FFT: FAST FOURIER TRANSFORM ALGORITHMS

Swetha Gurram and Mohammed Yousuf

Abstract

The Fast Fourier Transform is a quick method of performing the Discrete Fourier Transform (DFT) that maximizes the performance by reducing the number of arithmetic operations. The FFT method is fast as it increases the speed by reducing the complexity. In this paper, we propose the different classes of FFT and the complexities involved in computing each one of them. This paper also deals with FFTW, an implementation of the Discrete Fourier transform that adapts to the hardware in order to maximize performance and also the matrix transposition in a memory hierarchy. We conclude by noting few points from a paper based on quantifying locality in the memory access patterns of hpc applications.

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 :

$$\mathrm{X(k)} = 1\,/\,\mathrm{N}\sum_{n=0}^{N-1}\mathrm{x(n)}e^{-\frac{j2\pi}{N}\mathrm{kn}} = 1\,/\,\mathrm{N}\sum_{n=0}^{N-1}\mathrm{x(n)W^{kn}}$$

for k= 0, 1, 2,……, N-1

Algorithm for DFT is given by :

```
for $p = 0{:}n-1$
    $\omega \leftarrow \exp(-2\pi ip/n)$
    $F(p,0) \leftarrow 1$
    for $q = 1{:}n-1$
        $F(p,q) \leftarrow \omega F(p,q-1)$
    end
end
```

The DFT 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.

Ordinarily, a DFT would involve $8n^2$ flops, since there are n length-n inner products to perform where as the FFT involves 5*n 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.

Different Fast Fourier Transform

The different Fast Fourier transforms include:

- 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*).

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk}$$

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

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

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

is just F_4 with its even-indexed columns grouped first. The key, however is to regard this permutation of F_4 as a 2 by 2 block matrix.

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

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

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

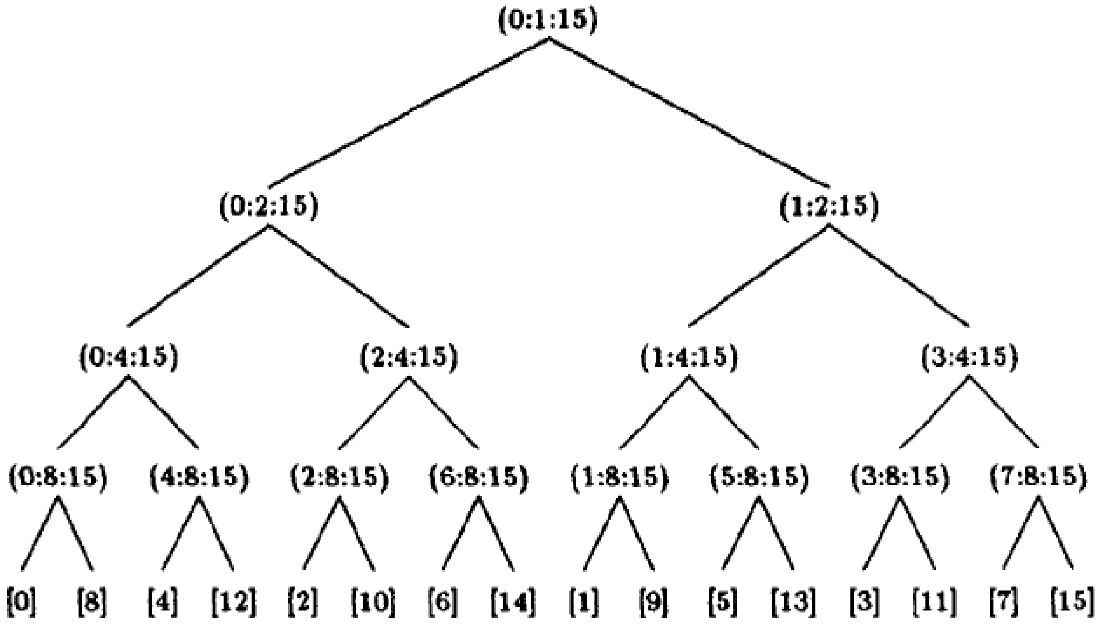
Thus, each block shown above is either F_2 or a diagonal scaling of F_2.

In general,

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

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

Suppose n = 16. Then the computation tree is shown below:

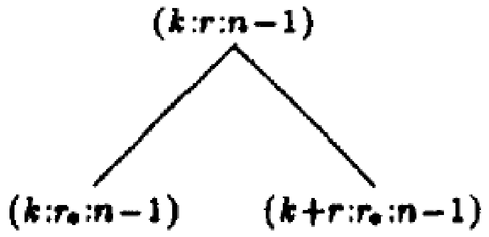


The structure of a radix-2 FFT ($n = 16$)

The Computation Tree in General

Suppose n = 2^t and that we number the levels of the computation tree from the bottom to the top, identifying the leaf level with level 0. Each level-q DFT has the form F_L x(k:r:n— 1), where L = 2^q and r = n/L. It is synthesized from a pair of level q - l DFTs, each of which has the form F_{L*} x(k:r*:n — 1), where L* = L / 2 and r* = 2r.

We depict this operation as follows:



A recursive Radix – 2 FFT procedure is given below:

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

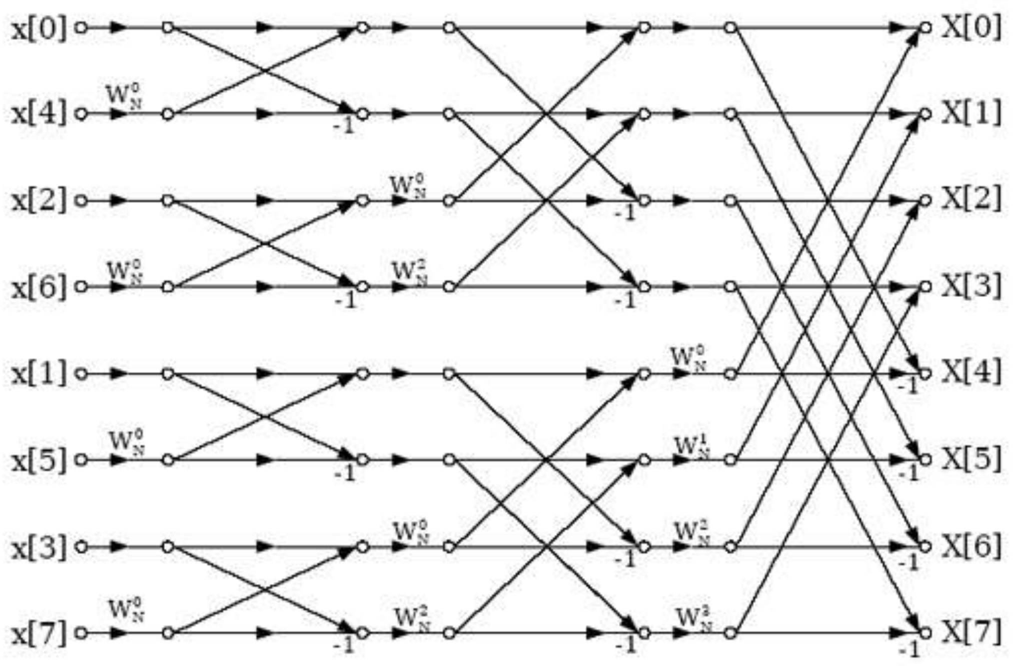
Here y = F_n x and n is a power of two.

Decimation in Time and Decimation in Frequency Radix – 2 FFT algorithms

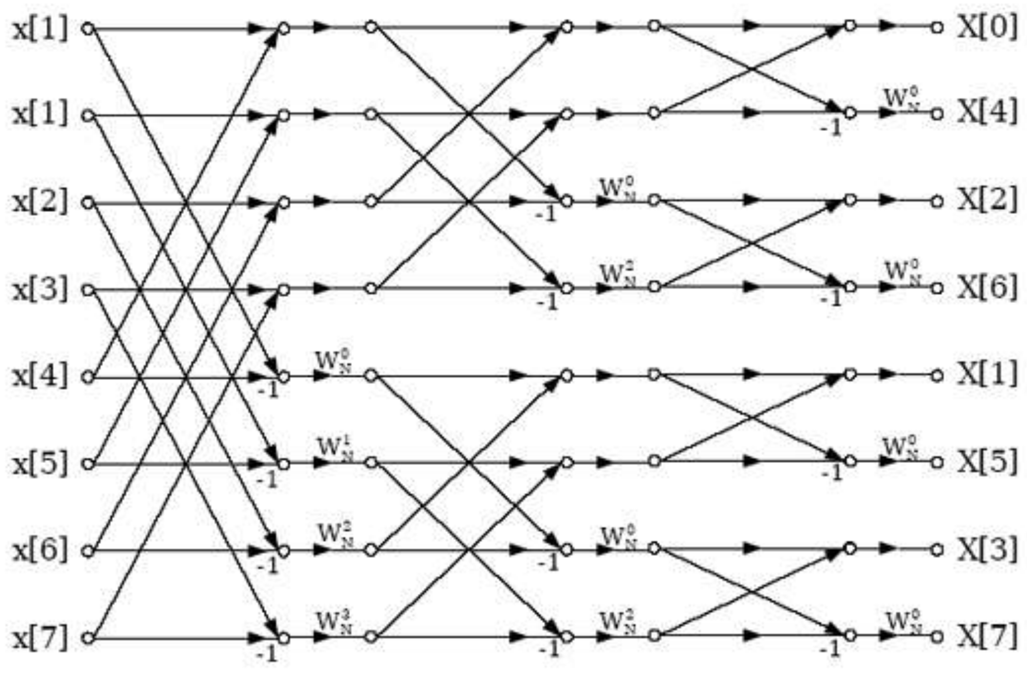
The radix-2 decimation-in-time algorithm rearranges the discrete Fourier transform (DFT) equation into two parts: a sum over the even-numbered discrete-time indices n=[0,2,4,…,N−2] and a sum over the odd-numbered indices n=[1,3,5,…,N−1]. This is called decimation in time because the time samples are rearranged in alternating groups.

For N = 8, the bit reversal is given as :

In-order index	In-order index in binary	Bit-reversed binary	Bit-reversed index
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7



The radix-2 decimation-in-frequency algorithm rearranges the discrete Fourier transform (DFT) equation into two parts: computation of the even-numbered discrete-frequency indices X(k) for k=[0,2,4,…,N−2] (or X(2r) as in Equation 2) and computation of the odd-numbered indices k=[1,3,5,…,N−1]. This is called decimation in frequency because the frequency samples are computed separately in alternating groups.

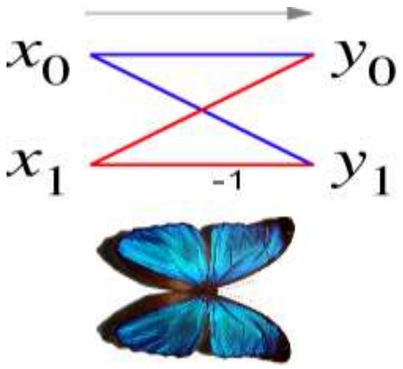


Butterfly Operators

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 :

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

$$\Omega_{L_*} = \mathrm{diag}(1, \omega_L, \ldots, \omega_L^{L_*-1})$$ where $L_* = 2L$

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

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

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

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

In general,

$$F_n x = A_t \cdots A_1 P_n^T x$$

where *Pn* is some permutation and *Aq* is a direct sum of the butterfly operators,

$$A_q = \text{diag}(\underbrace{B_L, \ldots, B_L}_{r}) = I_r \otimes B_L \qquad L = 2^q,\ r = n/L$$

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
```

```
$x \leftarrow P_n x$
$w \leftarrow w_n^{(long)}$
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
```

Radix – p splitting FFT

Suppose n = pm with 1 < p < n. The primary purpose of this section is to express Fn as a function of the two smaller DFT matrices Fp and Fm . The key result is the radix-p splitting:

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

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

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 have the form:

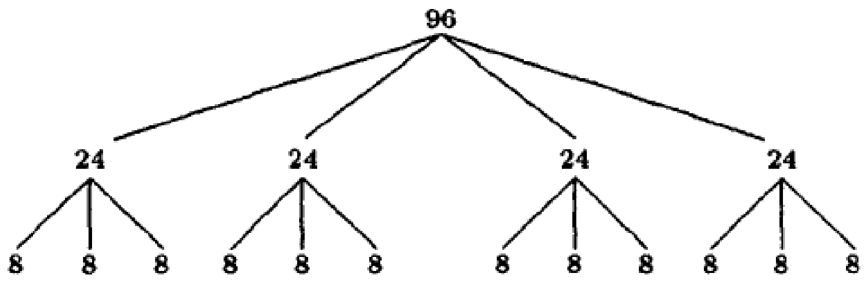
$$F_{96}x \longleftarrow \begin{cases} F_{24}x(0{:}4{:}95) \\ F_{24}x(1{:}4{:}95) \\ F_{24}x(2{:}4{:}95) \\ F_{24}x(3{:}4{:}95) \end{cases}$$

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

$$F_{96}x \longleftarrow \begin{cases} F_{24}x(0{:}4{:}95) & \longleftarrow \begin{cases} F_{12}x(0{:}8{:}95) \\ F_{12}x(4{:}8{:}95) \end{cases} \\ F_{24}x(1{:}4{:}95) & \longleftarrow \begin{cases} F_{8}x(1{:}12{:}95) \\ F_{8}x(5{:}12{:}95) \\ F_{8}x(9{:}12{:}95) \end{cases} \\ F_{24}x(2{:}4{:}95) & \longleftarrow \begin{cases} F_{6}x(2{:}16{:}95) \\ F_{6}x(6{:}16{:}95) \\ F_{6}x(10{:}16{:}95) \\ F_{6}x(14{:}16{:}95) \end{cases} \\ F_{24}x(3{:}4{:}95) & \longleftarrow \begin{cases} F_{4}x(3{:}24{:}95) \\ F_{4}x(7{:}24{:}95) \\ F_{4}x(11{:}24{:}95) \\ F_{4}x(15{:}24{:}95) \\ F_{4}x(19{:}24{:}95) \\ F_{4}x(23{:}24{:}95) \end{cases} \end{cases}$$

Mixed-radix FFT

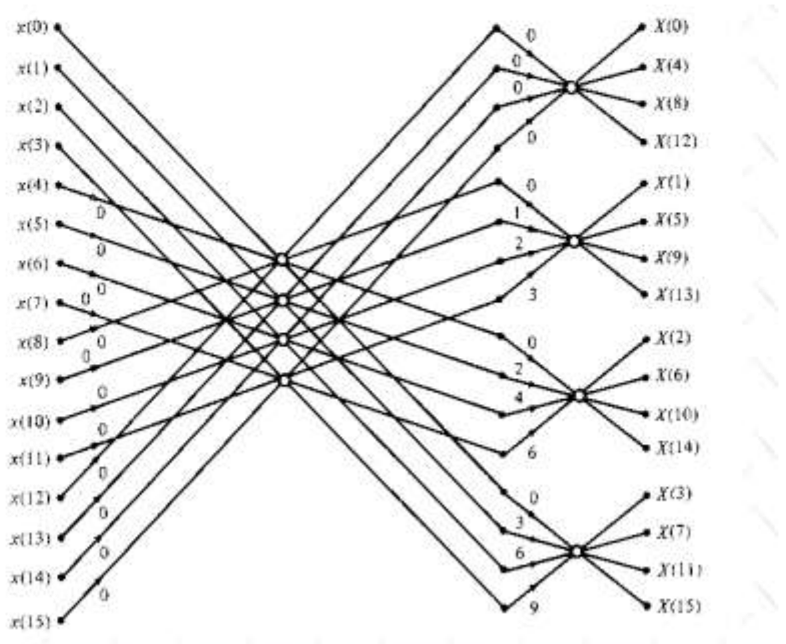
If the same splitting rules are used at each level, then a mixed-radix framework results. That is, if n is factored the same way across any level of the associated computation tree, then a mixed-radix FFT results.



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

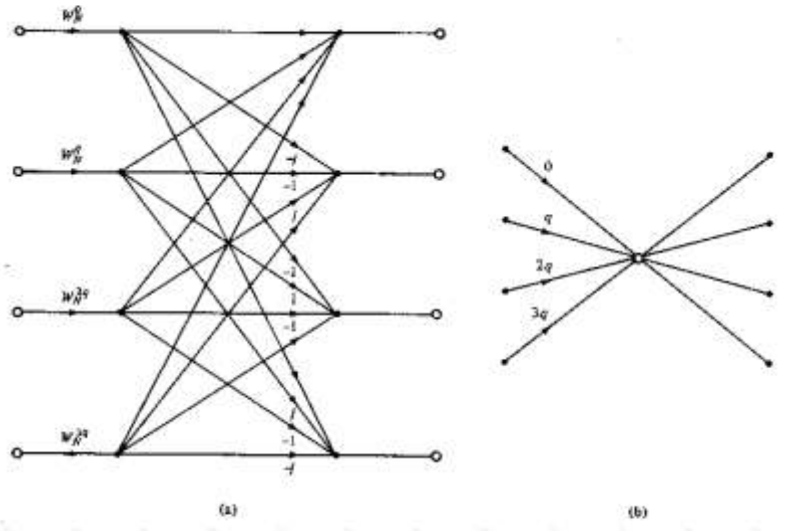
Radix - 4 Fast Fourier Transform

The radix – 4 approach assumes that $n = 4^t$. Properly arranged, these algorithms involve fewer flops and better memory traffic patterns than their radix-2 counterparts.



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 additions.



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

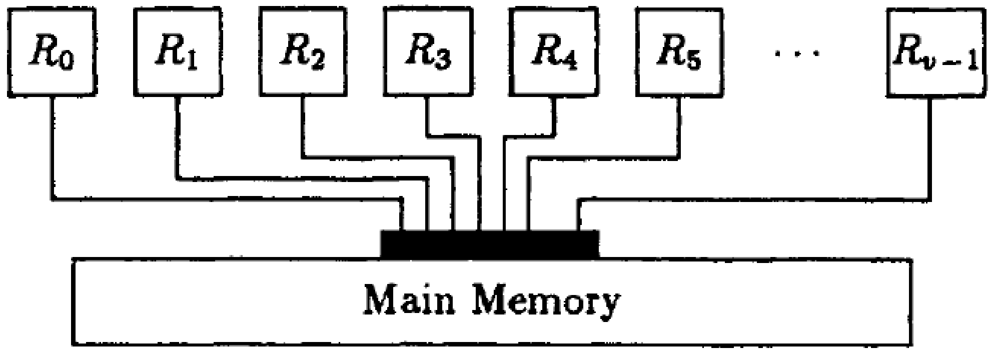
The exploitation of structure led to the reduced arithmetic requirements of the radix-4 butterfly. Let us repeat the exercise by developing an efficient implementation of the radix-8 butterfly. An 8-point DFT x = F_8 z has the form:

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

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 Iog n, 4.25n Iog n,* and *4.08n Iog n* flops, respectively, and so from that perspective, the radix-4 and 8 frameworks have added appeal. However, another advantage of these higher radix frameworks concerns the re-use of data.



Registers and memory

Assume that the *Ri* are vector registers capable of *real* vector operations such as vector addition, vector multiply, vector load, and vector store. The act of performing an operation of the form $z \leftarrow B_{p,L*}\, z$, where $L = pL_*$ involves (a) loading z and the weights into the registers, (b) performing the actual vector adds and multiplies, and (c) writing the updated z to memory. In a paradigm where loads and stores are costly overheads, it pays to engage in a lot of arithmetic once data is loaded into the registers.

From the table below, we see that the amount of arithmetic associated with length-L butterfly increases with radix.

Flops per butterfly.

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

Looked at another way, if n = 8^t, then 3t, 2t, and 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 make them attractive in many environments.

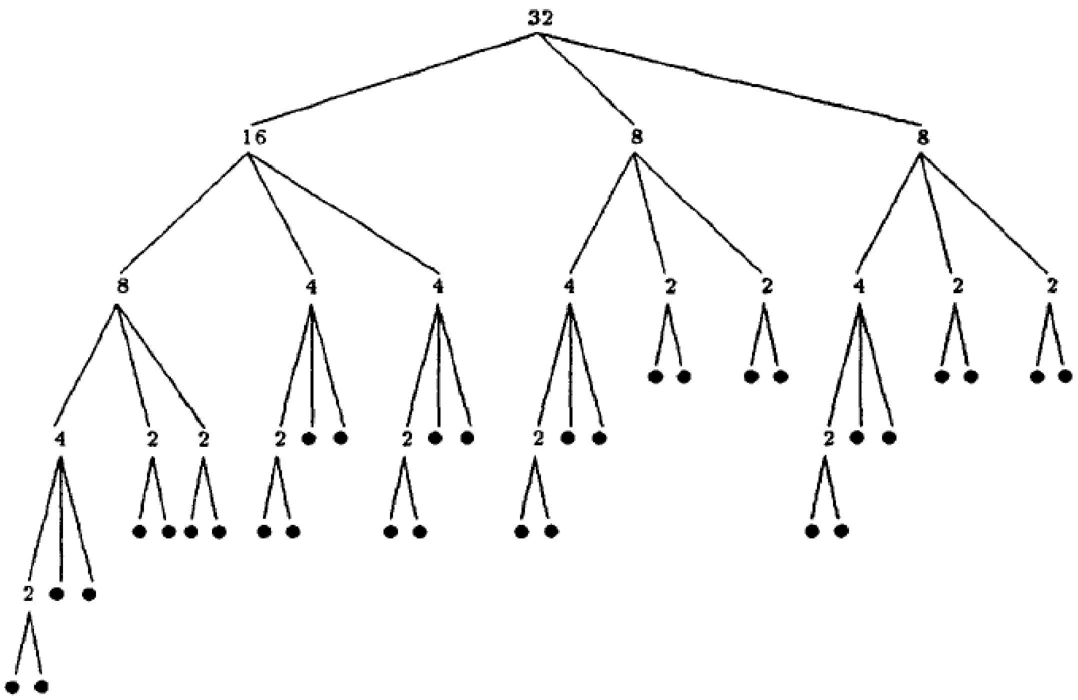
Split-radix Fast Fourier Transform

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.

$$\begin{tikzcd}
 & F_n x(0:n-1) \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
F_m x(0:2:n-1) & F_p x(1:4:n-1) & F_p x(3:4:n-1)
\end{tikzcd}$$

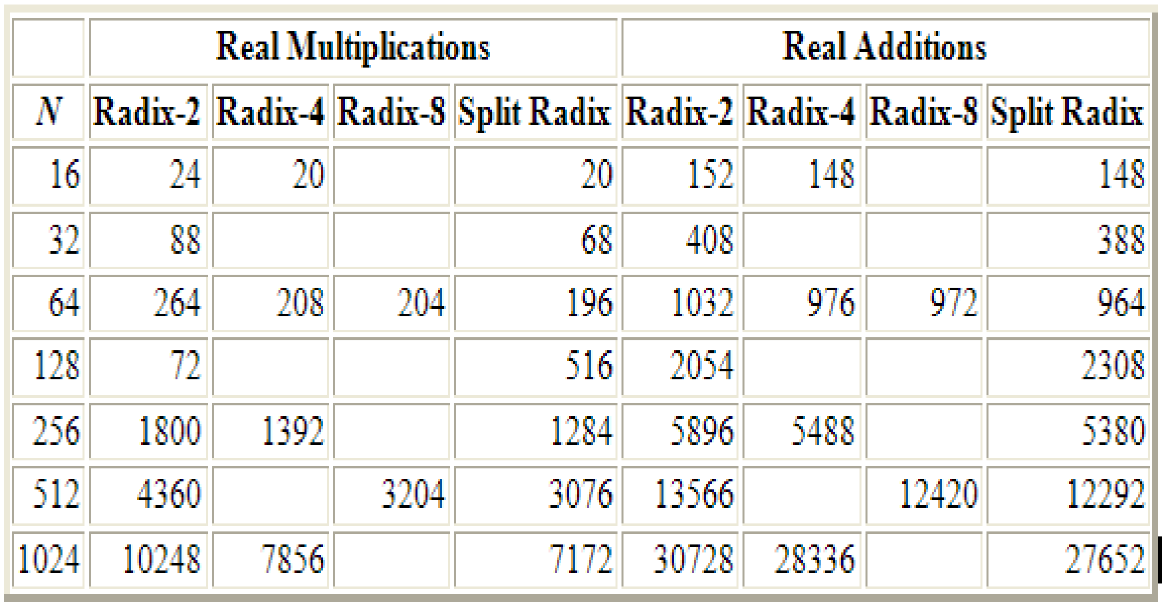
$$m = n/2$$
$$p = n/4.$$

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 4*n log n* flops.

Comparison between different FFTs



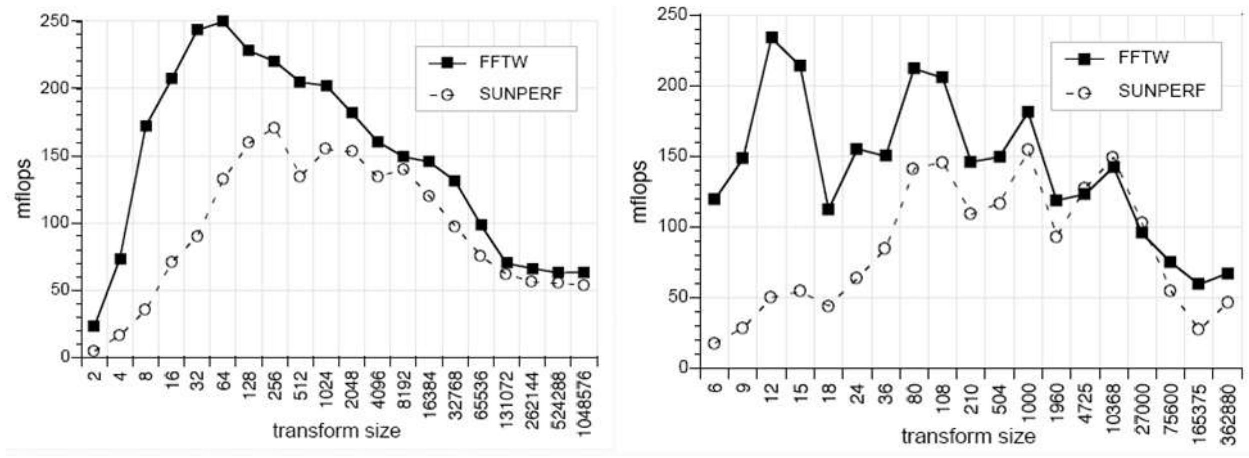
	Real Multiplications				Real Additions			
N	Radix-2	Radix-4	Radix-8	Split Radix	Radix-2	Radix-4	Radix-8	Split Radix
16	24	20		20	152	148		148
32	88			68	408			388
64	264	208	204	196	1032	976	972	964
128	72			516	2054			2308
256	1800	1392		1284	5896	5488		5380
512	4360		3204	3076	13566		12420	12292
1024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

FFTW – Fast Fourier Transform In the West

FFTW is a C-subroutine library for computing the DFT and is 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 also 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.

genfft operates in four phases :

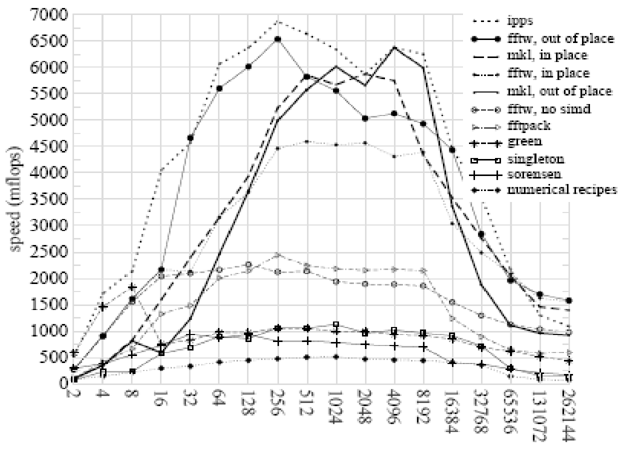
- **Creation phase**: In the creation phase, genfft produces a directed acyclic graph (dag) of the codelet, according to some well-known algorithm for the DFT. The generator contains many such algorithms and it applies the most appropriate.
- **Simplifier phase:** In the simplifier,genfft applies local rewriting rules to each node of the dag, in order to simplify it. In traditional compiler terminology, this phase performs

algebraic transformations and common-subexpression elimination, but it also performs other transformations that are specific to the DFT.

- **Scheduler phase:** In the scheduler, genfft produces a topological sort of the dag (a "schedule") that, for transforms of size 2^k, provably minimizes the asymptotic number of register spills.
- Finally, the schedule is unparsed to C. (It would be easy to produce FORTRAN or other languages by changing the unparser.)

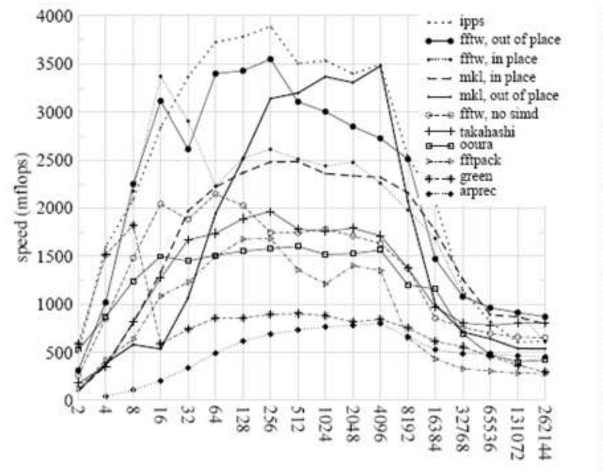
Features of FFTW

- FFTW is written in portable C and runs well on many architectures and operating systems.
- FFTW computes DFT in O(nlogn) 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.

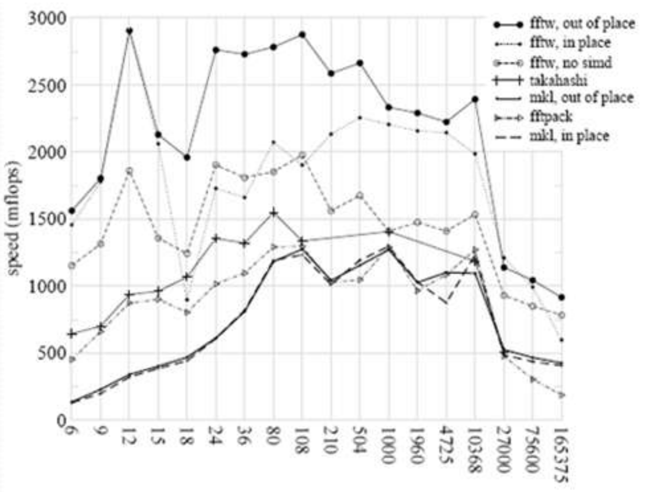


Comparison of single-precision 1d complex DFTs, power-of-two sizes, on a 2.8 GHz Pentium IV.

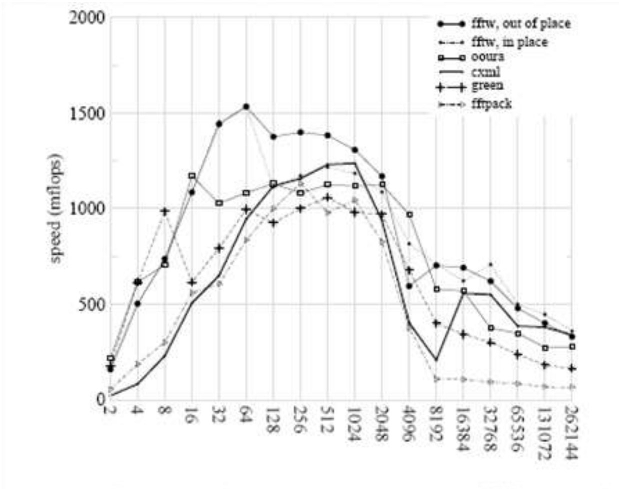
In addition to FFTW v. 3.0.1, the other codes benchmarked are as follows (some for only one precision or machine): arprec, "four-step" FFT implementation (from the C++ ARPREC library, 2002); cxml, the vendor-tuned Compaq Extended Math Library on Alpha; fftpack, the Fortran library ; green, free code 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; ooura, 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.



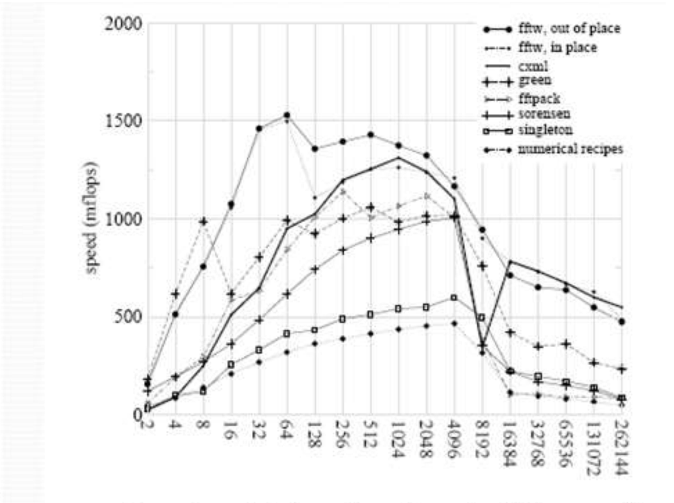
Comparison of double-precision 1d complex DFTs, power-of-two sizes, on a 2.8 GHz Pentium IV. Intel C/Fortran compilers v. 7.1, optimization flags `-O3 -xW` (maximum optimization, enable automatic vectorizer).



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. For example, for a complex transform of size n = 13, the generator employs an algorithm due to Rader. In its most sophisticated variant, this algorithm performs 214 real (floating-point) additions and 76 real multiplications. The generated code in FFTW for the same algorithm, however, contains only 176 real additions and 68 real multiplications, because genfft found certain simplifications that the authors of did not notice.

FMA Optimized FFTs

- Modern processors provide multiply-add or fused multiply-add (FMA) instructions that perform the ternary operation ±a±b×c in the same amount of time needed for a single floating-point addition or multiplication operation. FMA instructions have a significant effect on performance. With a decoding rate of one instruction per clock cycle, the peak throughput is two floating-point operations per cycle for FMA instructions. For individual add or multiply instructions, it is only one floating-point operation per clock cycle.

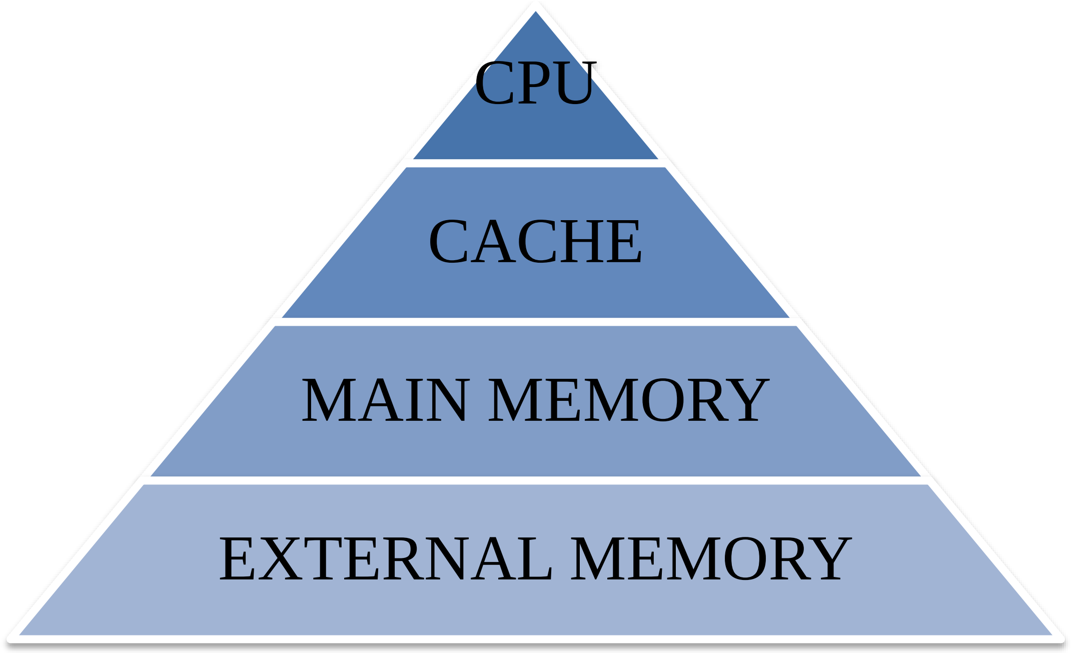
- Conventional Cooley-Tukey FFT algorithms require more real additions than real multiplications. Thus, in implementations of such algorithms it is not possible to schedule all multiplications so that they appear as genuine multiply-add operations. Implementing FMA Optimized FFTs helped achieve two main 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.

- In particular, a split-radix kernel and prime kernels (radix 2, 3, and 5) were optimized with respect to FMA utilization. FMA utilization between 87.5% and 100% was achieved.

Matrix Transposition in A Memory Hierarchy



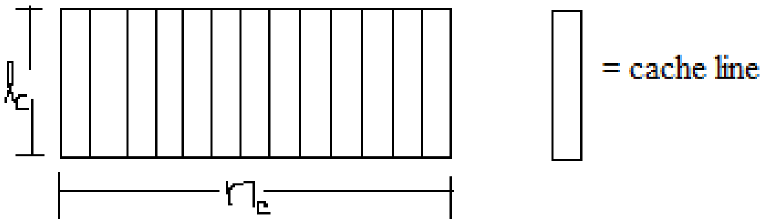
Cache is a small but fast memory whose purpose is to hold currently active data; thereby reducing the number of reads and writes to main memory, hence reducing the time to access the data from the main memory. Main memory in turn is smaller than external memory and CPU access time is less than External memory. Data transfers between memory levels are expensive and care should be taken so as to minimize their frequency.

There are two sub-problems regarding matrix transposition in a memory hierarchy.

1. The Matrix resides in main memory and the data transfer frequency between the cache and main memory should be reduced.
2. The Matrix resides in external memory and the data transfer frequency between the main memory and cache should be reduced.

Transposition with a cache

Cache is considered as an array with each column as a cache line of length l_c. The cache has in total n_c lines.



During the transposition Z <- X^T, array elements flow between the cache and main memory according to various read and write protocols. The matrices are represented using blocks.

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

Where, X_{kj} εC^{lcx1} and $Z_{kj} \varepsilon C^{lcx1}$ and we assume that $n_1=s_{lc}$ and $n_2=r_{\ lc}$

If we consider the basic **Scalar level Transposition algorithm**, given by

Given $X \varepsilon C^{n1xn2}$, 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 the CPU requests a component of X_{kj} and if this sub-column is in cache, a cache hit occurs and CPU retrieves it. But, if there is a cache miss, then it is copied from the main memory into a cache line, “bumping” the previous contents of the line. The following procedure is followed in the whole procedure.

- Z_{kj} is brought into cache if it is not already present
- Cache line associated with Z_{kj} is updated, and
- Sub-column in main memory is updated only when the cache line associated with Z_{kj} is bumped.

We will now consider the execution of above algorithm with the assumption that n_1 and n_2 are both much greater than n_c and l_c. Both X and Z resides in main memory. With the col-row loop ordering, matrix X is accessed by column and hence, it takes advantage of spatial locality. But, the matrix Z is accessed through different n_1 Z blocks. For instance, the assignment z_{00} <- x_{00} prompts the loading of $Z_{00.}$ Unfortunately, this block is bumped off by the time this assignment z_{10} <- x_{01} is done. There are n_1-1 different assignments between them.

z_{01} <- x_{10}

.

.

.

z_{0n1-1} <- $x_{n1-1,0}$

Hence, the each Z block is loaded l_c times during the execution of the algorithm. Inversion of the loops causes the X matrix to load l_c times. Hence, the problem still persists.

Transposition by block:

We will consider the transposition by block to solve the above problem. Assume that $n_1=\alpha_1 N_1$ and $n_2=\alpha_2 N_2$ and partition X and Z arrays as follows.

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

This algorithm transposes block by block fashion.

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

Suppose α_1 and α_2 are multiples of l_c and that $2\alpha_1\alpha_2 \le n_c l_c$. This implies that an X-block and Z-block can simultaneously fit into cache. With this arrangement, each entry of X and Z are read exactly once and each component of Z is written to main memory exactly once.

External Memory Matrix Transpose:

The main constraints are

- The matrix X is so large that it cannot be entirely housed in main memory.
- A pair of main memory work spaces is available denoted by *u* and *v*, each of length B.
- Array entries move between memory levels in length *b* blocks.

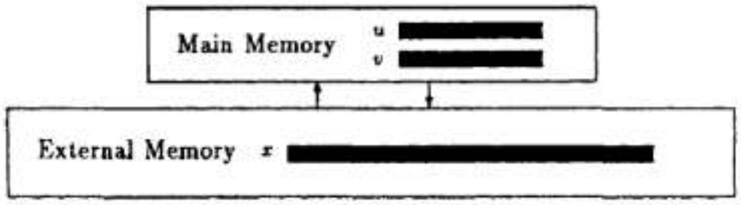


FIG. 3.2.3. *In-place external memory transposition.*

The Large Buffer case for Square Matrices:

Assume $n_1=n_2=Nb$ and $b^2 \le B$ so that *u* and *v* can each house a block from the partitions

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

Algorithm: For $0 \le k, j \le N$, X_{kj} is defined as $X_{kj} = X(kb{:}(k+1)b-1, jb{:}(j+1)b-1)$, If the main memory buffers has the length b^2 then this algorithm overwrites X with X^T .

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

The two main observations are:

- The assignments to *u* and *v* require a read from external memory, whereas copying *u* and *v* into an X block involves a write to external memory.
- Reading from, or writing to main memory requires a loop of length *b*.

For instance, the external memory read operation $u_{b\times b} \leftarrow X_{kj}$ is carried out 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
```

This algorithm is an example of a *single pass* transpose algorithm because each entry in external array *x* is read exactly once.

The Large Buffer case for Rectangular Matrices:

Assume $n_1=pn_2$ and $n_2=Nb$. The sub-matrices in matrix X partitioning are not contiguous in *x*.

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

Thus, we need to permute so that we can get the following result.

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

For Instance, consider n_2=4 & p=3 and denoting X_0=A, X_1=B and X_2=C. The matrix is given by

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

To make blocks A, B and C contiguous in *x*, we apply mod 3 permutation. Hence, in general we compute

$$x_{n_2\times n_1} \leftarrow x_{n_2\times n_1}\Pi_{p,n_1}$$

We then apply the previous algorithm for square blocks and then obtain the transpose of the matrix.

Quantifying Locality in The Memory Access Patterns Of HPC Applications

We only concerned the following topics in this particular paper written by Jonathan Weinberg, Michael O. McCracken, Allan Snavely and Erich Strohmaier.

- 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. They defined a *stride* as a metric to measure the spatial locality access patterns of HPC applications.

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. Now, they defined a single number spatiality score as

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

Where, 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. For Instance, an application made a memory access with stride 1 then it gets a score of 1, but if it makes a memory access with stride 2 then the score for that application is 0.5.

Temporal Locality:

Temporal locality is the tendency of an application to reference the same memory addresses that it referenced recently. They defined *Reuse Distance* as the metric for temporal locality.

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.

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

Where, 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. This *Metasim tracer* is used to measure both the strides and reuse distance from which the score for spatial locality and temporal locality are calculated.

Results:

Testing and analysis is done on relevant serial benchmarks from the HPCC and Apex-MAP. The following are a few benchmarks from the suite.

- *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.

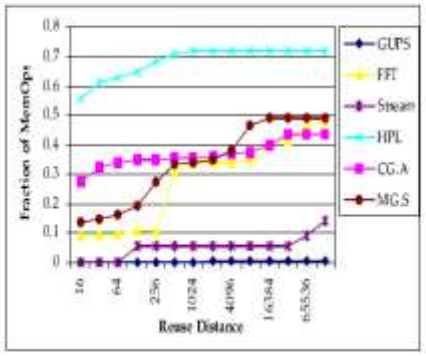


Figure 2: Temporal locality functions of HPCC and NPB benchmarks

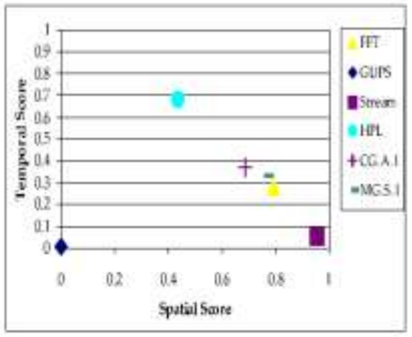


Figure 1: Locality scores of the HPCC and NPB benchmarks

GUPS performs random updates to a large memory and therefore displays neither spatial nor temporal locality. STREAM on the other hand, performs only regularly strided memory access to a large memory and therefore displays a great degree of spatial locality but very little temporal. HPL exhibits especially high levels of temporal locality while FFT does similarly with spatial locality. The end result is that these metrics allow us to compare benchmarks via two single-number scores and make comparisons such as STREAM has more spatial locality than CG" or "FFT has lower temporal locality than HPL" in a straightforward and meaningful way.

Conclusion:

The first half of the paper discussed the different types Fast Fourier Transform algorithms and their complexities, the FFTW and the use of FMA's for FFT. In the second half of the paper, Temporal and spatial locality issues have been discussed in a memory hierarchy. A paper regarding the measurement of temporal and spatial localities has been discussed.

References

- The paper "*A Fast Fourier Transform Compiler,*" by Matteo Frigo, Georgia, May 1999.
- "*FFTW: An Adaptive Software Architecture for the FFT*", by M. Frigo and S. G. Johnson.
- *Fast Transforms Algorithms, Analyzes, Applications* by Doughlas F. Elliott and K. Ramamohan Rao.
- *Van Loan C.F. Computational Frameworks for the Fast Fourier Transform* (SIAM,1992)
- The paper Quantifying *Locality In The Memory Access Patterns of HPC Applications* by Jonathan Weinberg, Michael O. McCracken, Allan Snavely and Erich Strohmaier