A COMPARATIVE ANALYSIS OF FFT ALGORITHMS

Aravind Ganapathiraju, Jonathan Hamaker, Joseph Picone

Institute for Signal and Information Processing (ISIP)
Department of Electrical and Computer Engineering
Mississippi State University
Mississippi State, MS 39762

Anthony Skjellum

High Performance Computing Lab (HPCL)
Department of Computer Science
Mississippi State University
Mississippi State, MS 39762

ABSTRACT

With the rapid development of computer technology, general purpose CPUs have made inroads into many signal processing applications; of which the Fast Fourier Transform (FFT) continues to be an integral part. A large number of FFT algorithms have been developed over the years, notably the Radix-2, Radix-4, Split-Radix, Fast Hartley Transform (FHT), Quick Fourier Transform (QFT), and the Decimation-in-Time-Frequency (DITF) algorithms. How these algorithms fare in comparison with each other is of considerable interest to developers of signal processing technology. In previous benchmarking efforts, only the computation speed or the number of mathematical operations were used for assessing efficiency. Moreover, most of these benchmarks have been limited to special purpose CPUs like DSPs.

In this paper, we present a rigorous analysis of the aforementioned algorithms on general purpose processors, such as the DEC Alpha, Intel Pentium Pro and Sun UltraSparc. The analysis of each algorithm includes the number of mathematical operations, computation time, memory requirements, and compiler effects. Our work is one of the first efforts to characterize FFT algorithms in terms of memory requirements and detailed operation counts. The results indicate that the FHT is the overall best algorithm on all platforms, offering the fastest execution time and requiring reasonably small amounts of memory.

EDICS: SP 2.2.6

CORRESPONDENCE: Aravind Ganapathiraju
Institute for Signal and Information Processing
Department of Electrical and Computer Engineering, PO Box 9571
Mississippi State University, Mississippi State, MS 39762
Phone: (601) 325-8335 Fax: (601) 325-3149
Email: ganapath@isip.msstate.edu

1. INTRODUCTION

The first major breakthrough in implementation of Fast Fourier Transform (FFT) algorithms was the Cooley-Tukey [1] algorithm developed in the mid-1960s, which reduced the complexity of a Discrete Fourier Transform from O(N^2) to O($N \cdot \log N$). At that time, this was a substantial saving for even the simplest of applications. Since then, a large number of FFT algorithms have been developed. The Cooley-Tukey algorithm became known as the Radix-2 algorithm and was shortly followed by the Radix-3, Radix-4, and Mixed Radix algorithms [8,11,12]. Further research led to the Fast Hartley Transform (FHT) [2,3,4] and the Split Radix (SRFFT) [5,11,12] algorithms. FFT research was considered a fairly mature area by the mid-1980's, but recently, two new algorithms have emerged: the Quick Fourier Transform (QFT) [6] and the Decimation-In-Time-Frequency (DITF) algorithm [7].

While there has been extensive discussion on the theoretical efficiency of these algorithms, there has been little research to-date comparing algorithms in practical terms. Efficiency is intrinsically related to how an algorithm can be implemented on a given architecture. The important issues to be considered in such evaluations of efficiency are the computation speed, memory, algorithm complexity, machine architecture and compiler design. Many benchmarks for FFTs [16,17,18,19,22] have incorporated some subset of these, but none are as comprehensive as the study presented here. One of the earliest benchmarking efforts [19] is typical of such studies, and included Radix algorithms evaluated on Cray, VAX, PDP and IBM machines. This study included complexity analysis in terms of computations. In developing the FHT, Bracewell [16,21] provided a comparison of the FHT and an optimized Radix FFT, and confirmed the efficiency of FHT on a HP platform. A more recent effort started at the Laboratory for Computer Science at MIT [24] includes a comparison of many publicly available algorithms on contemporary high speed platforms like Pentium Pro, UltraSparc etc. Their comparison, however, is focused on implementations rather than algorithms, and does not include a detailed complexity analysis or coding of the algorithms in a common framework, which are essential for a fair complexity analysis.

In this paper we provide a comprehensive comparison of several contemporary FFT algorithms on state-of-the-art processors. The criteria used are the operations count, CPU time, memory usage, processor,

and compiler. The processors evaluated include the DEC Alpha, Intel Pentium Pro and Sun UltraSparc. Preliminary work on quantifying the effects of compilers on these algorithms is also presented. We chose the following algorithms for our analysis: Radix-2 (RAD2), Radix-4 (RAD4), SRFFT, FHT, QFT, and DITF. The choice of these algorithms was influenced by a desire to do a longitudinal study as well as a complexity analysis. The QFT and the DITF are the latest proposed algorithms, while the Radix-2 is the oldest. As we expected in doing such a study, contrary to published results [6,7] these newest entries are not the fastest algorithms available.

2. REVIEW OF FFT ALGORITHMS

The basic principle behind most Radix-based FFT algorithms is to exploit the symmetry properties of a complex exponential that is the cornerstone of the Discrete Fourier Transform (DFT). These algorithms divide the problem into similar sub-problems (butterfly computations) and achieve a reduction in computational complexity. All Radix algorithms are similar in structure differing only in the core computation of the butterflies. The FHT differs from the other algorithms in that it uses a real kernel, as opposed to the complex exponential kernel used by the Radix algorithms. The QFT postpones the complex arithmetic to the last stage in the computation cycle by separately computing the Discrete Cosine Transform(DCT) and the Discrete Sine Transform(DST). The DITF algorithm uses both the Decimation-In-Time (DIT) and Decimation-In-Frequency (DIF) frameworks for separate parts of the computation to achieve a reduction in the computational complexity.

2.1. Radix-2 Decimation in Frequency Algorithm

The RAD2 DIF algorithm is obtained by using the divide-and conquer approach to the DFT problem. The DFT computation is initially split into two summations, one of which involves the sum over the first $N/2$ data points and the other over the next $N/2$ data points, resulting in

$$X(k) = \sum_{n=0}^{N/2-1} x(n) \cdot W_N^{kn} + \sum_{n=N/2}^{N-1} x(n) \cdot W_N^{kn}. \quad (1)$$

Since $W_N^k = e^{-j2\pi k/N}$ and $W_N^{kN/2} = (-1)^k$, the above equation can be simplified to

$$X(k) = \sum_{n=0}^{N/2-1} \left(x(n) + (-1)^k \cdot x\left(n + \frac{N}{2}\right) \right) \cdot W_N^{kn} . \quad (2)$$

Considering the even and odd-numbered frequency samples separately results in

$$X(2k) = \sum_{n=0}^{N/2-1} \left(x(n) + x\left(n + \frac{N}{2}\right) \right) \cdot W_{N/2}^{kn} \text{ and,} \quad (3)$$

$$X(2k+1) = \sum_{n=0}^{N/2-1} \left\{ \left(x(n) - x\left(n + \frac{N}{2}\right) \right) \cdot W_{N/2}^{kn} \right\} \cdot W_{N/2}^{kn} . \quad (4)$$

The same computational procedure can be repeated through decimation of the $N/2$-point DFTs X(2k) and X(2k+1). The entire process involves $\upsilon = \log_2 N$ stages with each stage involving $N/2$ butterflies. Thus the RAD2 algorithm involves $N/2 \cdot \log_2 N$ complex multiplications and $N \cdot \log_2 N$ complex additions, or a total of $5N \cdot \log_2 N$ floating point operations. Observe that the output of the whole process is out-of-order and requires a bit-reversal operation to place the frequency samples in the correct order.

2.2. Radix-4 Algorithm

The RAD4 algorithm is very similar to the RAD2 algorithm in concept. Instead of dividing the DFT computation into halves as in RAD2, a four-way split is used. The N-point input sequence is split into four subsequences, $x(4n)$, $x(4n+1)$, $x(4n+2)$, and $x(4n+3)$, where $n = 0, 1, \ldots, N/4 - 1$. Then,

$$X(k) = \sum_{n=0}^{N/4-1} x(n) \cdot W_N^{kn} + \sum_{n=N/4}^{N/2-1} x(n) \cdot W_N^{kn} + \sum_{n=N/2}^{3N/4-1} x(n) \cdot W_N^{kn} + \sum_{n=3N/4}^{N-1} x(n) \cdot W_N^{kn} . \quad (5)$$

Setting

$$F(l,q) = \sum_{m=0}^{N/4-1} x(l,m) \cdot W_{N/4}^{mq}, \quad (6)$$

$$X(p, q) = X\left(\frac{N}{4} \cdot p + q\right), \tag{7}$$

and,

$$x(l, m) = x(4m + l) \quad , \quad \begin{matrix} l, p = 0, 1, 2, 3 \\ m, q = 0, 1, \ldots, N/4 - 1 \end{matrix}, \tag{8}$$

the matrix formulation of the butterfly becomes

$$\begin{bmatrix} X(0, q) \\ X(1, q) \\ X(2, q) \\ X(3, q) \end{bmatrix} = \begin{bmatrix} W_N^0 & W_N^0 & W_N^0 & W_N^0 \\ W_N^q & -jW_N^q & -W_N^q & jW_N^q \\ W_N^{2q} & -W_N^{2q} & W_N^{2q} & -W_N^{2q} \\ W_N^{3q} & jW_N^{3q} & -W_N^{3q} & -jW_N^{3q} \end{bmatrix} \cdot \begin{bmatrix} F(0, q) \\ F(1, q) \\ F(2, q) \\ F(3, q) \end{bmatrix}. \tag{9}$$

Figure 1 shows the computation of a RAD4 butterfly. The decimation process is similar to the RAD2 algorithm, and uses $\upsilon = \log_4 N$ stages, where each stage has $N/4$ butterflies. The RAD4 butterfly involves 8 complex additions and 3 complex multiplications, or a total of 34 floating point operations. Thus, the total number of floating point operations involved in the RAD4 computation of an N-point DFT is $4.25N \cdot \log_2 N$, which is 15% less than the corresponding value for the RAD2 algorithm.

2.3. Split-Radix Algorithm

Standard RAD2 algorithms are based on the synthesis of two half-length DFTs and similarly RAD4 algorithms are based on the fast synthesis of four quarter-length DFTs. The SRFFT algorithm is based on the synthesis of one half-length DFT together with two quarter-length DFTs. This is possible because, in the RAD2 computations, the even-indexed points can be computed independent of the odd-indexed points. The SRFFT algorithm uses the RAD4 algorithm to compute the odd-numbered points. Hence, the N-point DFT is decomposed into one $N/2$-point DFT and two $N/4$-point DFTs

$$X(2k) = \sum_{n = 0}^{N/2 - 1} \left(x(n) + x\left(n + \frac{N}{2}\right)\right) \cdot W^{4kn}, \tag{10}$$

$$X(4k+3) = \sum_{n=0}^{N/4-1} [g(n) + jf(n)]W^{3n} \cdot W^{4nk} \text{ and,} \quad (11)$$

$$X(4k+1) = \sum_{n=0}^{N/4-1} [g(n) - jf(n)]W^{n} \cdot W^{4nk}, \quad (12)$$

where,

$$g(n) = x(n) - x\left(n + \frac{N}{2}\right) \qquad and \qquad f(n) = x\left(n + \frac{N}{4}\right) - x\left(n + \frac{3N}{4}\right). \quad (13)$$

An N-point DFT is obtained by successive use of these decompositions. Figure 2 illustrates the decomposition of a 32-point FFT using the SRFFT paradigm. Here we treat the computational process as a RAD2 algorithm with the unnecessary intermediate DFT computations eliminated. For each q, the associated line segments indicate which L-point DFT is computed, where $L = 2^q$. Thus at stage $q = 2$, five 4-point DFTs are computed. An analysis of the butterfly structures [15] for the SRFFT algorithm reveals that approximately $4N \cdot \log_2 N$ computations are required as compared to $4.25N \cdot \log_2 N$ for RAD4 and $5N \cdot \log_2 N$ for RAD2 algorithms.

2.4. Fast Hartley Transform

The main difference between the DFT computations previously discussed and the Discrete Hartley Transform (DHT) is the core kernel [2]. For the DHT, the kernel is real unlike the complex exponential kernel of the DFT. The k^{th} DHT coefficient is expressed in terms of the input data points as

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot \left[\cos\left(\frac{2\pi kn}{N}\right) + \sin\left(\frac{2\pi kn}{N}\right)\right]. \quad (14)$$

This results in the replacement of complex multiplications in a DFT by real multiplications in a DHT. For complex data, each complex multiplication in the summation requires four real multiplications and two real additions using the DFT. For the DHT, this computation involves only two real multiplications and one

real addition. There exists an inexpensive mapping of coefficients from the Hartley domain to the Fourier domain, which is required to convert the output of a DHT to the traditional DFT coefficients. Equation 15 relates the DFT coefficients to the DHT coefficients for an N-point DFT computation.

$$\begin{aligned} Re(DFT(k)) &= \frac{DHT(k) + DHT(N-k)}{2} \\ Im(DFT(k)) &= \frac{DHT(k) - DHT(N-k)}{2} \end{aligned} . \tag{15}$$

The FHT evolved from principles similar to those used in the RAD2 algorithm to compute DHT coefficients efficiently. It is intuitively simpler and faster than the FFT algorithms as the number of computations reduces drastically when we replace all complex computations by real computations. Similar to other recursive Radix algorithms, the next higher order FHT can be obtained by combining two identical preceding lower order FHTs. In fact all Radix-based algorithms used in FFT implementations can be applied to FHT computations [22].

For $N = 2$, the Hartley transform can be represented in a matrix form as

$$\begin{bmatrix} X(0) \\ X(1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} x(0) \\ x(1) \end{bmatrix} . \tag{16}$$

Following a similar procedure for $N = 4$, we get the matrix formulation,

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{bmatrix} , \tag{17}$$

which can be easily transformed to

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x(0) \\ x(2) \\ x(1) \\ x(3) \end{bmatrix} . \tag{18}$$

A closer look at this matrix product and a comparison with the matrix for $N = 2$ reveals that the matrix for $N = 4$ is composed of sub-matrices of the form of the matrix for $N = 2$. Thus a DHT of order 4 can be computed directly from a DHT of order 2. This idea can be extended to any order which is a power of 2 [4]. It is also worth noting that the Hartley Transform is a bilateral transform, i.e. the same functional form can be used for both the forward and inverse transforms. This is an added advantage of the FHT over other FFT algorithms.

2.5. Quick Fourier Transform

We have seen that the Radix-based algorithms exploit the periodic properties of the cosine and sine functions. In the Quick Fourier Transform (QFT) algorithm, the symmetry properties of these functions are used to derive an efficient algorithm.

$$\begin{aligned} \cos\left(\frac{2\pi(N-n)k}{N}\right) &= \cos\left(\frac{2\pi nk}{N}\right) \\ \sin\left(\frac{2\pi(N-n)k}{N}\right) &= -\sin\left(\frac{2\pi nk}{N}\right) \end{aligned} \tag{19}$$

We define an $N + 1$ -point DCT as

$$X_{DCT}(k) = \sum_{n=0}^{N} x(n)\cos\frac{\pi nk}{N} \qquad ,k = 0, 1, \ldots, N. \tag{20}$$

An $N - 1$ -point DST can also be similarly defined as

$$X_{DST}(k) = \sum_{n=1}^{N-1} x(n)\sin\frac{\pi nk}{N} \qquad ,k = 1, 2, \ldots, N-1. \tag{21}$$

We can divide an N-point input sequence into its even and odd parts as

$$\begin{aligned} x_e(0) &= x(0) \\ x_e(k) &= x(k) + x(N-k) \qquad ,k = 1, 2, \ldots, N/2-1, \text{ and} \\ x_e(N/2) &= x(N/2) \end{aligned} \tag{22}$$

$$x_o(k) = x(k) - x(N-k) \qquad ,k = 1, 2, \ldots, N/2 - 1 . \tag{23}$$

Using the above sequences and properties in Equation 19 we can define an N-point DFT as

$$\begin{aligned} X(k) &= X_{DCT}(k) - jX_{DST}(k) \\ X(N-k) &= X_{DCT}(k) + jX_{DST}(k) \end{aligned} \qquad ,k = 1, 2, \ldots, N/2 - 1 . \tag{24}$$

In order to derive a recursive formulation of DCT and DST computations, we define a new sequence, x_e as

$$x_e(k) = x(k) + x(N-k) \qquad ,k = 1, 2, \ldots, N/2 - 1 . \tag{25}$$

Also, the $N/2^{th}$ point of this sequence is the same as that of the original sequence. Thus we can formulate the recursive DCT for the even numbered points as

$$DCT(2k, N+1, x) = DCT\left(k, \frac{N}{2} + 1, x_e\right) \qquad ,k = 1, 2, \ldots, N/2 - 1 , \text{ and} \tag{26}$$

$$\begin{aligned} X(0) &= X_{DCT}(0) \\ X(N/2) &= X_{DCT}(N/2) \end{aligned} . \tag{27}$$

We can define a recursive equation for the odd DCT points using a new sequence x_o defined as

$$x_o(k) = \frac{x(k) - x(N-k)}{2\cos(\pi k/N)} \qquad ,k = 1, 2, \ldots, N/2 - 1 . \tag{28}$$

Then,

$$DCT(2k+1, N+1, x) = DCT\left(k, \frac{N}{2} + 1, x_o\right) + DCT\left(k+1, \frac{N}{2} + 1, x_o\right), \tag{29}$$

where $k = 1, 2, \ldots, N/2 - 1$. A similar recursive formulation can be derived for the DST using symmetry properties of the sine function which results in

$$\begin{aligned} DST(2k, N-1, x) &= DST\left(k, \frac{N}{2} - 1, x_o\right) \\ DST(2k+1, N-1, x) &= DST\left(k, \frac{N}{2} - 1, x_e\right) + DST\left(k+1, \frac{N}{2} - 1, x_e\right) \end{aligned} , \tag{30}$$

where $k = 1, 2, \ldots, N/2 - 1$. Since the complex operations occur only in the last stage of the computation where the DCT and DST are combined using Equation 24, the QFT is well suited for operation on real data. The number of operations required to perform an N-point QFT is $11N/2 \cdot \log N - 27N/4 + 2$ [7]. This, however, does not include the cost of computing the odd and even parts of the data sequence at each stage of the computation.

2.6. Decimation-In-Time-Frequency (DITF) Algorithm

The DITF algorithm is based on the observation that in a DIF implementation of a RAD2 algorithm, most of the computations (especially complex multiplications) are performed during the initial stages of the algorithm. In the DIT implementation of the RAD2 algorithm, the computations are concentrated towards the final stages of the algorithm. Thus, starting with the DIT implementation and then shifting to the DIF implementation at some transition stage intuitively seems to be a computation saving process.

Equations (3) and (4) define the DIF RAD2 computation. The DIT RAD2 computation is defined as

$$X(k) = \sum_{n=0}^{N/2-1} x(2n) \cdot W_{N/2}^{kn} + \left(\sum_{n=0}^{N/2-1} x(2n+1) \cdot W_{N/2}^{kn} \right) \cdot W_N^k . \quad (31)$$

Note that the first summation in the above equation is the $N/2$-point DFT of the sequence comprised of the even-numbered points of the original sequence and the second summation is the $N/2$-point DFT of the sequence comprised of the odd-numbered points of the original sequence. The transition stage consists of a conversion from the DIT coefficients to the DIF coefficients,

$$DIF(k) = W_N^{pq} \cdot DIT(k), \quad (32)$$

where p is the index of the set to which k belongs and q is the position of k in that set. The indices of each set need to be bit-reversed. The total number of real multiplications involved in the DITF computation is $2N \cdot \log N - 10N + 8N/2^s + 8 \cdot 2^s - 8$, where s is the transition stage. On minimizing this

expression, we get the optimal transition stage for minimum number of multiplications as $\frac{\log N}{2}$.

3. BENCHMARKING CRITERIA

Most preceding FFT complexity studies have been conducted on special purpose hardware such as digital signal processing (DSP) chips [9,10]. Typically, the primary benchmarking criteria have been the number of mathematical operations (multiplications and additions) and/or the overall computation speed. Since a large portion of the DSP application market has transitioned to general purpose computers, benchmarks for these CPUs have become increasingly important. It has been a traditional belief that the efficiency of an algorithm is most influenced by the arithmetic complexity, usually expressed in terms of a count of real multiplications and additions. However, on general purpose computers this is not a very good benchmark and other factors need to be considered as well. For instance, the issue of memory usage is very important for memory constrained applications. Similarly, compiler optimizations play an important part in the execution speed of algorithms. Many modern benchmarks now include compiler effect characterization to study algorithm performances [17].

3.1. Number of Computations

Since many general purposes CPUs have significantly different speeds on floating point and integer operations, we decided to individually account for floating point and integer arithmetic. It is a well known fact that most new architectures compute floating point operations more efficiently than integer operations (ten years ago, this was not the case)[25,26]. Also, most indexing and loop control is done using integer arithmetic. Therefore the integer operations count directly measures the cost of indexing and loop control. Many FFT algorithms require a large number of division-by-two operations which is efficiently accomplished by using a binary shift operator. To account for this common operation, we include a count of binary shifts in our benchmarks.

3.2. Computation Speed

In most present-day applications for general purpose computers, with easy availability of faster CPUs and memory not being a primary constraint, the fastest algorithm is by far treated as the best algorithm. Thus, a

common choice to rank order algorithms is by their computation speed. Since the computation time for most lower order FFTs is less than a millisecond, transients in the measuring process could skew the data considerably. Typically, these anomalies are a result of variations in the loading time of CPUs. A common strategy employed to avoid this is to measure time over a number of iterations of the algorithm. Previous benchmarking efforts have shown that the median time taken over a number of iterations is a better measure than the timing of one iteration. Using this iterative approach helps us deal with the measurement noise and transients more effectively. Therefore we report median CPU time in our benchmarks.

3.3. Memory Usage

One of the classic trade-offs seen in algorithm development is that of memory usage versus speed. In most portable signal processing applications, the FFT is a core computational component. However, few applications can afford a large memory space for evaluating FFTs. Constraints on run-time memory resources coupled with static memory in the form of code space define the application's memory requirements. While memory usage is important for specification of hardware, memory accesses also account for a significant portion of computation time. This is attributed to cache misses, swapping and other paging effects. These effects are more prominent when computing higher order FFTs (typically over 4K points). The swap space required in such cases exceeds the cache size. These observations prompted us to include memory usage as one of the yardsticks in judging the effectiveness of the various FFT algorithms.

3.4. Compiler Optimizations:

With advances in compiler technology, compiler optimizations can now result in computational speed-ups as high as 300%. Some of the important optimizations that modern compilers try to achieve are:

1. *Tail recursion elimination* — converts self-recursive procedures into iterative procedures, saving manipulation time

2. *Loop-invariant code motion* — locates and removes computations that yield the same result

3. *Profiling* — allows optimizations to adapt themselves to program behavior

4. *Induction-variable strength reduction* — replaces slower operations by faster ones

5. *Loop unrolling* — reduces run-time by reducing loop overhead and increasing opportunities for more efficient instruction pipelining

6. *Loop inversion* — convert pre-test loops into post-test loops, reduce the number of branches required per iteration

Loop-invariant code motion and loop-unrolling are the most effective in optimizing FFT algorithms because the nature of the problem requires loop cores with intermixed additions and multiplications. Such computations provide for significant pipelining opportunities which is by far the most important issue in iterative, computation-intensive algorithms. To quantify these effects, we evaluate the performance of the FFT algorithms using two popular compilers for C++, **MSVC++** [13] and GNU's **gcc [**14]**.**

4. BENCHMARKING RESULTS AND ANALYSIS

Each of the algorithms was implemented under a common framework using common functions for operations such as bit-reversal and lookup table generation so that differences in performance could be attributed solely to the efficiency of the algorithms. Following this, we comprehensively benchmarked each algorithm according to the criteria discussed in the previous section. In the process we observed several results that were contrary to published theory. Many of these contradictions can be attributed to compiler optimizations rather than discrepancies in the algorithms or in the measurement process, as some algorithm implementations tended to be more amenable to optimizations than others.

4.1. Computation Speed

Computation speed is typically the most prominent aspect of an FFT algorithm in current DSP applications. Apart from the direct calibration of algorithms, we also decided to study the effect of the computer architecture on the execution speed of these algorithms. It is interesting to note how the performance of the algorithms scales in terms of processor speed. This is very heavily influenced by the architecture, cache usage, and other hardware-related features.

In many applications, the designer does not know in advance the type of architecture to be used. In such cases, information related to the performance of the FFT algorithms pertaining to the relevant CPU architecture can be effectively used to select the optimal algorithm for that application. Thus, a comparative benchmark across various hardware platforms is quite useful for a DSP application developer. Our evaluations were performed on three of the most commonly used general purpose processors: the Sun UltraSparc (170 MHz), the DEC Alpha 21164 (300 MHz) and the Intel Pentium Pro (200 MHz). The properties of these processors are summarized in Table 1. The computation speed of an algorithm for large data sizes can often be heavily dependent on the clock speed, RAM size, cache size and the operating system. Hence, these factors must be taken into account.

We evaluated each of the algorithms using the same compiler (GNU gcc v2.7.2.1). For the 200 MHz Pentium Pro machine, the computation time of the worst algorithm (DITF) is more than three times greater than that of the best algorithm (FHT). It has been consistently observed in our benchmarks that the FHT is the most efficient algorithm in terms of computation speed. Table 2 shows the variation in performance of these algorithms as a function of the FFT order for the Pentium Pro architecture.

The relative ranking of other algorithms does, however, change when benchmarked on a different machine. For example when evaluated on an UltraSparc2, the RAD4 algorithm performs better than a QFT. This is possibly attributed to the paging mechanism and cache usage on the different architectures and the degree to which the algorithms are susceptible to these factors.

A comparison of the computation time for the fastest algorithm, the FHT, on the three platforms is shown in Figure 3. The performance across the machines is clearly affected by the amount of RAM and cache. As expected, the effect is more pronounced for higher order FFTs where cache misses become common. We observe that the performance of the Pentium Pro does not scale up as gracefully as the DEC Alpha and the Sun UltraSparc. For the same processor speed the UltraSparc performs better than the Pentium Pro. One plausible explanation for this is the cache size. The cache on the Pentium Pro is one-fourth the size of that on the UltraSparc. This effect should, however, not play a significant role for lower order FFT computations. The performance of the CPUs on floating point operations versus integer operations is

significant as well. The Pentium Pro performs better than the UltraSparc on algorithms with a higher number of integer operations compared to the floating point operations. On the other hand, the UltraSparc consistently outperforms the Pentium Pro on algorithms with a higher number of floating point operations.

4.2. Number of Computations

The number of arithmetic computations has been the traditional measure of algorithmic efficiency. With the advent of on-chip arithmetic units, the relative importance of this figure of merit has dwindled. As our results suggest, the number of computations derived from the typical butterfly diagrams for FFTs no longer reflect the real execution cost when a designer exploits the efficiencies inherent in the implementation to avoid some redundant computations.

We have generated conclusive numbers for comparisons based on arithmetic computations. The number of operations required by each algorithm for a 1024-point real DFT are displayed in Table 3. We observe that the faster algorithms require performing a smaller number of computations. However, there is a trade-off between integer operations and floating point operations. Savings in floating point operations can be achieved at the cost of increasing the number of integer operations. This translates to achieving lower numbers of computations at the cost of more indexing operations. An example of this is seen in the excessive number of integer additions in the QFT. Most of the integer arithmetic is accounted for by loop control or re-indexing. In the QFT implementation, the DCT and DST recursions are implemented by accessing pointers in a common workspace. This results in the large number of integer operations. The large number of operations for the DITF algorithm are attributed to the bit-reversal process at various stages of the computation. This aspect seems to have been overlooked in previous evaluations [7,8]. Overall, the FHT and the SRFFT are the best in terms of effectively using computations, which translates to greater computation speed.

One should, however, not be blindly swayed by the performance of FHT. The main drawback of the FHT is that the complex FHT is computed via two real FHT computations. The QFT also uses a similar methodology. The number of computations doubles when moving from real data to complex data using these algorithms. The corresponding change for the other algorithms is insignificant.

4.3. Compiler Effects

Compiler technology has advanced greatly over the past decade. Earlier benchmarks were not as sensitive to compiler optimizations. In contrast, our preliminary tests revealed that variations in the compiler (gcc) optimization level improved the computation speed by as much as 300%. This prompted a closer exploration of this issue. Also, an application developer could look at benchmarks performed using code compiled on gcc and assume the effects would translate smoothly to code compiled using MSVC++. Unfortunately, this is not necessarily true. Figure 4 demonstrates this by plotting the difference in computation speeds between SRFFT and FHT on real data when compiled using gcc and MSVC++ on the Pentium Pro architecture. The compiler effects are not found to be uniform across algorithms, and there is not enough evidence gained from these numbers to help us predict the relationship between the compilers and the algorithms. One possible method to gain a better insight into the compiler effects is to incorporate a cache model into the software and trace the profile of the code. These issues highlight the need for a closer study of compiler effects on algorithms.

Table 5 shows the effect of different levels of optimization on the algorithm implementations when compiled using gcc. Level 2 performs all the optimizations described in Section 3.4, which do not involve time-space trade-offs. However, the compiler does not perform loop unrolling or function in-lining. Level 3 is a superset of level 2, and additionally turns on function in-lining. The SRFFT and RAD4 algorithms seem to benefit from this more than the other algorithms.

4.4. Memory Usage and Object Code Size

One of the key issues in portable applications is memory usage. Quantification of memory requirements is glaringly missing from most benchmarks published for FFT algorithms. In our work, memory usage also includes the input and output data arrays, lookup tables and any intermediate swap space used by the algorithm. Since we wanted to keep the structure of algorithms uniform, we have implemented all algorithms with lookup tables. Thus, any difference in memory usage can be attributed to variations in the actual swap space usage. When implemented in a uniform framework, the object code size and executable size are also direct measures of the complexity of an algorithm. Most of the faster algorithms have a

comparatively large object code size. Table 4 shows the memory usage profile of different algorithms for a 1024 point FFT on a Pentium Pro compiled using gcc.

We see from Table 4 that the RAD2 algorithm is the most memory efficient algorithm, and the QFT is the least. In the case of the QFT, this is due to the large work space required to perform the recursions in the DCT and the DST algorithms. The FHT is the most inefficient in terms of the executable size. Notice that, as was expected, the executable size is a good measure of the complexity of the algorithm with the FHT being the most complex and the RAD2 the least complex algorithm.

5. CONCLUSIONS

The existence of an abundance of algorithms for FFT computations and an even greater number of their implementations calls for a comprehensive benchmark which teases out the implementation-specific differences and compares the algorithms directly. We have tried to achieve this objective by implementing algorithms in a very consistent framework. Our results indicate that the overall best algorithm for DFT computations is the FHT algorithm. This has been, and will likely continue to be, a point of argument for many years [17,18,20,22,23]. Another feature in favor of the FHT is its bilateral formulation. Unlike DFT algorithms, FHT has the same functional form for both its forward and inverse transforms.

Our work is one of the first efforts to characterize FFT algorithms in terms of memory requirements. The FHT is the fastest algorithm on all platforms with a reasonable dynamic memory requirement. However, it is the most inefficient in terms of static memory usage (measured in terms of the executable size). If an FFT algorithm needs to be chosen solely on the basis of static memory requirements, the RAD2 algorithm is the still the best, owing to its simple implementation. The SRFFT and the FHT are comparable in terms of the number of computations and are the most efficient. In the existing studies, the overhead involved in the computations used for the QFT and the DITF algorithms — especially array indexing and data preparation such as the computation of even and odd components of a data set — is often neglected. Our benchmarks account for all such memory requirements as well.

For the same processor speed and RAM size, the Sun UltraSparc outperforms the Intel Pentium Pro for

algorithms with greater number of floating point operations than integer operations (e.g. FHT); while the Pentium Pro does better on algorithms with a greater number of integer operations (e.g. QFT). This trend is attributed to the performance of the CPUs on floating point operations vs. integer operations. Typically the Pentium class of CPUs perform worse than the Sparcs on floating point operations. This observation is validated by other benchmarks where floating point and integer specifications, SPECfp95 and SPECint95, of various CPUs are compared[25,26]. Table 6 shows the performance specifications for the Pentium, UltraSparc and the DEC Alpha.

As noted earlier, compiler effects were quite significant, though the underlying phenomena could not be explained directly by the algorithm implementations. The results of our benchmarks suggest the need for a cache model in our code design to bring insights into the cache-related and compiler-related issues.

6. ACKNOWLEDGEMENTS

This work was supported by DARPA through US Air Force's Rome Laboratories under contract F30602-96-1-0329. We would like to express our appreciation to Mr. Shane Hebert and Mr. Gerhard Lehnerer of HPCL, Department of Computer Science, Mississippi State University, for their help in running these benchmarks.

7. REFERENCES

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

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

[3] R.N. Bracewell, "Fast Hartley Transform", *Proceedings of IEEE*, pp. 1010-1018, 1984.

[4] H.S. Hou, "The Fast Hartley Transform Algorithm", *IEEE Transactions on Computers*, pp. 147-155, February 1987.

[5] P. Duhamel and H. Hollomann, "Split Radix FFT Algorithm," *Electronic Letters*, vol. 20, pp. 14-16, January 1984.

[6] H. Guo, G.A. Sitton, and C.S. Burrus, "The Quick Discrete Fourier Transform," *Proceedings of International Conference on Acoustics, Speech and Signal Processing*, vol. 3, pp. 445-447, Adelaide, Australia, April 1994.

[7] A. Saidi, "Decimation-In-Time-Frequency FFT Algorithm," *Proceedings of International Conference on Acoustics, Speech and Signal Processing*, vol. 3, pp. 453-456, Adelaide, Australia, April 1994.

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

[9] http://www.ti.com/sc/docs/dsps/literatu.htm

[10] http://www.lsi-dsp.com/c6x/tech/wpsynop.htm

[11] J.G. Proakis, D.G. Manolakis, *Digital Signal Processing - Principles, Algorithms and Applications*, Macmillan Publishing Company, NY, USA, 1992.

[12] A.V. Oppenheim, R.W. Schafer, *Digital Signal Processing*, Prentice-Hall International Inc., Englewood Cliffs, NJ, USA, 1989.

[13] http://www.microsoft.com/visualc

[14] http://www.maths.lancs.ac.uk/~smithdm1/GNU/GNUWeb/documentation.html

[15] C.V. Loan, *Frontiers in Applied Mathematics - Computational Frameworks for the Fast Fourier Transform*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA., 1992.

[16] R.N. Bracewell, "Assessing the Hartley Transform," *IEEE Transactions on Acoustics Speech and Signal Processing*, pp. 2174-2176, December 1990.

[17] M. Popovic, D. Sevic, "A new look at the Comparison of Fast Hartley and Fourier Transforms,"

IEEE Transactions on Signal Processing, vol. 42, pp. 2178-2182, August 1994.

[18] P.R. Uniyal, "Transforming Real-Valued Sequences: Fast Fourier versus Fast Hartley Transform Algorithms," *IEEE Transactions on Signal Processing*, vol. 42, pp. 3249-3253, November 1994.

[19] M.A. Mehalic, P.L. Rustan, and G.P. Route, "Effects of Architecture Implementation of DFT Algorithm Performance," *IEEE Transactions on Acoustics Speech and Signal Processing*, pp. 684-693, June 1985.

[20] P. Duhamel and M. Vetterli, "Improved Fourier and Hartley Transform Algorithms: Application to Cyclic Convolution of Real Data," *IEEE Transactions on Acoustics Speech and Signal Processing*, pp. 818-824, June 1987

[21] H.V. Sorensen, D.L. Jones, M.T. Hiedeman, and C.S. Burrus, "Real-valued fast Fourier transform Algorithms," *IEEE Transactions on Acoustics Speech and Signal Processing*, pp. 849-863, June 1987.

[22] H.V. Sorensen, D.L. Jones, M.T. Hiedeman, and C.S. Burrus, "On Computing the Discrete Hartley Transform," *IEEE Transactions on Acoustics Speech and Signal Processing*, pp. 1231-1238, October 1985.

[23] R.D. Preuss, "Very Fast Computation of the Radix-2, Discrete Fourier Transform," *IEEE Transactions on Acoustics Speech and Signal Processing*, pp. 595-607, March 1982.

[24] http://theory.lcs.mit.edu/~fftw

[25] http://www.contrib.andrew.cmu.edu/usr/sdavis/processor.html

[26] http://www.digital.com/semiconductor/micro-rpt-21164.htm

List of Tables

List of Figures

Processor type	Hardware Properties			
	Speed	RAM size	Cache size	Operating System
Sun UltraSparc	200MHz	256kB	512KB	Sun Solaris 2.5
DEC Alpha 21164	300MHz	128kB	2MB	Windows NT
Intel Pentium Pro	200MHz	256kB	256KB	Sun Solaris 2.5

Table 1: Properties of the general purpose CPUs used for benchmarking the FFT algorithms

Algorithm	FFT Order					
	16	**64**	**256**	**1024**	**4096**	**16384**
RAD2	20	60	260	1960	6800	30500
RAD4	20	60	300	1800	6940	29000
SRFFT	20	40	140	660	3700	17260
FHT	**20**	**40**	**120**	**560**	**3240**	**14020**
QFT	20	40	180	1020	5460	27760
DITF	20	80	380	1780	8780	40200

Table 2: Computation time (in microseconds) of various algorithms in computing several orders of FFT

Algorithm	Float Adds	Float Mults	Integer Adds	Integer Mults	Binary Shifts
RAD2	14336	20480	19450	2084	1023
RAD4	8960	14336	12902	3071	277
SRFFT	**5861**	**5522**	12664	2542	1988
FHT	7420	8841	**3235**	2048	**12**
QFT	9026	2560	29784	**1048**	144
DITF	14400	17664	20333	1076	1074

Table 3: Number of computations involved in computing a 1024-point FFT

Algorithm	Memory Usage (Bytes)	Object Code (Bytes)
RAD2	**72440**	**5190**
RAD4	72536	5293
SRFFT	72508	6275
FHT	72652	11506
QFT	122072	9800
DITF	78632	8691

Table 4: Memory usage and object code size in computing a 1024-point FFT

Algorithm	Optimization	
	Level 2	Level 3
RAD2	1.0	1.0
RAD4	9.2	7.9
SRFFT	13.2	13.2
FHT	3.0	3.0
QFT	2.0	0.0
DITF	2.3	3.1

Table 5: Percent change in computation time of algorithms for a 1024-point FFT compared to level 1 optimization in gcc

Specification	DEC Alpha 21164	UltraSparc	Pentium Pro
SPECfp95/MHz	0.0421	0.0468	0.0240
SPECint95/MHz	0.0309	0.0384	0.0387

Table 6: Floating point and integer performance specifications for the various CPUs

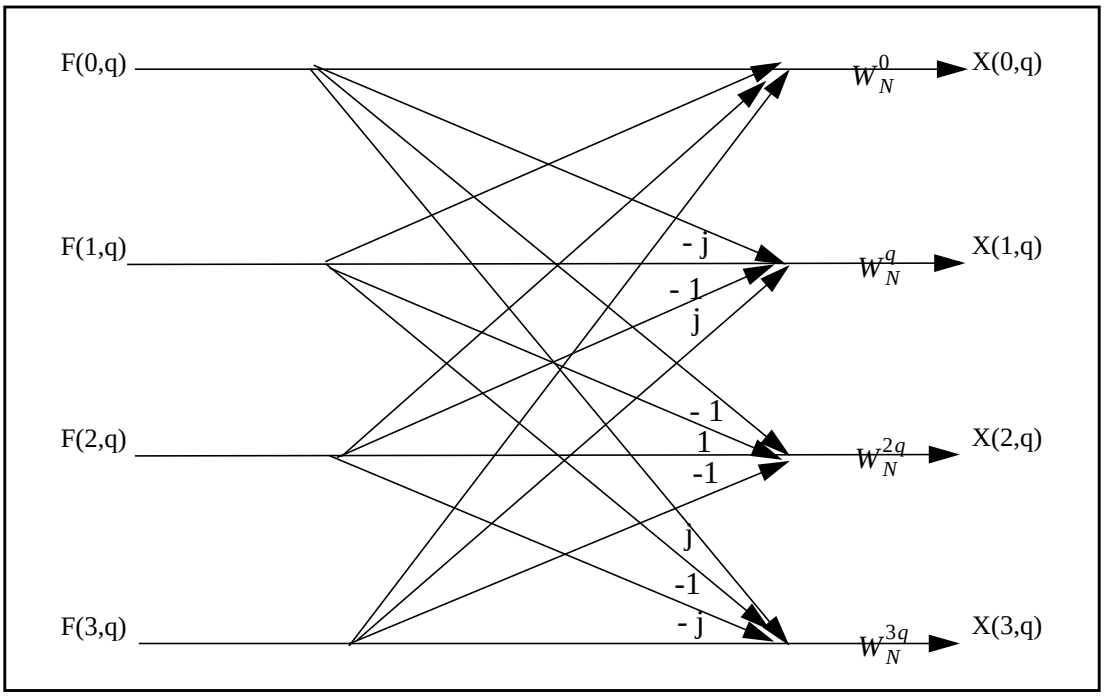

Figure 1. Radix-4 butterfly involving 3 complex multiplications and 12 complex additions

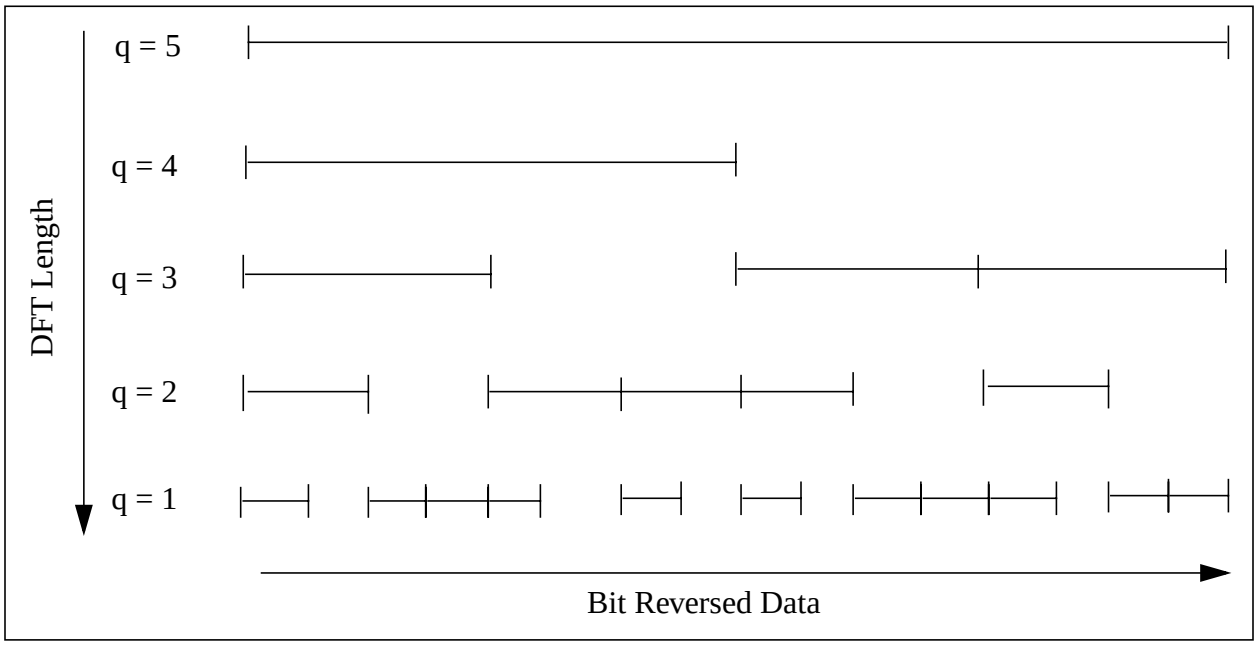

Figure 2. Split-Radix computational framework

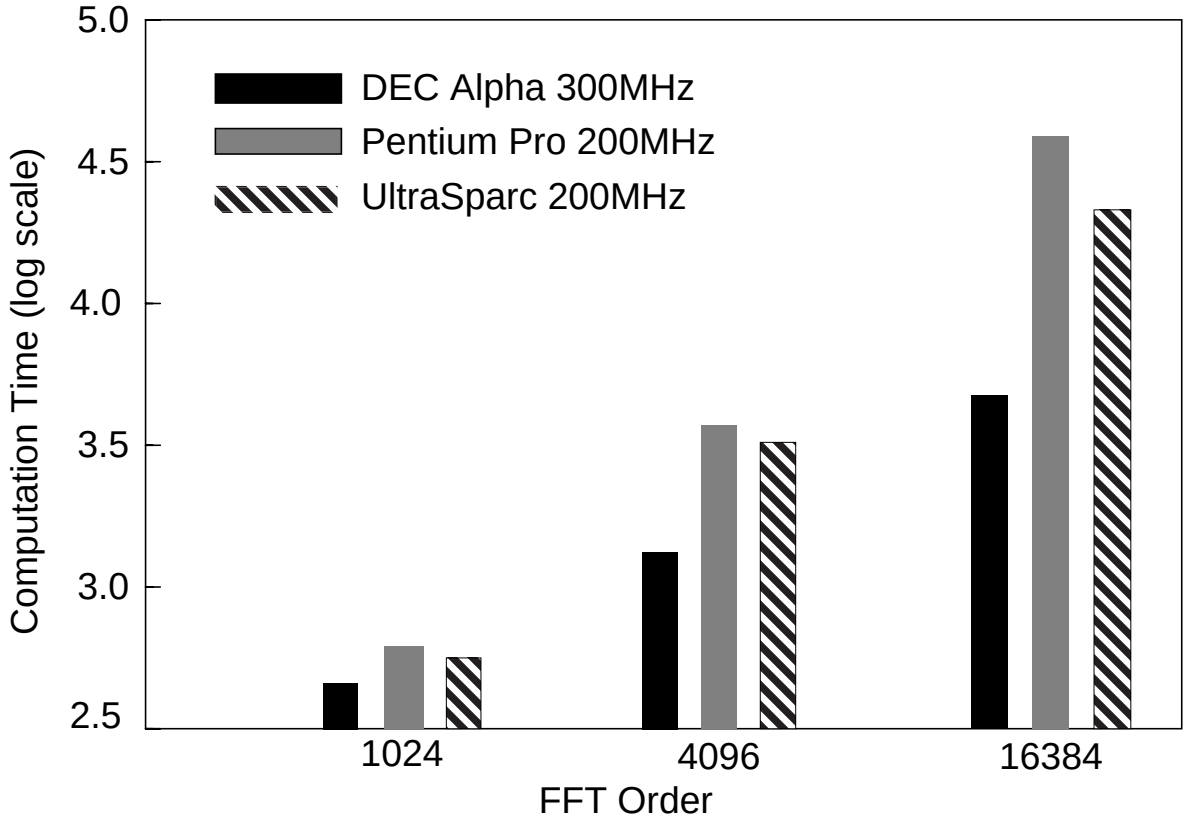

Figure 3. Comparison of computation speed across three different CPUs for the FHT algorithm

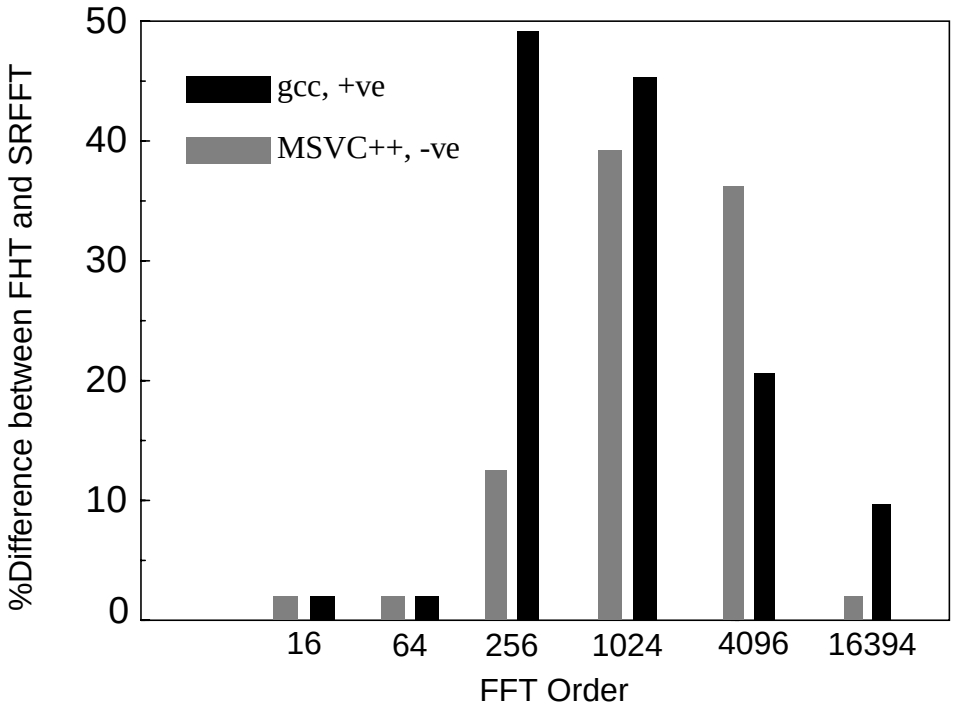

Figure 4. Difference in computation time of FHT and SRFFT compiled using gcc and MSVC++