Recent Developments in High Performance FFT Algorithms and Architectures: A Learning Module for Undergraduate Algorithms and Architecture Courses

Gauri Sabane

Department of Computer Science

St. Cloud State University
720 Fourth Avenue South
St. Cloud, Minnesota 56301-4498

Abstract

The Discrete Fourier Transform (DFT) has a large number of signal processing applications such as computational medicine, communications, instrumentation, biomedical engineering, sonics and acoustics, numerical methods, applied mechanics and engineering. DFTs are not computed directly, but instead depend on Fast Fourier Transform (FFT). FFT algorithms efficiently compute the Fourier Transform by speeding up the number of computations as compared to DFT. Many of these algorithms generally adopt parallel approaches of FFT computation to improve performance.

Introducing FFT algorithms to Computer Science and Engineering students is not an easy task. Therefore, we attempted to develop an active learning module to help students understand recent developments in FFT algorithms and architectures. This paper will focus on both sequential and parallel FFT algorithms and recent advances in hardware architectures.

1. Introduction

The *Fourier Transform* is one of the most important algorithms widely used in scientific and engineering applications. It is often being used in digital applications such as voice recognition, image processing, ECG and EEG signal processing. The *Discrete Fourier Transform*, DFT, provides a means to transform a time domain signal into an inverse time or frequency domain signal. In the case of digital applications it is necessary that the signal be represented in sampled form in both time and frequency domains. Thus, the DFT is of primary interest for obtaining the Fourier Transform when both time and frequency variables are in discrete form. The improved form of DFT is called *Fast Fourier Transform*, FFT, which is a class of efficient algorithms for computing the DFT that gives considerable savings in the number of computations.

The term “fast Fourier transform” was originally used to describe the fast DFT algorithm popularized by Cooley and Tukey (1965) [13]. Their paper cited Danielson and Lanczos (1942) [14] describing a FFT algorithm. Danielson and Lanczos referred to two papers written by Runge and K¨onig in 1924 [15]. Those papers described methods to reduce the number of operations required to calculate a DFT and achieve much greater gains in efficiency, such that complexity can be brought below O(N^2). Almost fifteen years after Cooley and Tukey’s paper, Heideman et al. (1984) [16], published a paper providing even more insight into the history of the FFT.

Nearly every computing platform has a library of highly optimized FFT routines for a wide range of application such as:

- Medical Signal Analysis:
  Examples: Electrocardiogram (ECG), Electromyogram (EMG), Electroencephalogram (EEG). Analyzing ECG signals is an effective method for diagnosing heart abnormalities. The ECG signal collected from a patient is analyzed by a comparison of the amplitudes and period of the P-Q-R-S-T wave to normal waves, using Fourier Transform.
- Solving linear partial differential equations:
  Examples: Laplace equation, Biharmonic equation, Heat Conduction Diffusion and the wave equation.
- Designing and using antennas:
  Examples: Seismic arrays and streamers, Multibeam echo sounder and side scan sonar Interferometers, Synthetic Aperture Radar.
- Image Processing and filters:
  Examples: Transformation, representation, and encoding, Smoothing and sharpening Restoration, blur removal, and the Wiener filter.
- Data Processing and Analysis:
  Examples: Signal and noise estimation, high-pass, low-pass, and band-pass filters, Cross correlation, transfer functions.

This paper describes the content of a learning module prepared to help students understand recent developments in FFT algorithms and architectures. The module includes examples, sequence diagrams, sequential programs, parallel MPI programs and architectures. Section 2 provides an overview of Fourier analysis. Section 3 describes recent high performance FFT algorithms and their implementations. Sequence diagrams are used to help students understand the computations involved. Section 4 analyzes performance measurements. Section 5 provides a summary of the paper.

2. Overview of Fourier analysis, DFT and FFT

Fourier Analysis:

Fourier analysis is based on the concept that periodic signals can be approximated in the frequency spectrum by summing up these periodic signals at different frequencies. Plotting magnitudes on the y-axis and the frequency on the x-axis generates a frequency spectrum.

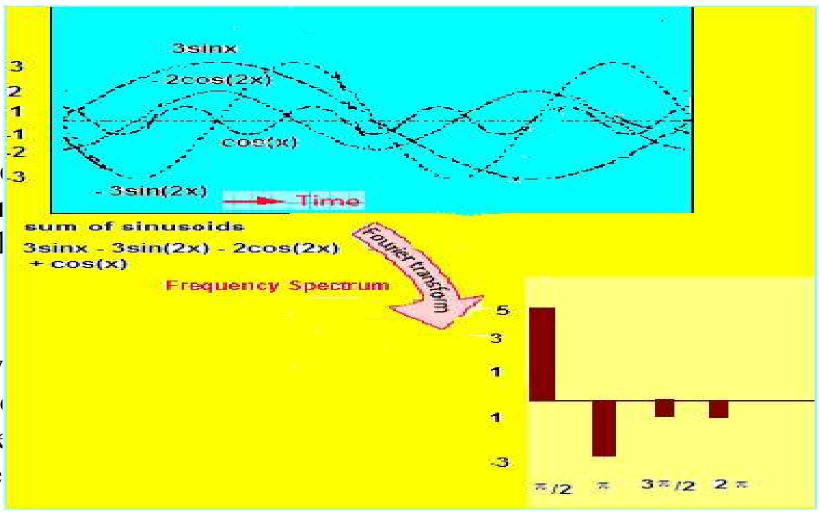


Figure 2.1:

Several techniques have been developed that
spectrum of a signal. The first step in all cas
of digitized samples. This is done by sampli
such as DFT and FFT make use of this sampl

Discrete Fourier Transform

The *Discrete Fourier Transform* can be view
represents a sampling of a signal's distributi
to another sequence, which represents th
function of frequency (the number of comple

For N points, DFT can be represented using the equation:

$$X(k) = \sum_{n=0}^{N-1} \mathrm{x(n)}\, e^{-i2\pi nk/N} \qquad \text{…………..EQ-2.1}$$

Where x(n) is digitized samples in the time domain and X(k) is a sequence in the frequency domain. Here N is the number of input samples and k = 0,1, …….N-1.

DFT is a straight forward implementation with time complexity $\Theta(n^2)$.

Example 2.1:

Let x(n) = {3,5} (Discrete samples in time domain) where n=0,1

To compute 2-point DFT, use equation 2.1:

With k = 0, 1 Fourier Coefficients can be represented as:

$X(0) = x(0)\ e^{-i2\pi\ (0*0)/2} + x(1)\ e^{-i2\pi\ (1*0)/2} \Rightarrow 3*1+5*1 = 8$

$X(1) = x(0)\ e^{-i2\pi\ (1*0)/2} + x(1)\ e^{-i2\pi\ (1*1)/2} \Rightarrow 3*1 + 5*(-1) = -2$

Figure 2.2 shows diagrammatic representation of above results.

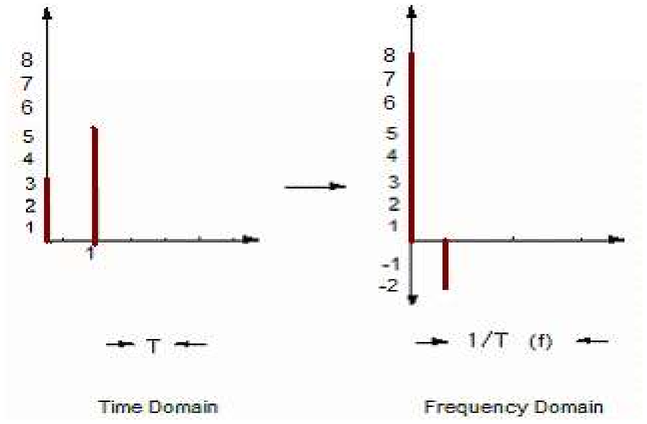


Figure 2.2: DFT - Transformation of Sampled Signal in Time Domain to Frequency domain

The DFT equation can always be represented as Matrix-Vector product: *Fx.*

Here, $f_{i,j} = e^{-i2\pi\ kn/N} = \cos(2\pi*kn/N) - i\sin(2\pi*kn/N) = \omega_N^{kn}$

Where ω_N is called the Twiddle Factor which is an "N^th root of unity" since $\omega_N^N = e^{-i2\pi N/N}$

= 1. And, *x* is the input vector (sampled signal).

Thus the Matrix-Vector product for 2-Point DFT can be represented as:

$$\begin{bmatrix} e^{-i2\pi(0*0)/2} & e^{-i2\pi(1*0)/2} \\ e^{-i2\pi(1*0)/2} & e^{-i2\pi(1*1)/2} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} \omega_2^{0*0} & \omega_2^{0*1} \\ \omega_2^{1*0} & \omega_2^{1*1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} X_0 \\ X_1 \end{bmatrix}$$

Example 2.1 can also be solved using the above Matrix-Vector product representation:

$$\begin{bmatrix} \omega_2^{0*0} & \omega_2^{0*1} \\ \omega_2^{1*0} & \omega_2^{1*1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 3 \\ 5 \end{bmatrix} = \begin{bmatrix} 8 \\ -2 \end{bmatrix} = \begin{bmatrix} X_0 \\ X_1 \end{bmatrix}$$

Hence, the DFT can be computed using N * (N-1) additions and N^2 multiplications. Therefore, roughly 2 N^2 operations are required to calculate the DFT of a length-N sequence.

Fast Fourier Transform

The *Fast Fourier Transform* provides an efficient algorithm to compute DFT by reducing the number of computations from $O(N^2)$ to O(N log N). It basically makes use of two important strategies to obtain an optimized algorithm: one is the *divide and conquer strategy* while the other one is applying the *Halving Lemma*[10] which describes the special quality of ω_N, which is periodicity. By making use of these basic strategies, FFT is generally computed using sequential, parallel and pipeline approaches:

- The *Sequential Approach* makes use of iterative or recursive computations as shown in Figure 2.3.

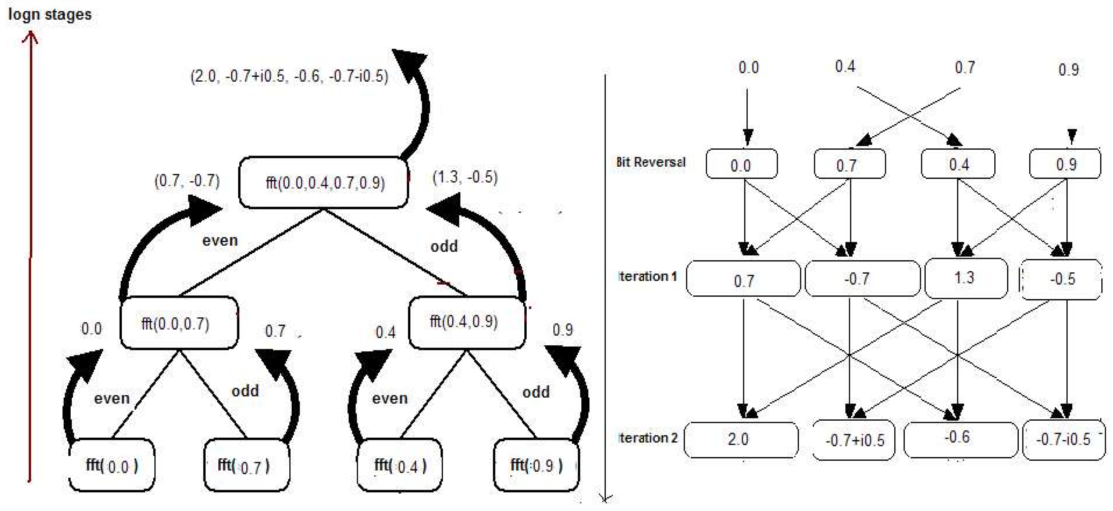


Figure 2.3: Recursive approach **Iterative Approach**

- The *Parallel Approach,* shown in Figure 2.4*,* causes further reduction in the computation time as compared to the sequential approach using Foster's design methodology [10].

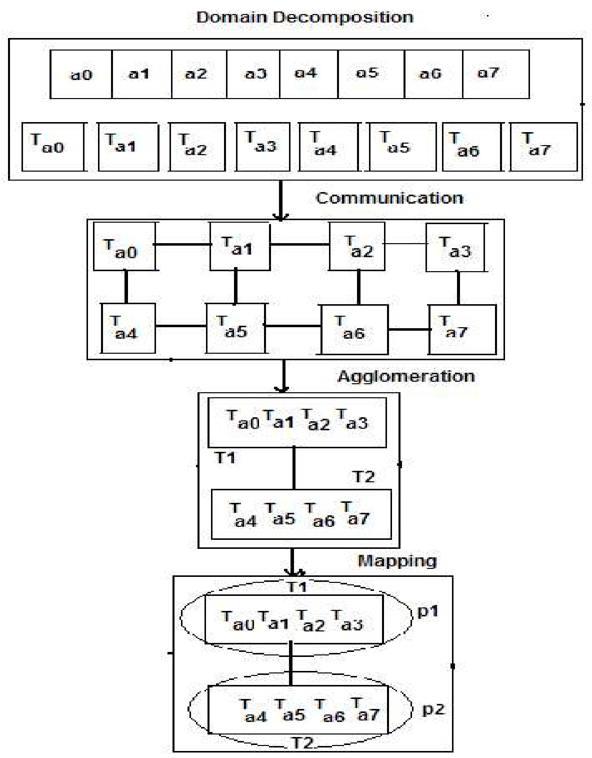


Figure 2.4: Foster’s Parallel FFT Algorithm Design Methodology

- The *Pipeline Approach,* shown in Figure 2.5, is useful for FFTs that require high data throughput. The commutator (denoted C in the Figure 2.5), located between two butterfly stages, reorders the output data from one stage and to the following.

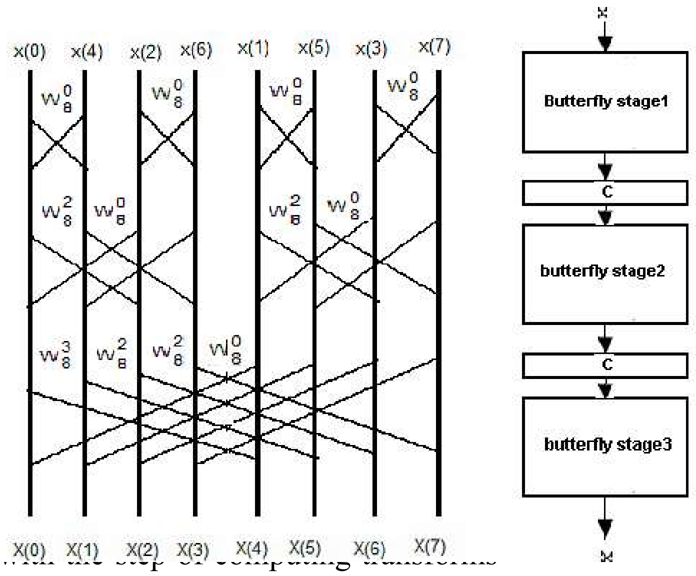

x(0) x(4) x(2) x(6) x(1) x(5) x(3) x(7)
Butterfly stage1
C
butterfly stage2
C
butterfly stage3
X(0) X(1) X(2) X(3) X(4) X(5) X(6) X(7)

Figure 2.5: Basic Structure of Pi

3. Recent High Performance Algori

This section presents several parallel and pipelin
computations and reduce the communication overh

3. 1 Parallel Tree Algorithm:

The name, “Tree” is given to the algorithm becau
place among the processors. The algorithm starts
[3].

Algorithm:

Single dimensional data of size N is saved in a two dimensional Array with dimensions $\sqrt{N} * \sqrt{N}$. Data is scattered to all processors as shown in Figure 3.1, so each processor will have $\sqrt{N}/p$ rows and $\sqrt{N}$ columns of data.

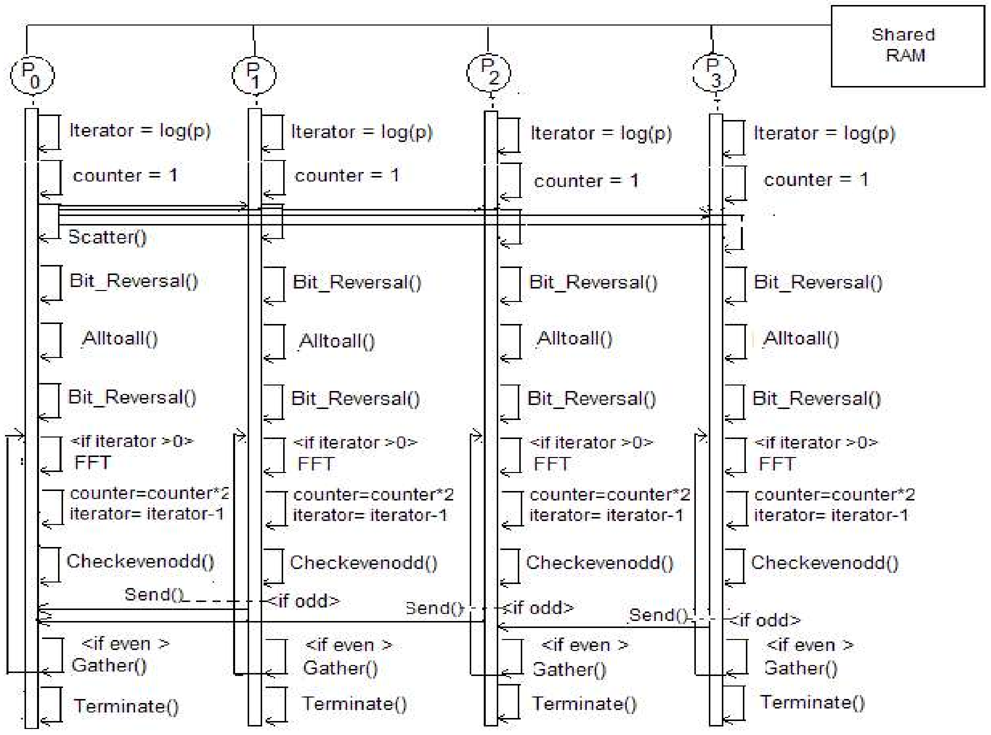


Figure 3.1: Parallel Tree Algorithm

Each processor performs a bit reversal to reorder the input data in the assigned rows. The array is transposed in p steps. All to all communication adds communication overhead as each processor communicates with other active processors. While transposing, each processor sends a square block with size $\sqrt{N}/p * \sqrt{N}/p$ to all other processors. Then, each processor performs a bit reversal rearranging the data for the second time for each assigned row.
Figure 3.2 shows the tree algorithm for 4 processors. . Each processor performs a N/P -point transform. Assume a counter with value 1. Odd-numbered processors whose IDs do not divided by counter *2 send their data to the even-numbered processors whose IDs divided by counter *2 and then terminate the algorithm. After that, the even numbered processors do the next point transform ((counter *2* N)/p –point transform). At this point the counter value gets set to counter *2. Then, the above process is repeated log (p) times until all the data reaches the master processor P0. The communications between the processor follow the tree structure.

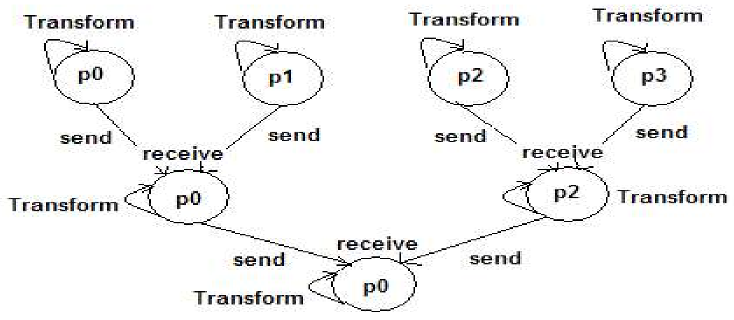


Figure 3.2: Tree algorithm (4 processors), from [3]

The Architecture:
The Parallel Tree Algorithm [3] is implemented on a Symmetric Multiprocessor Architecture, SMP. This is a tightly-coupled Multiple CPU with a shared memory. Each CPU in SMP has full access to the shared memory through a common bus. Communication between nodes occurs via shared memory.

The SMP system used for this Parallel Tree algorithm consists of:

- 8 SUN 250 MHz Ultra SPARC processors
- 2 GB of RAM and 45.5 GB of hard drive space
- Gigaplane bus (crossbar interconnection) where each processor can communicate with any processor directly.

Programming:
The programming model for Parallel Tree Algorithm is based on the standard and widely used Message Passing Interface for programming concurrent activities in parallel computers [4]. This standard library consists of over 125 routines, which is used to develop message passing programs either in C or Fortran.

3.2 Parallel Transpose Algorithm:

The Tree Algorithm suffers from imbalance load. As shown in Figure 3.2, some processors idle while other processors work. P0 finishes the last stage of the algorithm. This causes heavy load imbalance. The Transpose Algorithm provides a solution to this problem [3]. This algorithm distributes the load among the processors while computing Fourier Transform.

Algorithm Description:

This algorithm is similar to the tree algorithm until each processor performs $\sqrt{N}$ -point transform as data size associated with each processor is $\sqrt{N}$. After this the algorithm follows different set of steps than Tree algorithm. The array is transposed (all-to all communication) as before. Each processor does $\sqrt{N}$ -point transform and finally the array is transposed to get the final result. Thus the communication overhead in the case of the Transpose algorithm is 2 times more as compared to the Tree algorithm.

Architecture:
Transpose algorithm is also implemented on the SMP using standard MPI library to carry out certain standard routines such as sending messages from one processor to another, performing all-to-all communication among the processors, computing time required for executing the code etc.

3.3 Radix-4 Modular Pipeline FFT [5]

This algorithm is used to compute long Discrete Fourier Transform (DFT). Radix-4 indicates that the size of data is always power of 4 i.e. $N = 4^v$. The name, modular pipeline FFT, suggests that the architecture used for implementing this algorithm is composed of 2 modules of pipeline FFTs joined together to give the required result.

Algorithm Description:

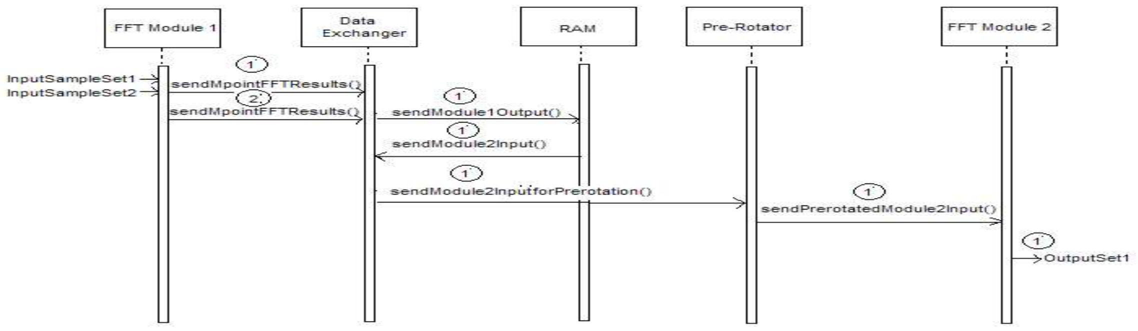


Figure 3.3: Modular Pipeline FFT

As shown in Figure 3.3, the algorithm is divided into following stages [5]:

- First stage: FFT module 1 performs M-point transform on each of M sets of total N points to get intermediate result.
- Second stage: Pre-rotates intermediate results using a correction factor
- Third stage: FFT module 2 performs M-point transform on each of M sets of pre-rotated results to get the final result.

The reason for pre-rotation is that the first stage performs M-point FFT on M sets of total N points. So the twiddle factor used in all M-point transforms is $\omega_{N/M}^{k}$ and not ω_{N}^{k}. But we want to compute FFT of N-points. So before passing the result of the first stage to the second stage, we need to adjust the values obtained in the first stage using the correction factor (ω_{N}^{k}) to get the correct results.

Architecture:
The architecture for implementing this algorithm is shown in Figure 3.4. It consists of two FFT modules, two data exchangers and a pre-rotator. FFT0 is the first stage of pipeline. FFT1 is the second state of pipeline.

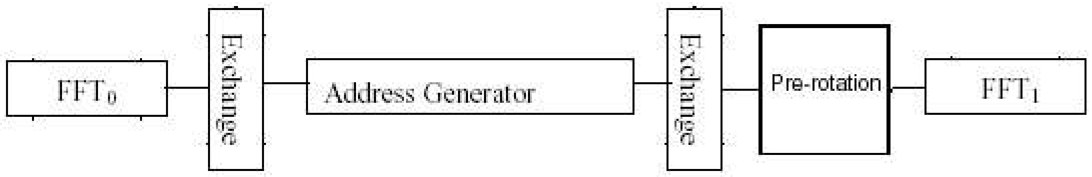


Figure 3.4: Architecture of Modular Pipeline FFT

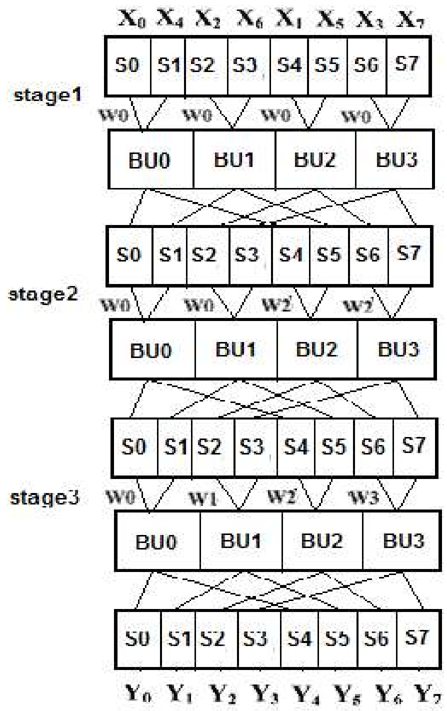
The Data Exchanger switches four times per second stage transform. Each second stage input vector requires a single transformed output of M transforms from each of M sets of first stage. The Address Generator is used to write the outputs from the first stage pipeline into the same memory locations as the points leaving the memory for the second stage pipeline. The Pre-rotator stores the pre-rotation coefficients required for the first stage output in ROM to eliminate on the fly computations that would otherwise be required.

This architecture processes four discrete points per clock cycle. Once *N* points have been output from the FFT0 to the memory, the system can send the data to the FFT1. Simultaneously, a new set of N points can begin processing in the FFT0 [5].

3.4 Array Processing Mapping Algorithm

This algorithm, based on the Singleton Algorithm, [1], makes every two by two computation using a uniform structure. The algorithm makes use of array processing architecture. Efficient mapping reduces data transfers between the processes.

Algorithm:
Figure 3.5 illustrates the Array processing mapping algorithm [1]. In Singleton algorithm each two-by-two transform has a uniform structure and are independent of each other. They can be processed in parallel.



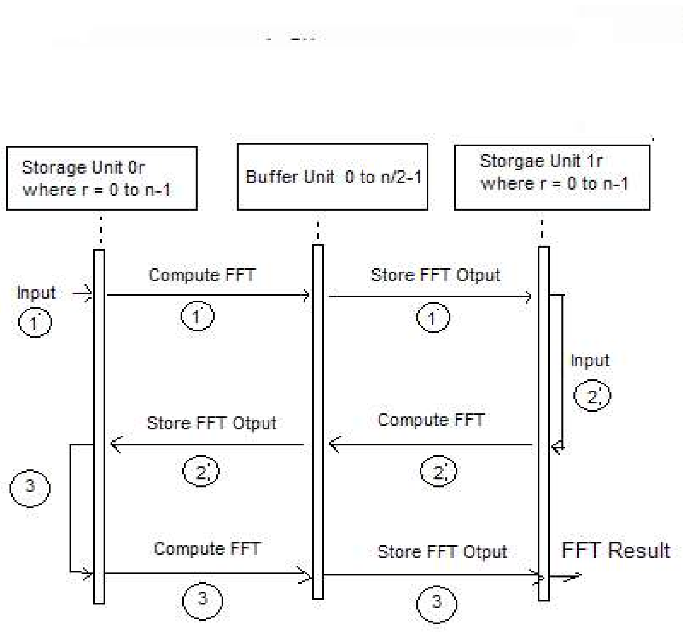


Figure 3.5: Singleton Algorithm   Array Processing Mapping Algorithm, from [1]

Thus, this mapping algorithm basically follows same set of equations used in the Singleton algorithm as shown below:

$$y_j = x_{2j} + x_{2j+1} e^{-i2\pi(j \div 2^{s-k})/2^{k-1}} \qquad \text{EQ..3.18}$$

$$y_{j+n/2} = x_{2j} - x_{2j+1} e^{-i2\pi(j \div 2^{s-k})/2^{k-1}} \qquad \text{EQ..3.19}$$

where, k = 1,2…..s (here s indicates the umber of stages)
j= 0 ,1, ….. n/2-1 (n is the number of input samples)

The sequence diagram of the algorithm is given in Figure 3.7. The array structure with n/2 buffer units BU0… BUn/2-1 is presented in one column. 2n storage elements are shown in two columns. n is the total number of input samples. Storage units in this structure are represented as S_{qr} where r ranges from 0 to n-1 while q has the value of either 0 (for left column) or 1(for right column). So in the first stage, BUs access the input data from left column and store the result in the right column. In the first stage, the left column is used as the source and right column is used as the destination, and so on. In the first stage, ith Buffer Unit (i ranging from 0 to n/2) fetches data from $S_{0(2i)}$ th and $S_{0(2i+1)}$ th storage units from the left column and stores the result to $S_{1(i)}$ th and $S_{1(i+n/2)}$ th storage units of the right column. While in the second stage, ith Buffer Unit (BU) fetches data from $S_{1(2i)}$ th and $S_{1(2i+1)}$ th storage units of right column and stores the result to $S_{0(i)}$ th and $S_{0(i+n/2)}$ th storage units of left column. Thus the process continues for all log n stages.

Architecture:

The Array processing mapping algorithm makes use of an array architecture which typically comprises a number of independent processing elements with local buffers, interconnected through a network.

4. Performance Measurements

This section describes several performance metrics used to compare the algorithms.

Comparison of Parallel Tree and Transpose Algorithms

The Complexity is analyzed as follows [3]:

(Tp) =Computation cost +Communication cost
$Tp = W/P + O(P*P)/P = \Theta(2N\log N/P + P)$
Where W is computation time for sequential algorithm and P is the number of processes

The Speedup and Efficiency are analyzed as follows [3]:

Speedup $(S) = W/Tp = W/(W/P+O(P*P)/P$
Efficiency $(E)= S/P = (W/Tp)/P = W/((W/P + O(P*P)/P)*P = W/(W+O(P*P))$

For 2 processors the Tree algorithm provides better efficiency than the Transpose algorithm since the load imbalance does not have any effect. As the number of processors is increased, the Transpose algorithm provides better efficiency as compared with the Tree algorithm.

Isoeffciency [3] determines whether an algorithm maintains constant efficiency. To have a constant efficiency the rate of increase in the number of computations should be directly proportional to the rate of increase in communication overhead.

Re-writing the efficiency equation:
W = E/E-1 $*O(P*P)$
$= k *O(P*P)$ => equation of Isoeffciency

Scalability [3] of these algorithms is computed by determining the changes in speedup or efficiency as we increase the number of processors. The Transpose algorithm is more scalable than the Tree, as it avoids imbalance load.

Comparison of Modular Pipeline and Regular Pipeline Algorithms:
The analysis of the Modular Pipeline Algorithm is done based on ***Latency, Complexity and Efficiency*** *[5]*. Latency of the pipeline algorithm is determined by the amount of time required to perform N-point FFT. The Modular pipeline requires an additional

amount of time, compared to the conventional pipeline. So the equation of Latency for modular pipeline can be written as:
$T_{Modular} = T_{conventional} + T_{Delay} = 2 *(N/r) + 2(\sqrt{N} /r - 1)$

The Complexity of the pipeline algorithm is computed in terms of coefficient memory i.e. ROM. The Modular pipeline requires less amount of ROM as compared with the conventional pipeline. Efficiency of the pipeline is measured as the percentage of utilized memory which is quite high in the case of the modular pipeline compared with the conventional pipeline.

Analysis of Array Processing Mapping Algorithm:
Analysis of this algorithm is done based on criteria such as Latency and Throughput [1].

The Latency of the array processing algorithm is given by:
$(s\times 2^{s-k} + pipeline_{length})\times t_{clk}$
and the throughput of the array processing algorithm is given by:
$n/(s\times 2^{s-k} \times t_{clk})$
Where, 2^s represents input size (n) of FFT where s indicates the number of stages required to compute FFT.2^k represents the number of buffer units required for computation of FFT where k ranges from 0 to s-1 and t_{clk} denotes the clock period of the system.

5. Conclusion

This paper presented recent advances in parallel and pipeline FFT algorithms and their mapping architectures. Out of these algorithms, the Parallel Tree and Transpose, follows the parallel architecture while modular pipeline algorithm follows Parallel pipelined architecture and Array processing mapping follows array architecture. A learning module that can be used in the classroom based on the algorithms and architectures presented in this paper is at [12].

The FFTW [11], Fastest Fourier Transform in the West, consists of MPI routines that adapt to the hardware and are competitive with hand tuned libraries. FFTW is searching for flexible and adaptive software architecture.

Acknowledgments

I sincerely thank Dr. Don Hamnes, Dr. S. Herath and Dr. J. Herath for their help in this research.

References

[1] Zhenyu Liu, Yang Song, Takeshi Ikenaga, Satoshi Goto, “A VLSI Array Processing Oriented Fast Fourier Transform Algorithm and Hardware Implementation”, *ACM Great Lake Symposium on VLSI*, 2005

[2] K. Scott and Hemmert Keith D. Underwood “An Analysis of the Double-Precision Floating-Point FFT on FPGAs”, *IEEE Symposium on FPGAs for Custom Computing Machines*, 2005

[3] Rami A. AL Na'mneh, W. David Pan, B. Earl Wells “Two Parallel Implementations for One Dimension FFT on Symmetric Multiprocessors”, *ACM Southeast* 2004.

[4] A. Grama, A. Gupta, V. Kumar and G. Karypis,” Introduction to Parallel Computing”, Pearson Education 2003.

[5] Ayman M. El-Khashab, Earl E. Swartzlander, “Jr. An Architecture for a Radix-4 Modular Pipeline Fast Fourier Transform”, *IEEE Symposium on Application-specific Systems, Architectures and Processors,* 2003.

[6] Jonas Claeson, “Design and implementation of an asynchronous pipelined FFT “, 2003

[7] Ayman M. El-Khashab Earl E. Swartzlander “A Modular Pipelined Implementation of Large Fast Fourier Transforms”, *IEEE Proceedings of the Thirty-Sixth Asilomar Conference on Signals, Systems, and Computers*, 2002

[8] D. Takahashi, “High-Performance Parallel FFT Algorithms for the HITACHI SR8000”, *The Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, Vol. 1, pp. 192–199, May 2000.

[9] D. Takahashi, Y. Kanada, “High-Performance Radix-2, 3 and 5 Parallel 1-D Complex FFT Algorithms for Distributed-Memory Parallel Computers”, *Journal of Supercomputing*, Vol. 15, No. 2, pp. 207-228, February 2000.

[10] M .J. Quinn, Designing Efficient Algorithm for Parallel Computers, New York: McGraw Hill, 2004.

[11] Matteo Frigo, Steven G. Johnson, “The Design and Implementation of FFTW3”, *Proceedings of the IEEE* **93** (2), 216–231 (2005).

[12] http://web.stcloudstate.edu/jherath/csci697/FFTAcLearn.htm

[13] Cooley, J. W. and J. W. Tukey, “An Algorithm for the Machine Calculation of Complex Fourier Series.”, Math. of Comput., volume 19, pp. 297–301, April 1965.

[14] Danielson, G. C. and C. Lanczos. “Some Improvements in Practical Fourier Analysis and Their Application to X-ray Scattering From Liquids.”, J. Franklin Inst., volume 233, pp. 365–380,435–452, April 1942.

[15] Runge, C. and H. K¨onig, “Die Grundlehren der Mathematischen Wissenschafter.”, Vorlesungen ¨uber Numerisches Rechnen, volume 11, Berlin, 1924. Julius Springer.

[16] Heideman, M. T., D. H. Johnson, and C. S. Burrus, “Gauss and the History of the Fast Fourier Transform.”, IEEE ASSP Magazine, pp. 14–21, October 1984.