BENCHMARKING OF FFT ALGORITHMS[*]

Michael Balducci, Aravind Ganapathiraju, Jonathan Hamaker, Joseph Picone
Department of Electrical and Computer Engineering
Mississippi State University
Mississippi State, Mississippi 39762, USA
Ph (601) 325-3149 - Fax (601) 325-3149
{ganapath, hamaker, picone}@isip.msstate.edu

Ajitha Choudary, Anthony Skjellum

Department of Computer Science
Mississippi State University
Mississippi State, Mississippi 39762, USA
Ph (601) 325-8435 - Fax (601) 325-8997
{ajitha, tony}@cs.msstate.edu

Abstract - **A large number of Fast Fourier Transform (FFT) algorithms have been developed over the years. Among these, the most promising are the Radix-2, Radix-4, Split-Radix, Fast Hartley Transform (FHT), Quick Fourier Transform (QFT), and the Decimation-in-Time-Frequency (DITF) algorithms. In this paper, we present a rigorous analysis of these algorithms that includes the number of mathematical operations, computational time, memory requirements, and object code size. The results of this work will serve as a framework for creating an object-oriented, poly-functional FFT implementation which will automatically choose the most efficient algorithm given user-specified constraints.**

INTRODUCTION

Though development of the Fast Fourier Transform (FFT) algorithms is a fairly mature area, several interesting algorithms have been introduced in the last ten years that provide unprecedented levels of performance. The first major breakthrough was the Cooley-Tukey algorithm developed in the mid-sixties which resulted in a flurry of activity on FFTs [2]. Further research led to the development of the Fast Hartley Transform, and Split-Radix algorithm. Recently, two new algorithms have also emerged: the Quick Fourier Transform and the Decimation-in-Time-Frequency algorithm.

While there has been extensive research on the theoretical efficiency of these algorithms, there has been little research to-date comparing these algorithms on practical terms. Architectures have become quite complex today with multi-level caches, super-pipelined processors, long word instruction sets, etc. Efficiency, as we will show, is intricately related to how an algorithm can be implemented on a given architecture. The issues to be considered include computation speed, memory, algorithm complexity, machine architecture, and the compiler design. In this paper, we report on preliminary work to find the most efficient algorithm under application-specific constraints. Our approach is to benchmark each algorithm under a variety of constraints, and to use the benchmark statistics to create a wrapper capable of choosing the algorithm best suited for a given application. Obviously, object-oriented programming methodologies will play a large role.

[*] This work was supported by DARPA through US Air Force’s Rome Laboratories under contract F30602-96-1-0329.

ALGORITHMS

The definition of the Discrete Fourier Transform (DFT) is shown in (1).

$$X(k) = \sum_{n=0}^{N-1} x_n e^{\frac{-j2\pi kn}{N}} \qquad (1)$$

Most of the algorithms take the divide-and-conquer approach to reduce computations. The Radix-2 and Radix-4 approaches decompose the N-point DFT computations into sets of two and four-point DFTs, respectively [2]. The core computation in a Radix-4 butterfly involves fewer complex multiplications than the Radix-2 butterfly, yielding an increase in efficiency when the order of the transform is a power of 4. To take advantage of this fact, the Split-radix algorithm makes use of both the Radix-2 and Radix-4 decomposition [3].

The Hartley Transform, shown in (2), further reduces computation by replacing the complex exponential term in the DFT with a kernel using real variables [4].

$$X_H(k) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x_n \left[\cos\left(\frac{2\pi kn}{N}\right) + \sin\left(\frac{2\pi kn}{N}\right) \right] \qquad (2)$$

This reduces the number of real multiplications and additions, with only a modest gain in memory. The main drawback of the Hartley Transform is the additional computation needed to transform the results from the real Hartley coefficients to the standard complex Fourier coefficients. However, since the relationship between the two forms of coefficients is linear, the additional cost incurred is less than the efficiency gained.

The Quick Fourier Transform (QFT) uses the symmetry of the cosine and sine terms to reduce the number of complex calculations [5]. The QFT breaks a signal into its even and odd components. A Discrete Cosine Transform (DCT) is used on the even samples to calculate the real portions of the Fourier coefficients, while a Discrete Sine Transform (DST) is used on the odd samples to compute the imaginary portions. Both the DCT and DST are in turn computed recursively. An important aspect of the QFT is that all complex operations occur at the last stage of recursion, making it well-suited for real data.

Finally, the Decimation-In-Time-Frequency (DITF) algorithm leverages the Radix-2 approach in both the time-domain (DIT) and frequency (DIF). The DITF is based on the observation that the DIT algorithm has a majority of its complex operations towards the end of the computation cycle and the DIF algorithm has a majority towards the beginning. The DITF makes use of this fact by performing the DIT at the outset and then switching to a DIF to complete the transform. Combining these algorithms comes at the cost of computing complex conversion factors at the transition stage [6].

EVALUATION METHODOLOGY

Criteria: Computation speed was selected as the core criteria for comparison since the fastest method is generally the most desirable one. However, the amount of memory available is not unlimited; therefore, memory was also included as a measure of efficiency. The number of mathematical operations is also important since it is directly related to the computation time and the hardware requirements. The additions and multiplications were broken into floating-point and integer operations because floating point operations are more costly in computation time than are integer operations (on most hardware). Of course, all of this is mitigated by the degree to which the compiler can perform optimizations. Modern compilers are able to optimize code for speed and hardware usage by such techniques as loop-unrolling, delayed-branching, etc. The level of optimization performed on an algorithm will be highly algorithm and implementation dependent.

Implementation: Bearing in mind the evaluation criteria, we designed a class structure that is intuitive and allows for easy inclusion of new algorithms to the existing set. Computation speed is measured using system utilities accurate to $1\ \mathrm{ms}$. For evaluation of memory usage, we developed floating-point and integer classes that have the features of accumulating a count for every variable that is declared and counting the number of mathematical operations. This gives a very efficient method for viewing the dynamic memory usage. An iterative approach to testing was also used to reduce the transients of processor loading. This method involved running each test for a large number of iterations, using median values for comparison.

RESULTS

An often used criterion for comparison of FFT algorithms thus far has been the number of mathematical operations involved. These statistics usually do not, however, account for the cost of incrementing integer counters and indices. Integer operations can be a significant portion of the computation time. Operation counts for a 1024-point complex FFT is presented in Table 1. Most algorithms trade floating-point operations for integer operations.

A criterion closely related to the number of operations is the computation speed. A summary of the computation times for the selected algorithms is shown in Table 2. One would expect the algorithm with the least number of operations to be the fastest. However, we show that compiler optimizations play a large role by virtue of the large difference between the FHT and all other algorithms. It is interesting to note that the difference in performance tends to decrease as the order increases. As the order increases in the FHT, the cost of converting the Hartley coefficients to Fourier coefficients seems to becomes substantial, explaining the less dramatic difference in performance.

It is a well-known fact that most FFT algorithms achieve an $O[N \log N]$ complexity. The constants of proportionality are what differentiates the performance of the algorithms. As shown in Table 2, second-order effects can be dominated by compiler efficiency. For lower orders the FHT algorithm in its present implementation seems to be making better use of the cache than the other algorithms. This advantage flattens out as the order increases, though.

Algorithm	Float Mults	Float Adds	Int Mults	Int Adds	Bin Shifts
RAD-2	20480	30720	0	15357	1024
RAD-4	15701	28842	336	8877	2738
SRFFT	10016	25488	502	12448	2937
FHT	18704	32056	0	8367	4246
QFT	8448	31492	16	70058	316
DITF	16640	28800	1076	18839	1086

Table 1. Comparison of mathematical operations involved in a 1024-point complex FFT.

	FFT Order				
Algorithm	64	256	1024	4096	16384
RAD-2	238	952	3952	17857	77714
RAD-4	191	762	3476	14714	60333
SRFFT	238	905	3810	17429	74524
FHT	48	286	1333	7143	31905
QFT	143	762	3476	15952	78000
DITF	238	1000	4191	18714	80333

Table 2. Comparison of computation speed [in μsec] for varying FFT orders on a 200 MHz processor UltraSparc machine compiled using gcc with optimization level 3. The reported speeds are median values over 1000 iterations.

Another common design criterion in practical systems is memory. Algorithm efficiency can always be traded for memory and code size. Table 3 illustrates this fact.

CONCLUSIONS

We have presented results on a comprehensive collection of FFT algorithms, each of which was programmed in a similar framework. We have generated statistics to supplement the mathematical formulation for the complexity of these algorithms. Combined, this data gives the developer a clear picture of the computational requirements of each algorithm.

At our current level of implementation, the FHT appears to be the best overall algorithm. It is interesting to note that the higher number of mathematical operations does not necessarily translate into a reduction in speed. The FHT requires a marginally higher number of floating-point operations than the Radix-4, yet the FHT is faster than the Radix-4 by more than a factor of 2. This implies that the FHT makes more efficient use of the resources available (cache, pipelining, etc.) than does the Radix-4. Also, this makes it even more important to look at the order of operations because multiplications and additions that can be pipelined need much less computation time than algorithms where the flow of operations cannot be easily pipelined. This is an especially important consideration for the current generation of complex architectures (such as the Pentium Pro chip).

Algorithm	Memory	Object Size
RAD-2	24616	1740
RAD-4	8344	2020
SRFFT	24656	2232
FHT	32832	4812
QFT	49152	7520
DITF	24616	3060

Table 3. Comparison of memory usage [in bytes] for a 1024-point complex FFT on a 200 MHz UltraSparc compiled using gcc with optimization level 3. Note that peak memory requirements are shown.

We also see that, in general, the number of integer additions is a good indication of speed. Note that the ranking of algorithms by speed is almost the same as the ranking of the algorithms by integer additions. The only exception is the QFT that seems to make up for the high number of integer operations by using very few floating-point multiplications.

Our work has laid the foundation for an “intelligent” environment which will automatically choose the best algorithm and execute it for a given set of user constraints. As the hardware available is becoming more specialized, it becomes imperative that the software take full advantage of the hardware capabilities. In the future, our work will be extended to parallel processing environments. Detailed documentation of our experiments can be downloaded from *http://www.isip.msstate.edu/software/parallel_dsp.*

ACKNOWLEDGEMENTS

We gratefully acknowledge the suggestions and assistance given by Shane Hebert of ICDCRL in the Computer Science Department at Mississippi State University.

REFERENCES

[1] J.W. Cooley and J.W. Tukey, “An Algorithm for Machine Computation of Complex Fourier Series,” *Math. Comp.*, vol. 19, pp. 297-301, April 1965.

[2] C.S.Burrus and T.W.Parks, *DFT/FFT and Convolution Algorithms: Theory and Implementation*, John Wiley and Sons, New York, NY, USA, 1985.

[3] P. Duhamel and H. Hollomann, “Split Radix FFT Algorithm,” *Electronic Letters*, vol. 20, pp. 14-16, Jan. 1984.

[4] R. Bracewell, *The Hartley Transform*, Oxford Press, Oxford, England, 1985.

[5] H. Guo, G.A. Sitton, and C.S. Burrus, “The Quick Discrete Fourier Transform,” *Proc. of ICASSP,* vol. III, pp. 445-447, Adelaide, Australia, April 1994.

[6] A. Saidi, “Decimation-In-Time-Frequency FFT Algorithm.” *Proceedings of ICASSP,* vol. III, pp. 453-456, Adelaide, Australia, April 1994.