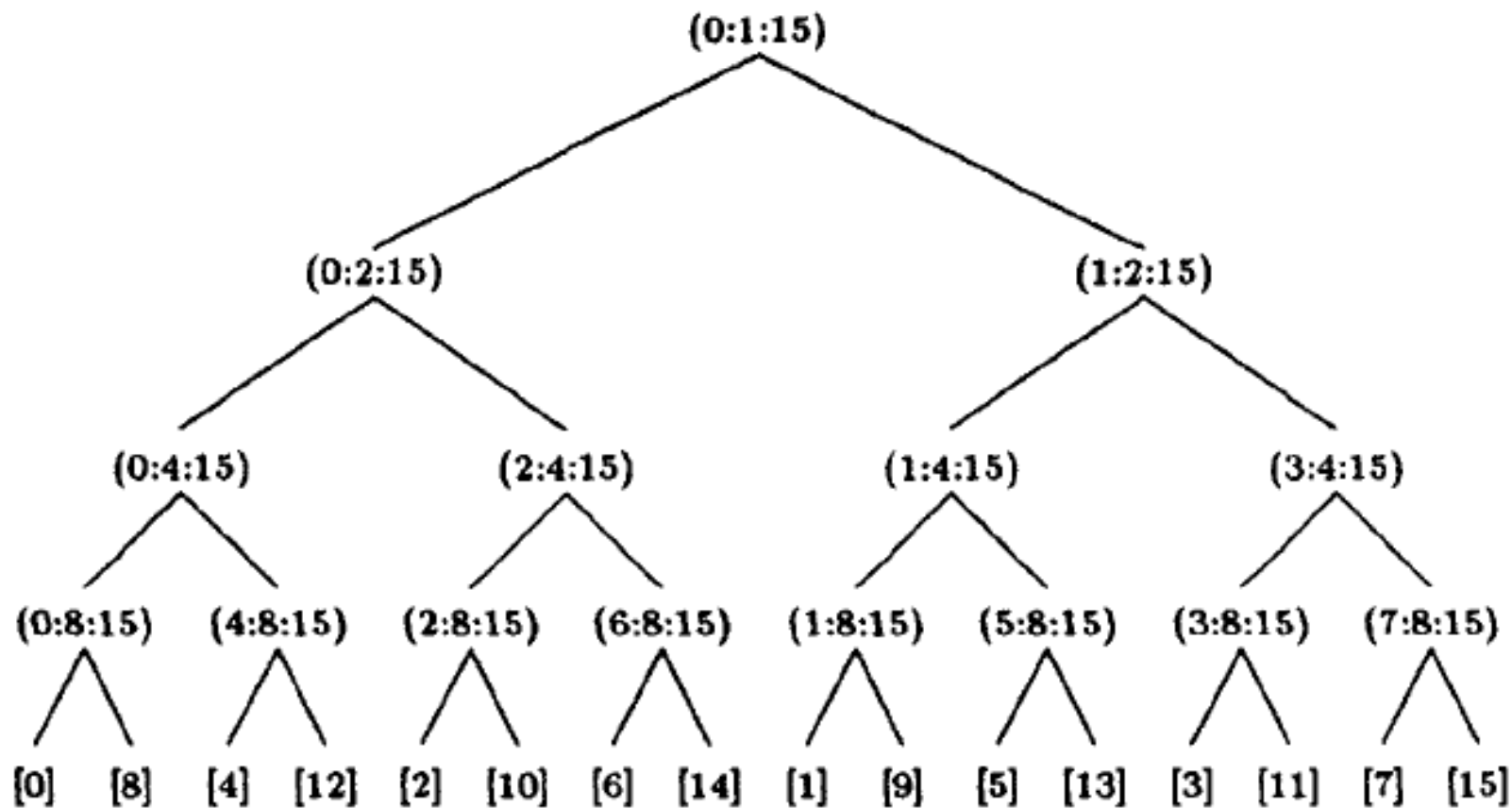
$$F_n x = \begin{bmatrix} I_m & \Omega_m \\ I_m & -\Omega_m \end{bmatrix} \begin{bmatrix} F_m x(0:2:n-1) \\ F_m x(1:2:n-1) \end{bmatrix}$$

- Suppose $n = 16$. Then $F_{16} x$ is a combination of the 8 point DFTs $F_8 x(0:2:15)$ and $F_8 x(1:2:15)$. This dependence can be depicted as :



- Thus $F_{16} x$ can be generated from four quarter length DFTs as follows :





The structure of a radix-2 FFT ($n = 16$)

- A recursive Radix – 2 FFT procedure is given below :

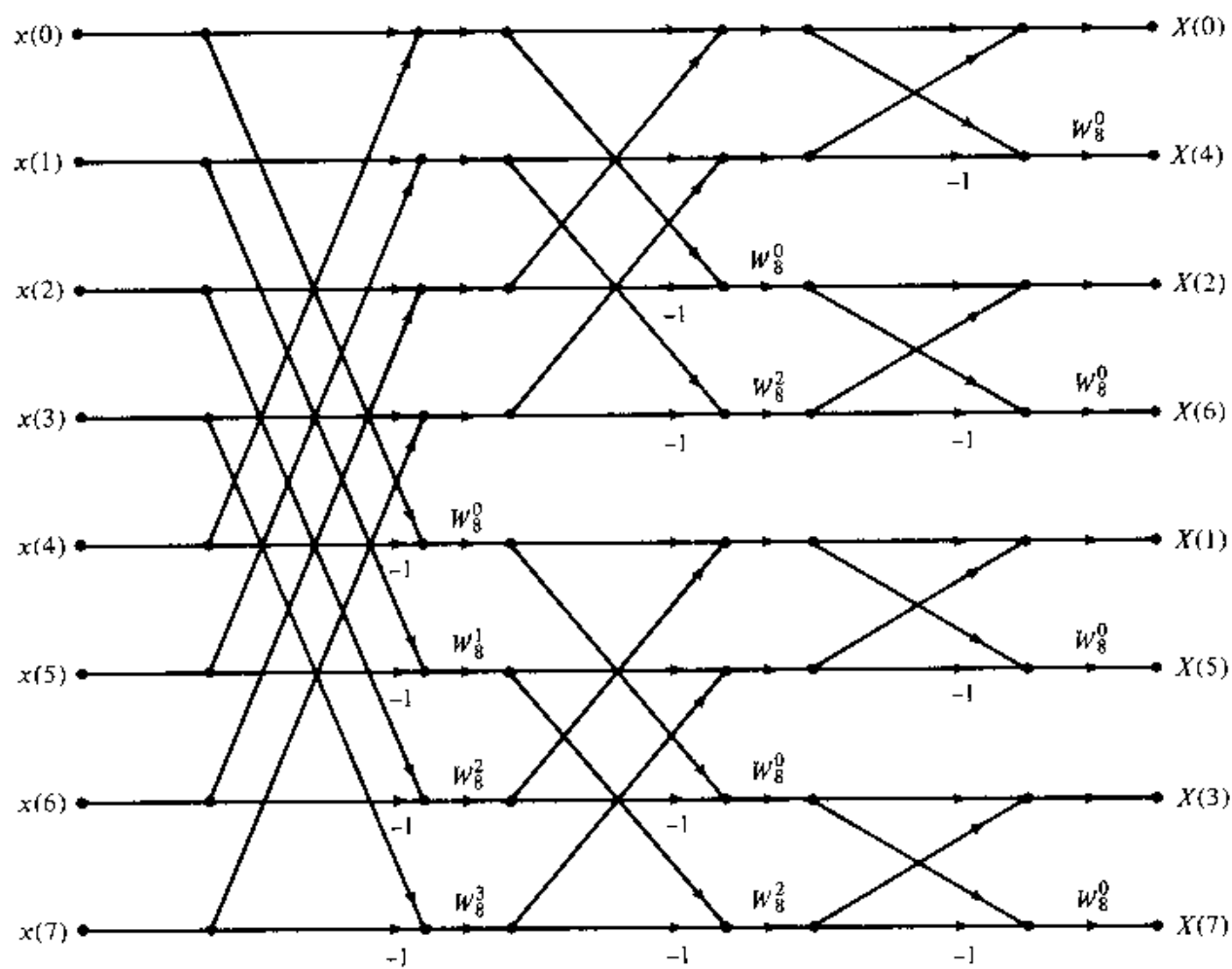
```

function  $y = \text{fft}(x, n)$ 
    if  $n = 1$ 
         $y \leftarrow x$ 
    else
         $m \leftarrow n/2$ 
         $\omega \leftarrow \exp(-2\pi i/n)$ 
         $\Omega \leftarrow \text{diag}(1, \omega, \dots, \omega^{m-1})$ 
         $z_T \leftarrow \text{fft}(x(0:2:n-1), m)$ 
         $z_B \leftarrow \Omega \text{fft}(x(1:2:n-1), m)$ 

         $y \leftarrow \begin{bmatrix} I_m & I_m \\ I_m & -I_m \end{bmatrix} \begin{bmatrix} z_T \\ z_B \end{bmatrix}$ 
    end
end

```

- A Radix – 2 FFT can either be a DIT FFT or a DIF FFT.



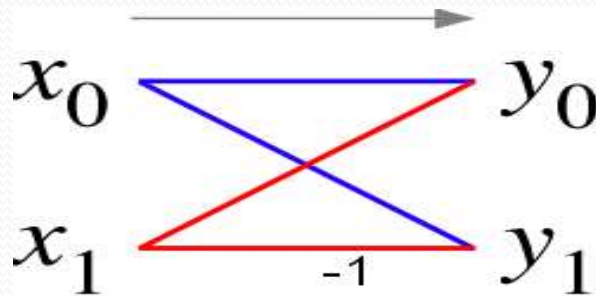
$N=8$ -point decimation-in-frequency FFT algorithm.

Butterfly Operators

- In the context of FFT algorithms, a butterfly is a portion of the computation that combines the results of smaller DFTs into a larger DFT, or vice versa (breaking a larger DFT up into sub transforms).
- In the case of the radix-2 Cooley-Tukey algorithm, the butterfly is simply a DFT of size 2 that takes two inputs (x_0, x_1) and gives two outputs (y_0, y_1) by the formula :

$$y_0 = x_0 + x_1$$

$$y_1 = x_0 - x_1$$



- The radix -2 butterfly matrix B_L is given by :

$$B_L = \begin{bmatrix} I_{L_*} & \Omega_{L_*} \\ I_{L_*} & -\Omega_{L_*} \end{bmatrix},$$

$$\Omega_{L_*} = \text{diag}(1, \omega_L, \dots, \omega_L^{L_*-1})$$

where $L_* = 2L$

$$F_n x = B_n \begin{bmatrix} F_{n/2} x(0:2:n-1) \\ F_{n/2} x(1:2:n-1) \end{bmatrix}$$

$$F_{n/2} x(0:2:n-1) = B_{n/2} \begin{bmatrix} F_{n/4} x(0:4:n-1) \\ F_{n/4} x(2:4:n-1) \end{bmatrix}$$

$$F_{n/2} x(1:2:n-1) = B_{n/2} \begin{bmatrix} F_{n/4} x(1:4:n-1) \\ F_{n/4} x(3:4:n-1) \end{bmatrix}$$

$$F_n x = B_n \begin{bmatrix} B_{n/2} & 0 \\ 0 & B_{n/2} \end{bmatrix} \begin{bmatrix} F_{n/4} x(0:4:n-1) \\ F_{n/4} x(2:4:n-1) \\ F_{n/4} x(1:4:n-1) \\ F_{n/4} x(3:4:n-1) \end{bmatrix}$$

- The two algorithms for the radix -2 Cooley – Tukey Framework include :
 - In – Place Formulation
 - Unit Stride Formulation

```

 $x \leftarrow P_n x$ 
for  $q = 1:t$ 
   $L \leftarrow 2^q$ ;  $r \leftarrow n/L$ ;  $L_* \leftarrow L/2$ 
  for  $j = 0:L_* - 1$ 
     $\omega \leftarrow \cos(2\pi j/L) - i \sin(2\pi j/L)$ 
    for  $k = 0:r - 1$ 
       $\tau \leftarrow \omega \cdot x(kL + j + L_*)$ 
       $x(kL + j + L_*) \leftarrow x(kL + j) - \tau$ 
       $x(kL + j) \leftarrow x(kL + j) + \tau$ 
    end
  end
end
end

```

```

 $x \leftarrow P_n x$ 
 $w \leftarrow w_n^{(long)}$ 
for  $q = 1:t$ 
   $L \leftarrow 2^q$ ;  $r \leftarrow n/L$ ;  $L_* \leftarrow L/2$ 
  for  $k = 0:r - 1$ 
    for  $j = 0:L_* - 1$ 
       $\tau \leftarrow w(L_* - 1 + j) \cdot x(kL + j + L_*)$ 
       $x(kL + j + L_*) \leftarrow x(kL + j) - \tau$ 
       $x(kL + j) \leftarrow x(kL + j) + \tau$ 
    end
  end
end
end

```

- Both procedures involve $5n \log n$ butterfly flops.

Radix – p splitting FFT

- Suppose $n = pm$ with $1 < p < n$. F_n can be expressed as a function of the two smaller DFT matrices F_p and F_m . The result is the radix-p splitting :

$$F_n \Pi_{p,n} = (F_p \otimes I_m) \text{diag}(I_m, \Omega_{p,m}, \dots, \Omega_{p,m}^{p-1}) (I_p \otimes F_m)$$

$$\Omega_{p,m} = \text{diag}(1, \omega_n, \dots, \omega_n^{m-1})$$

- Consider the case of $n = 96$. Then, F_{96} can be expressed in terms of $p=2,3,4,6,8,12,16,24,32$, or 48 smaller DFTs. If we set $p=4$, then the top level synthesis has the form :

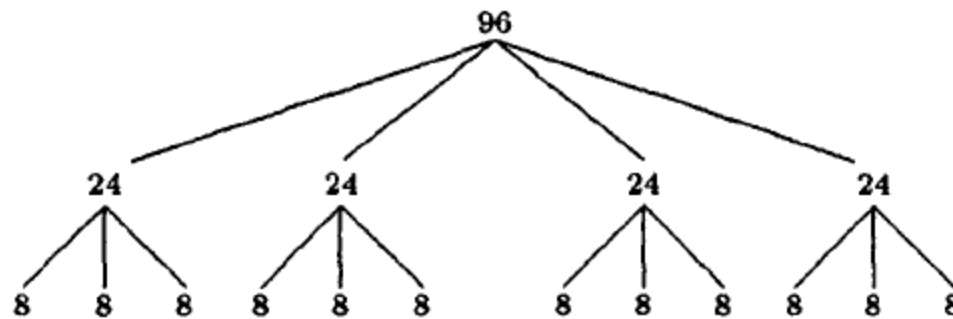
$$F_{96}x \leftarrow \begin{cases} F_{24}x(0:4:95) \\ F_{24}x(1:4:95) \\ F_{24}x(2:4:95) \\ F_{24}x(3:4:95) \end{cases}$$

- Each of the 24-point DFTs can be split in several ways :

$$F_{96}x \leftarrow \left\{ \begin{array}{l} F_{24}x(0:4:95) \leftarrow \left\{ \begin{array}{l} F_{12}x(0:8:95) \\ F_{12}x(4:8:95) \end{array} \right. \\ \\ F_{24}x(1:4:95) \leftarrow \left\{ \begin{array}{l} F_8x(1:12:95) \\ F_8x(5:12:95) \\ F_8x(9:12:95) \end{array} \right. \\ \\ F_{24}x(2:4:95) \leftarrow \left\{ \begin{array}{l} F_6x(2:16:95) \\ F_6x(6:16:95) \\ F_6x(10:16:95) \\ F_6x(14:16:95) \end{array} \right. \\ \\ F_{24}x(3:4:95) \leftarrow \left\{ \begin{array}{l} F_4x(3:24:95) \\ F_4x(7:24:95) \\ F_4x(11:24:95) \\ F_4x(15:24:95) \\ F_4x(19:24:95) \\ F_4x(23:24:95) \end{array} \right. \end{array} \right.$$

Mixed-radix FFT

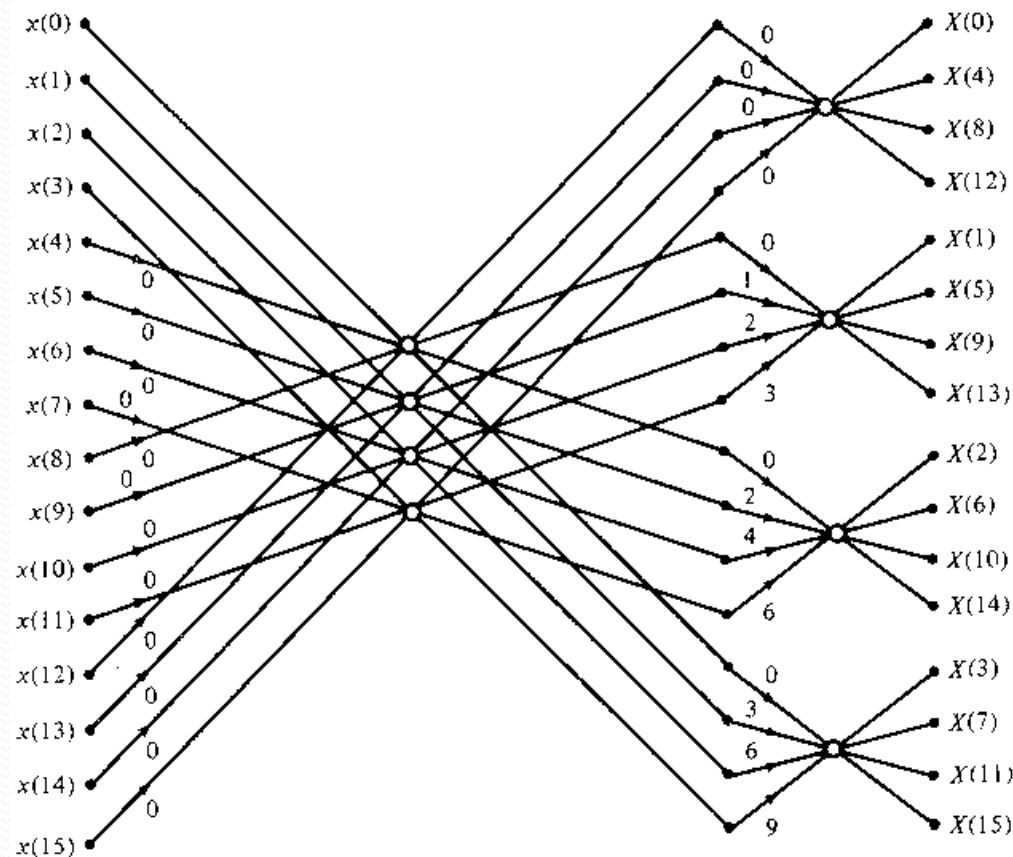
- If the same splitting rules are used at each level, then a mixed-radix framework results. That is, if n is factored the same way across any level of the associated computation tree, then a mixed-radix FFT results.



A mixed-radix framework $(p_1, p_2, p_3) = (8, 3, 4)$

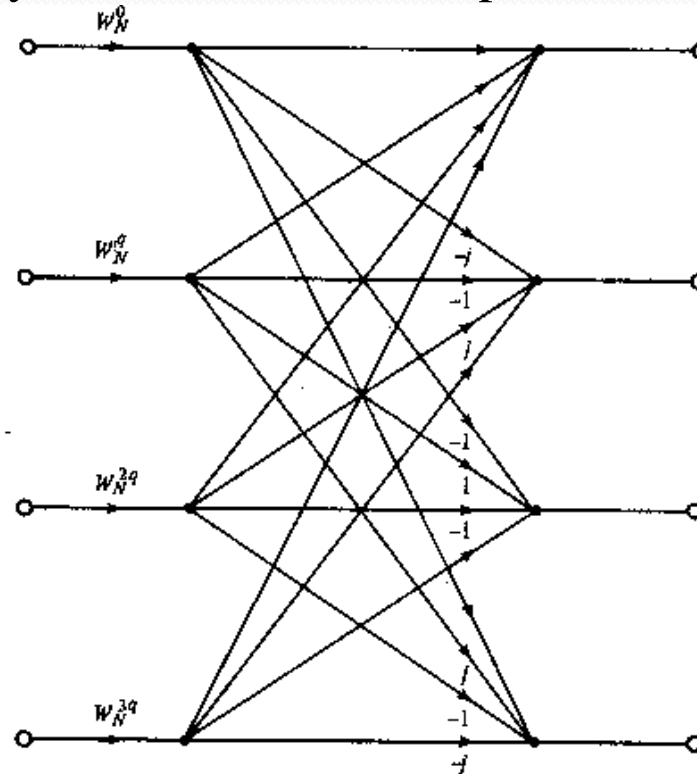
Radix - 4 Fast Fourier Transform

- The radix – 4 approach assumes that $n = 4^t$.

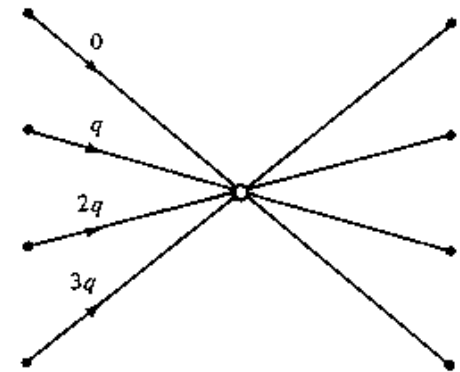


Sixteen-point radix-4 decimation-in-time algorithm with input in normal order and output in digit-reversed order

- Each butterfly involves three complex multiplications and 12 complex divisions.



(a)



(b)

Basic butterfly computation in a radix-4 FFT algorithm.

- The radix – 4 approach requires a total of $4.25n \log n$ flops, which amounts to a 15 percent reduction in flops compared to radix -2 version.

Radix - 8 Fast Fourier Transform

- The radix – 8 approach assumes that $n = 8^t$.
- An 8-point DFT $X = F_8 x$ has the form :

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & a & -i & b & -1 & -a & i & -b \\ 1 & -i & -1 & i & 1 & -i & -1 & i \\ 1 & b & i & a & -1 & -b & -i & -a \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -a & -i & -b & -1 & a & i & b \\ 1 & i & -1 & -i & 1 & i & -1 & -i \\ 1 & -b & i & -a & -1 & b & -i & a \end{bmatrix} \begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \end{bmatrix}$$

- The radix – 8 approach requires a total of $4.08n \log n$ flops, which amounts to a 20 percent reduction in flops compared to radix -2 version.

Data Re-Use and Radix

- The radix-2, 4, and 8 frameworks involve $5n \log n$, $4.25n \log n$ and $4.08n \log n$ flops, respectively, and so higher radix frameworks have added appeal.
- Another advantage concerns the re-use of data.

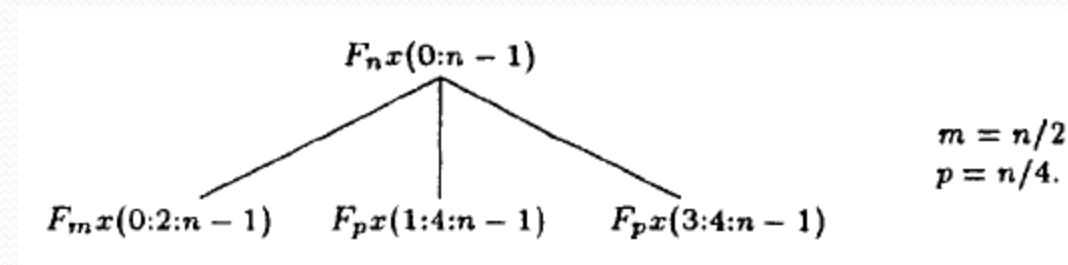
Flops per butterfly.

p	$B_{p,L}$ Flops
2	$5L$
4	$8.5L$
8	$12.25L$

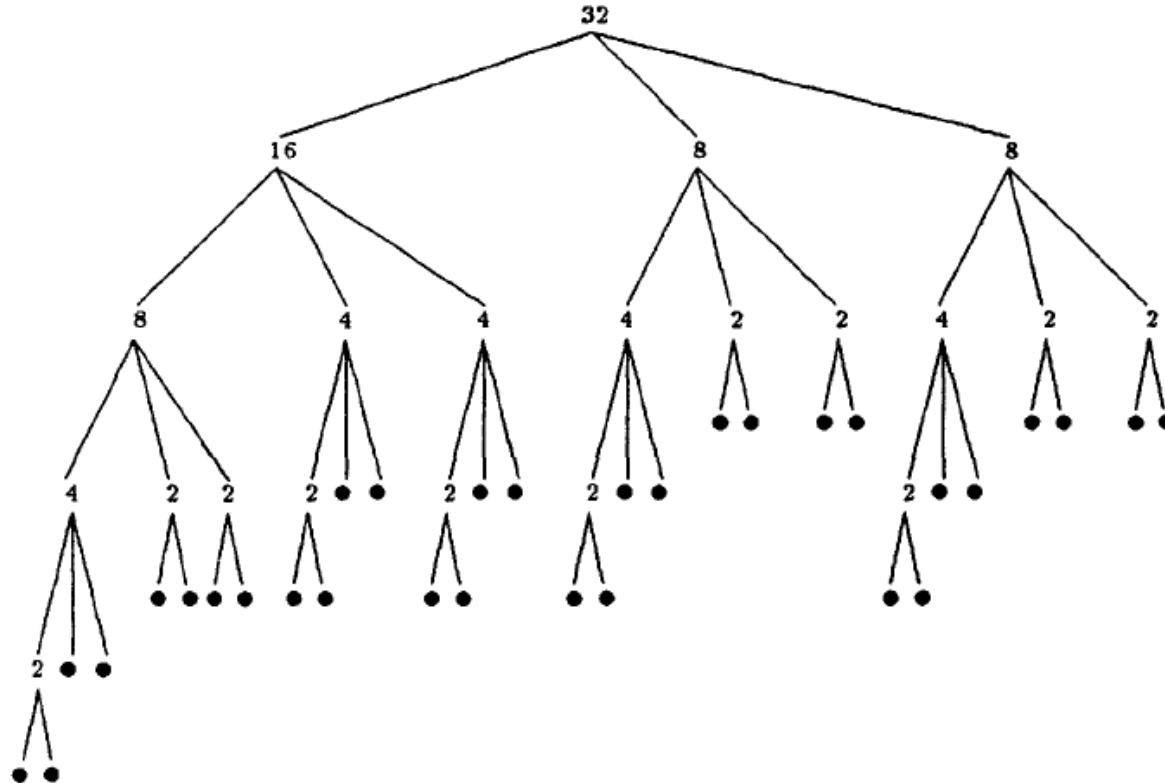
- If $n = 8^t$, then $3t$, $2t$, t passes through the data are required by the radix-2, 4, and 8 frameworks, respectively.
- The reduced memory traffic requirements of the higher radix frameworks makes them attractive in many environments.

Split-radix Fast Fourier Transform

- The split-radix algorithm is based upon a clever synthesis of one half-length DFT together with two quarter-length DFTs.
- The resulting procedure involves less arithmetic than any of the standard radix-2, radix-4 or radix-8 procedures.



- Split-radix computation tree for $n = 32$.



- The computation tree is not balanced and the algorithm does not produce all of the intermediate DFTs that arise in the radix-2 procedures.
- The split-radix approach requires a total of $4n \log n$ flops.

Comparison between different FFTs

	Real Multiplications				Real Additions			
N	Radix-2	Radix-4	Radix-8	Split Radix	Radix-2	Radix-4	Radix-8	Split Radix
16	24	20		20	152	148		148
32	88			68	408			388
64	264	208	204	196	1032	976	972	964
128	72			516	2054			2308
256	1800	1392		1284	5896	5488		5380
512	4360		3204	3076	13566		12420	12292
1024	10248	7856		7172	30728	28336		27652




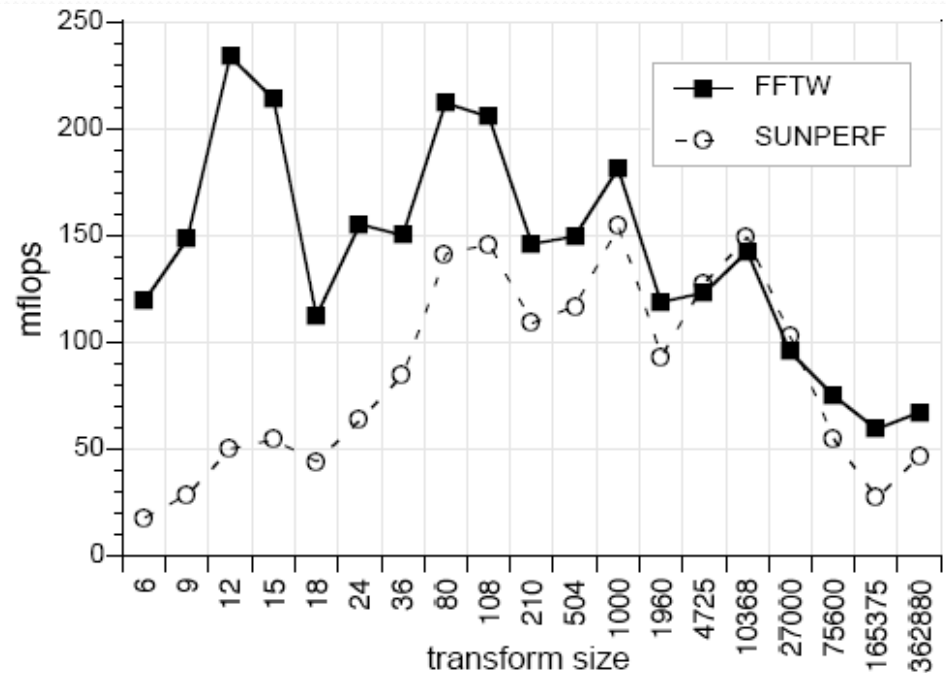
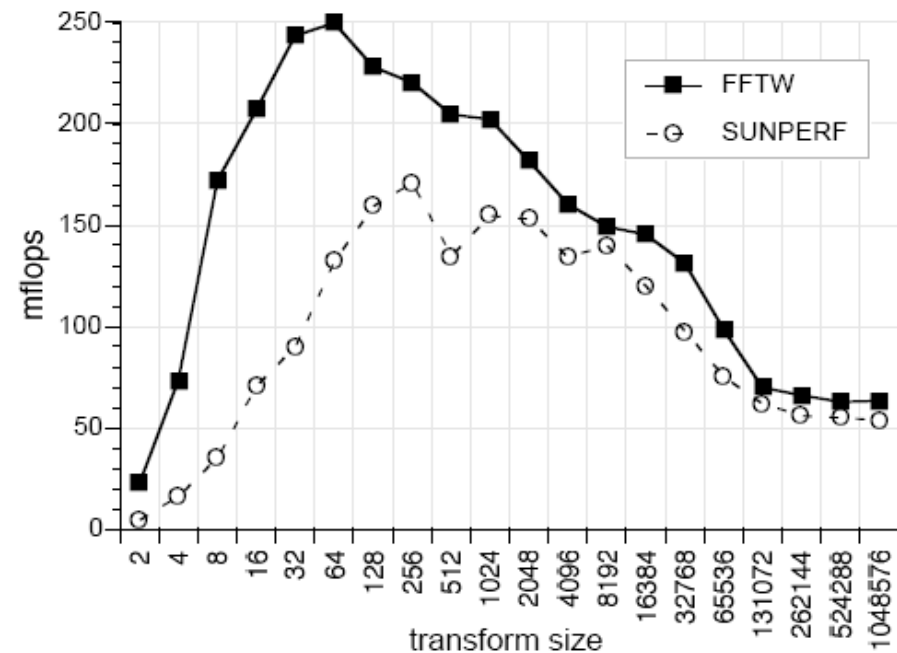
FFTW – Fast Fourier Transform In the West

<http://fftw.org>

Introduction

- FFTW is a C-subroutine library for computing the DFT.
- Developed by Matteo Frigo and Steven G Johnson at the Massachusetts Institute of Technology.
- FFTW is one of the fastest DFT programs and has maximized performance because of two features :
 - It automatically adapts the computation to the hardware,
 - Code is generated automatically by special purpose compiler - genfft written in Objective Caml.

- 
- FFTW does not implement a single DFT algorithm but is structured as a library of codelets (plan).
 - The precise plan depends on the size of the input and which codelets happen to run fast on the underlying hardware.
 - genfft compiler works opposite of the normal compiler and provides optimizations for the DFT programs.
 - genfft discovered algorithms that were previously unknown, and was able to reduce the arithmetic complexity of some other existing algorithms.



Graph of the performance of FFTW versus Sun's Performance Library on a 167 MHz Ultra SPARC processor in single precision. The graph plots the speed in "mflops" (higher is better) versus the size of the transform. The figure on the left shows sizes that are powers of two, while the figure on the right shows other sizes that can be factored into powers of 2, 3, 5, and 7.

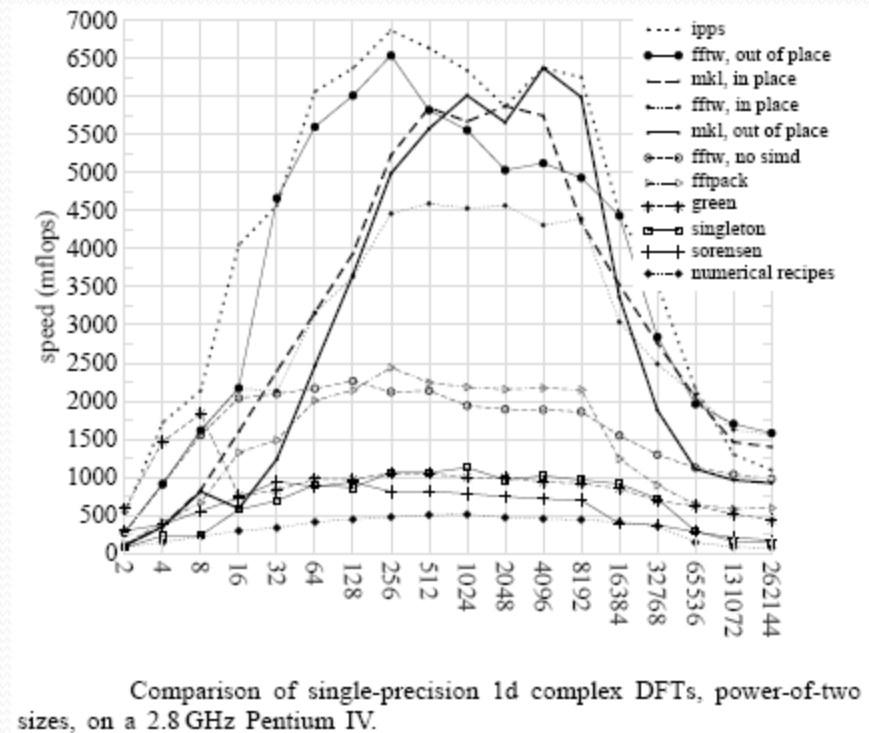
Operation of genfft

genfft operates in four phases :

- Creation phase – genfft produces the directed acyclic graph
- Simplifier phase – use of local rewriting rules in order to simplify it.
- Scheduler phase – produces a schedule of dag, that for transforms of size 2^k , minimizes the number of register spills
- Finally, schedule is unparsed to C.

Features of FFTW

- FFTW is written in portable C and runs well on many architectures and operating systems.
- FFTW computes DFT in $O(n \log n)$ as compared to other DFT implementations $O(n^2)$
- FFTW imposes no restrictions on the rank of multi-dimensional transforms and also supports multiple DFTs.
- FFTW supports DFTs of real data, as well as of discrete cosine / sine transforms.



In addition to FFTW v. 3.0.1, the other codes benchmarked are as follows (some for only one precision or machine): arprec, “four-step” FFT implementation (from the C++ ARPREC library, 2002); cxml, the vendor-tuned Compaq Extended Math Library on Alpha; fftpack, the Fortran library ; green, free code by J. Green (C, 1998); mkl, the Intel Math Kernel Library v. 6.1 (DFTI interface) on the Pentium IV; ipps, the Intel Integrated Performance Primitives, Signal Processing, v. 3.0 on the Pentium IV; numerical recipes, the C four1 routine; oura, a free code by T. Ooura (C and Fortran, 2001); singleton, a Fortran FFT; sorensen, a split-radix FFT; takahashi, the FFTE library v. 3.2 by D. Takahashi (Fortran, 2004) ; and vdsp, the Apple vDSP library on the G5.

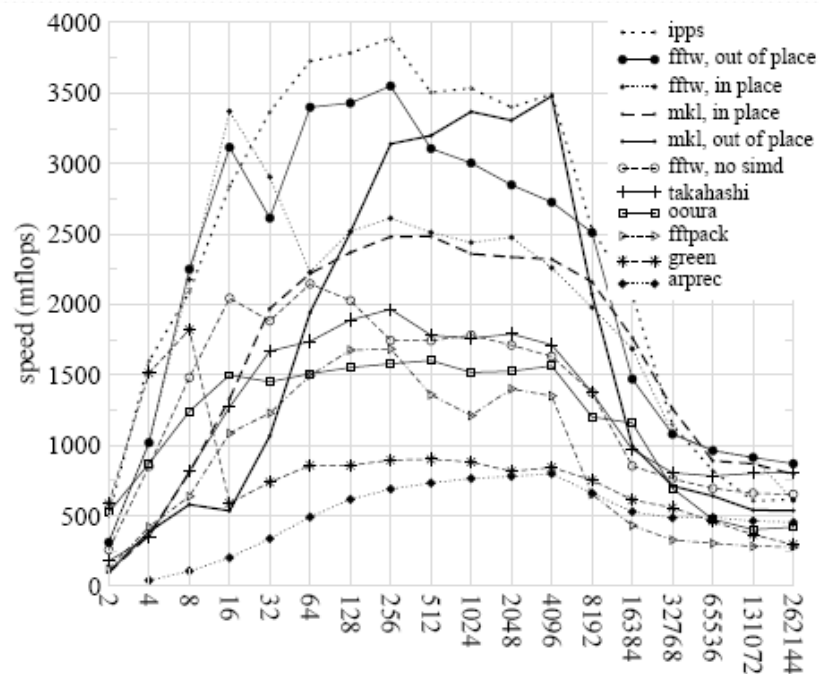


Fig. 1. Comparison of double-precision 1d complex DFTs, power-of-two sizes, on a 2.8 GHz Pentium IV. Intel C/Fortran compilers v. 7.1, optimization flags -O3 -xW (maximum optimization, enable automatic vectorizer).

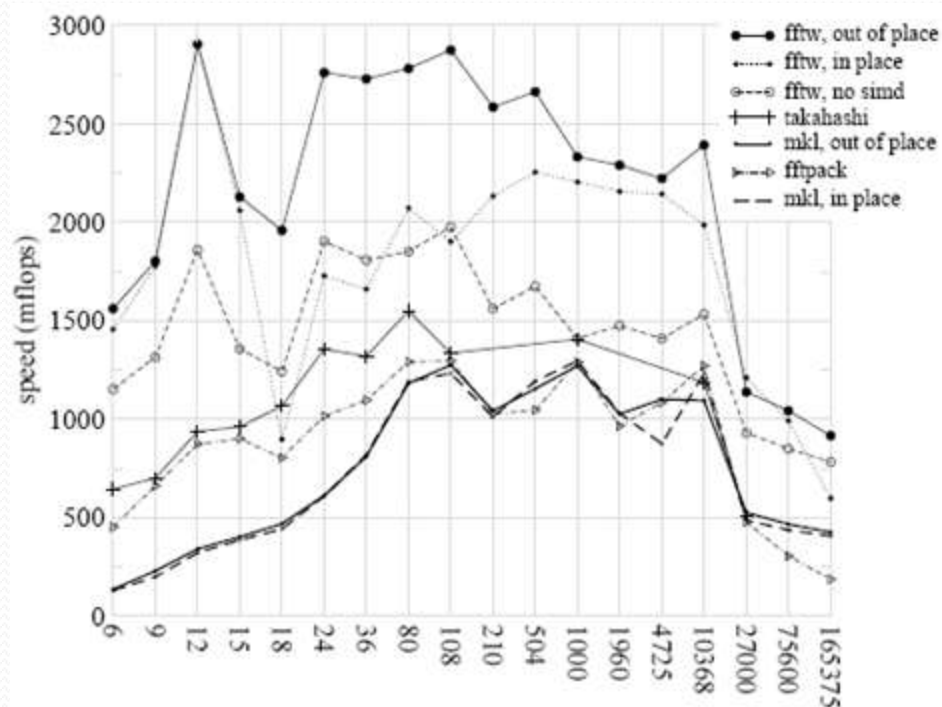
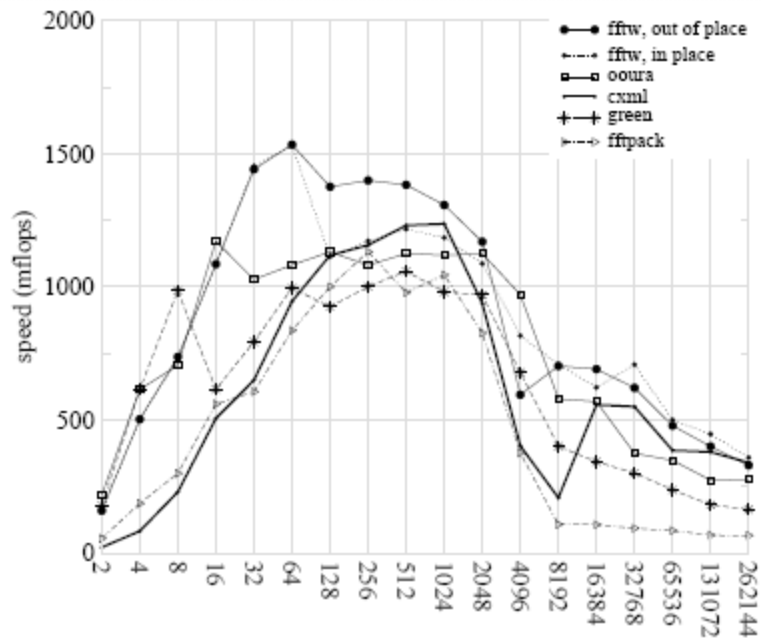
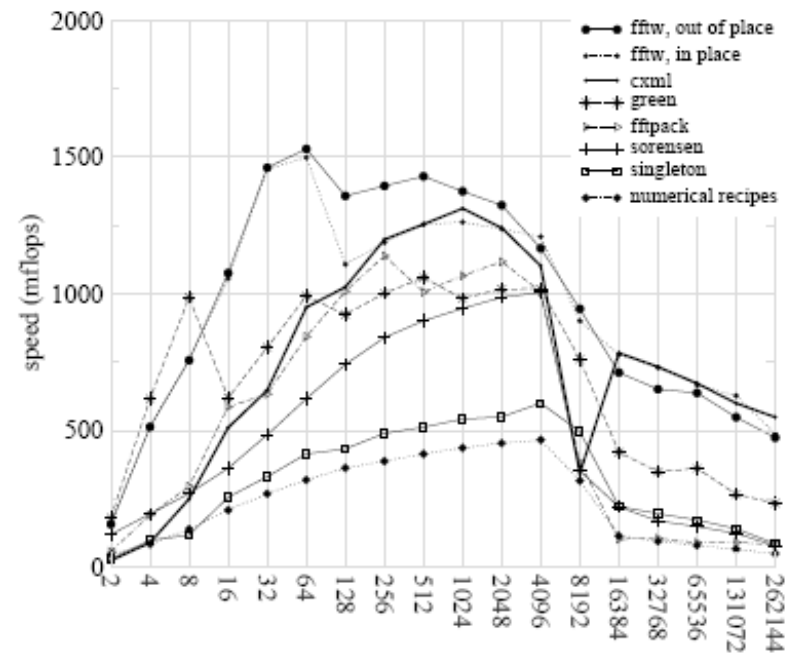


Fig. 2. Comparison of double-precision 1d complex DFTs, non-power-of-two sizes, on a 2.8 GHz Pentium IV.



Comparison of double-precision 1d complex DFTs, power-of-two sizes, on an 833 MHz Alpha EV6. Compaq C V6.2-505. Compaq Fortran X1.0.1-1155. Optimization flags: -newc -w0 -O5 -ansi_alias -ansi_args -fp_reorder -tune host -arch host.



Comparison of single-precision 1d complex DFTs, power-of-two sizes, on an 833 MHz Alpha EV6.

Significance of FFTW

- Performance is maximized with the use of the FFTW.
- Code generation is automated by the genfft compiler.
- Achieving correctness has been surprisingly easy.
- The generator is effective because it can apply problem-specific code improvements.
- Finally, genfft derived new algorithms, as in the example $n=13$.

FMA Optimized FFTs

- Modern processors provide multiply-add or fused multiply-add (FMA) instructions that perform the ternary operation $\pm a \pm b \times c$ in the same amount of time needed for a single floating-point addition or multiplication operation.
- Conventional Cooley-Tukey FFT algorithms require more real additions than real multiplications.
- Implementing FMA Optimized FFTs helped achieve two main goals :
 - Fusing multiplications with additions into FMA instructions, and thus yielding near optimal FMA utilization with respect to FFT algorithms.
 - Using unused multiplication slots to compute twiddle factors instead of loading these constants from memory.
- Achieved an FMA utilization between 87.5% and 100%.
 - <http://www.math.tuwien.ac.at/ascot/>

MATRIX TRANSPOSITION IN A MEMORY HIERARCHY



CPU

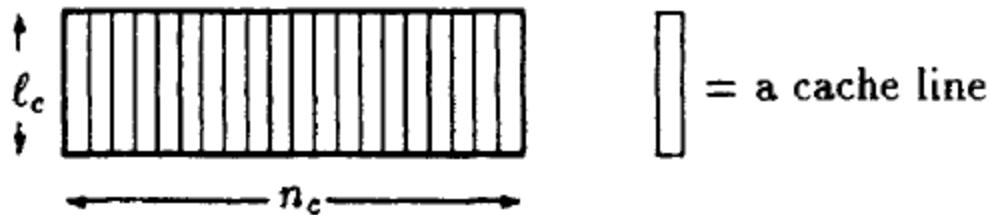
CACHE

Main Memory

External Memory

TRANSPPOSITION WITH A CACHE

- Cache line has length l_c and number of lines is n_c



$$X = \begin{bmatrix} X_{00} & \cdots & X_{0,n_2-1} \\ \vdots & & \vdots \\ X_{s-1,0} & \cdots & X_{s-1,n_2-1} \end{bmatrix}, \quad Z = \begin{bmatrix} Z_{00} & \cdots & Z_{0,n_1-1} \\ \vdots & & \vdots \\ Z_{r-1,0} & \cdots & Z_{r-1,n_1-1} \end{bmatrix}$$

- Where, $X_{kj} \in \mathbb{C}^{l_c \times 1}$ and $Z_{kj} \in \mathbb{C}^{l_c \times 1}$
- Assume that $n_1 = s l_c$ and $n_2 = r l_c$

MATRIX TRANSPOSITION

- Scalar-Level Transposition

- Given $X \in \mathbb{C}^{n_1 \times n_2}$, the following algorithm computes $Z = X^T$

```
for col=0:n2-1
    for row=0:n1-1
        Z(col,row) <- X(row,col)
    end
end
```

- If CPU wishes to store a component of Z_{kj} , then with write back protocol
 - Z_{kj} is brought into cache it is not already there,
 - The cache line is associated with Z_{kj} is updated, and
 - Sub column Z_{kj} in main memory is updated only when the cache line associated with Z_{kj} is bumped

- If n_1 and n_2 is much larger than l_c and n_c , then

- X is accessed by column
- Mapped into n_1 different Z-blocks
 - If $n_1 > n_2$,

$$z_{00} \leftarrow x_{00}$$

$$z_{01} \leftarrow x_{10}$$

$$\vdots$$

$$z_{0,n_1-1} \leftarrow x_{n_1-1,0}$$

$$z_{10} \leftarrow x_{01}$$

- Each Z-block is read into cache l_c times during the execution

Transposition by block

- Assume that $n_1 = \alpha_1 N_1$ and $n_2 = \alpha_2 N_2$

$$X = \begin{bmatrix} X_{00} & \cdots & X_{0,N_2-1} \\ \vdots & & \vdots \\ X_{N_1-1,0} & \cdots & X_{N_1-1,N_2-1} \end{bmatrix}, \quad X_{kj} \in \mathbb{C}^{\alpha_1 \times \alpha_2},$$

$$Z = \begin{bmatrix} Z_{00} & \cdots & Z_{0,N_1-1} \\ \vdots & & \vdots \\ Z_{N_2-1,0} & \cdots & Z_{N_2-1,N_1-1} \end{bmatrix}, \quad Z_{kj} \in \mathbb{C}^{\alpha_2 \times \alpha_1}.$$

Algorithm

```
for  $k = 0:N_1 - 1$   
  rows  $\leftarrow \alpha_1 k : \alpha_1(k + 1) - 1$   
  for  $j = 0:N_2 - 1$   
    cols  $\leftarrow \alpha_2 j : \alpha_2(j + 1) - 1$   
     $Z(\text{cols}, \text{rows}) \leftarrow X(\text{rows}, \text{cols})^T$   
  end  
end
```

- Suppose α_1 and α_2 are multiples of ℓ_c and that $2\alpha_1\alpha_2 \leq n_c\ell_c$.
- Implies that X and Z block can simultaneously fit into cache

External Memory Matrix Transpose

- The Large Buffer case for Square Matrices
- The Large Buffer case for Rectangular Matrices

The Large Buffer case for Square Matrices

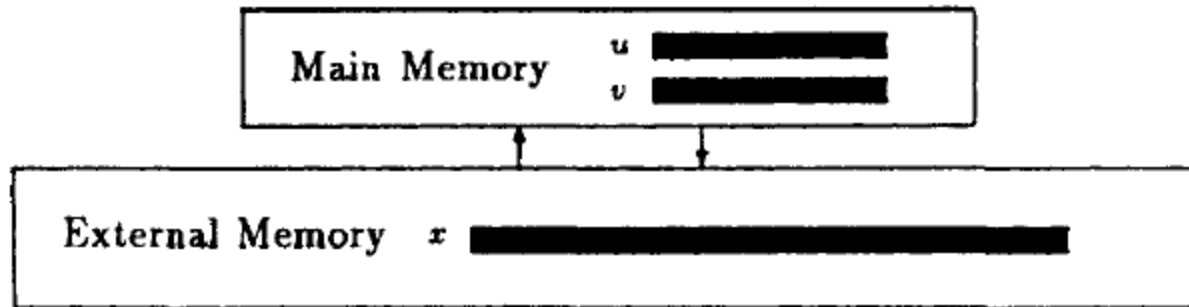


FIG. 3.2.3. *In-place external memory transposition.*

$$X = \begin{bmatrix} X_{00} & \cdots & X_{0,N-1} \\ \vdots & & \vdots \\ X_{N-1,0} & \cdots & X_{N-1,N-1} \end{bmatrix}, \quad X_{kj} \in \mathbb{C}^{b \times b}.$$

- $n_1=n_2=Nb$ and $b^2 \leq B$

- For $0 \leq k, j \leq N$
- Define $X_{kj} = X(kb:(k+1)b-1, jb:(j+1)b-1)$.

```

for  $k = 0:N-1$ 
     $u_{b \times b} \leftarrow X_{kk}$ ;  $u_{b \times b} \leftarrow u_{b \times b}^T$ ;  $X_{kk} \leftarrow u_{b \times b}$ 
    for  $j = k+1:N-1$ 
         $u_{b \times b} \leftarrow X_{kj}$ ;  $v_{b \times b} \leftarrow X_{jk}$ 
         $u_{b \times b} \leftarrow u_{b \times b}^T$ ;  $v_{b \times b} \leftarrow v_{b \times b}^T$ 
         $X_{kj} \leftarrow v_{b \times b}$ ;  $X_{jk} \leftarrow u_{b \times b}$ 
    end
end

```

- The read operation $u_{b \times b} \leftarrow X_{kj}$ is carried as follows

```

for  $\tau = 0:b-1$ 
     $u(\tau b:(\tau+1)b-1) \leftarrow x((jb+\tau)n_1 + kb:(jb+\tau)n_1 + (k+1)b-1)$ 
end

```

The Large Buffer case for Rectangular Matrices

- Assume $n_1 = pn_2$ and $n_2 = Nb$

$$x_{n_1 \times n_2} = X = \begin{bmatrix} X_0 \\ \vdots \\ X_{p-1} \end{bmatrix}, \quad X_j \in \mathbb{C}^{n_2 \times n_2},$$

- Thus permutation has to be used so that we can get

$$x_{n_2 \times n_1} = X^T = [X_0^T \mid \cdots \mid X_{p-1}^T]$$

- For example, $n_2 = 4$, $p = 3$. and $X_0=A, X_1=B, X_2=C$, then

$$x_{4 \times 12} = \left[\begin{array}{ccc|ccc|ccc|ccc} a_{00} & b_{00} & c_{00} & a_{01} & b_{01} & c_{01} & a_{02} & b_{02} & c_{02} & a_{03} & b_{03} & c_{03} \\ a_{10} & b_{10} & c_{10} & a_{11} & b_{11} & c_{11} & a_{12} & b_{12} & c_{12} & a_{13} & b_{13} & c_{13} \\ a_{20} & b_{20} & c_{20} & a_{21} & b_{21} & c_{21} & a_{22} & b_{22} & c_{22} & a_{23} & b_{23} & c_{23} \\ a_{30} & b_{30} & c_{30} & a_{31} & b_{31} & c_{31} & a_{32} & b_{32} & c_{32} & a_{33} & b_{33} & c_{33} \end{array} \right]$$

- In general, we get $x_{n_2 \times n_1} \leftarrow x_{n_2 \times n_1} \Pi_{p, n_1}$

Quantifying Locality In The Memory Access Patterns of HPC Applications

- Spatial Locality
- Temporal Locality
- Measuring Locality
- Results

Spatial Locality

- Spatial locality is the tendency of applications to access memory address near other recently accessed addresses
- Stride of a memory access is defined as the minimum absolute distance, in 64-bit words, of that memory reference to its nearest neighbor among the W previously accessed addresses
- Single number spatiality score

$$\sum_{i=1}^{\infty} stride_i / i$$

- Stride_i denotes the fraction of total dynamic memory operations that are of stride length i
- The idea is to simply generate a normalized score in the range $[0,1]$ that is inversely proportional to the average stride length.

Temporal Locality

- Temporal locality is the tendency of an application to reference the same memory addresses that it referenced recently
- Reuse distance of some reference of memory address A is defined as the number of unique memory addresses that have been accessed since the last access to A
- The reuse function plots reuse distances against the percentage of an application's dynamic memory operations with reuse distance less than or equal to that distance

- The metric used is the cumulative fraction of memory operations at each reuse distance is scaled by some increasing function of that reuse distance and summed
- Idea is that since the reuse function is monotonically increasing, each memory access increases the application's total score in a manner inversely proportional to its reuse distance

$$\frac{\sum_{i=0}^{\log(N)-1} ((reuse_{2(i+1)} - reuse_{2i}) * \log_2(N) - i)}{\log_2(N)}$$

- $Reuse_i$ denotes the fraction of dynamic memory operations with reuse distance less than or equal to i

Measuring Locality

- Memory access characteristics are gathered using Metasim tracer, a tool for dynamic analysis of a program's memory references.
- It collects statistics about memory strides and simulates cache behavior as the program runs

Results

- Testing and analysis is done on relevant serial benchmarks from the HPCC and Apex-MAP
- **STREAM** - Measures sustainable memory bandwidth and the corresponding computation rate for a vector kernel
- **Random Access (GUPS)** - Measures the rate of integer random updates of memory
- **FFT** - Measures floating point rate of execution of doubly precision complex one-dimensional Discrete Fourier Transform
- **HPL** – Linpack TPP benchmark that measures the floating point rate for solving a linear system of equations

- Locality scores of the four applications

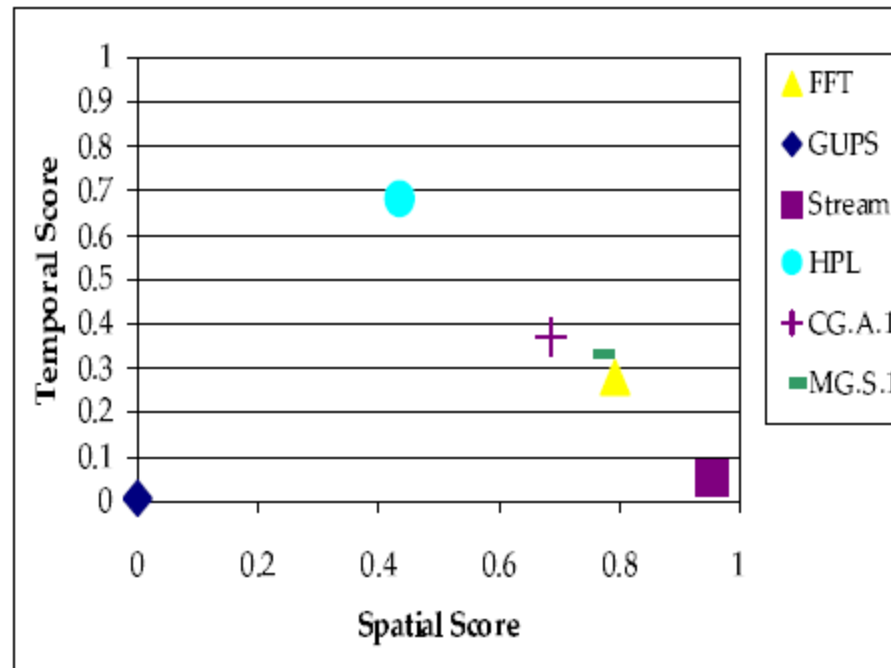


Figure 1: Locality scores of the HPCC and NPB benchmarks

- Temporal reuse functions

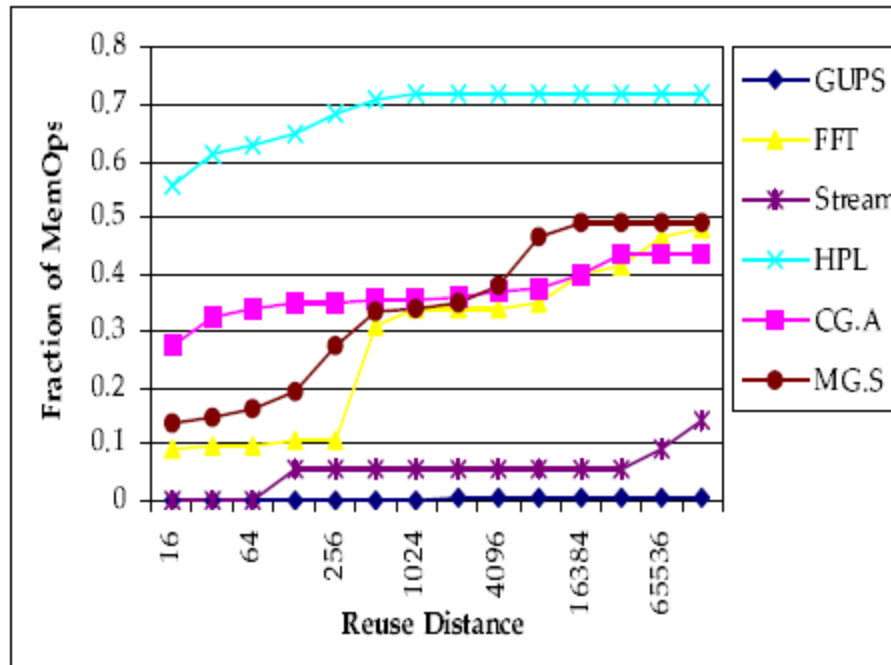


Figure 2: Temporal locality functions of HPCC and NPB benchmarks

References

- The paper ["A Fast Fourier Transform Compiler,"](#) by Matteo Frigo, appears in the Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation ,vAtlanta, Georgia, May 1999.
- ["FFTW: An Adaptive Software Architecture for the FFT"](#), by M. Frigo and S. G. Johnson.
- Fast Transforms Algorithms, Analyzes, Applications by Douglas F. Elliott and K. Ramamohan Rao.
- Van Loan C.F. Computational Frameworks for the Fast Fourier Transform (SIAM, 1992)(0898712858).
- The paper [Quantifying Locality In The Memory Access Patterns of HPC Applications](#) by Jonathan Weinberg, Michael O. McCracken, Allan Snaveley and Erich Strohmaier