AN ADAPTIVE SOFTWARE LIBRARY FOR
FAST FOURIER TRANSFORMS
ON REAL INPUT DATA

A Thesis
Presented to
the Faculty of the Department of Computer Science
University of Houston

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

By
Fredrick Mwandia
November, 2000
AN ADAPTIVE SOFTWARE LIBRARY FOR
FAST FOURIER TRANSFORMS
ON REAL INPUT DATA

Fredrick Mwandia

APPROVED:

Dr. S. Lennart Johnsson
Department of Computer Science

Dr. Dragan Mirkovic
Department of Computer Science

Dr. Barbara Chapman
Department of Computer Science

Dean, College of Natural Sciences and Mathematics
Acknowledgements

I would like to give my heart felt gratitude to my advisor, Dr. Lennart Johnsson, whose leadership and inspiration was invaluable in guiding me through this work. My appreciation also goes out to Dr. Dragan Mirković who worked tirelessly to ensure the success of this work.

I would also like to thank Dr. Barbara Chapman, for her insite as she reviewed my work as a member of my committee. I am greatfull for all the support that I recieved from the San Diego Supercomputing Center (through NPACI) and NCSA in affording us an opportunity to test our work on their clusters.

Above all, I would like to thank God for giving me the strength and sustaining me through this time.
AN ADAPTIVE SOFTWARE LIBRARY FOR
FAST FOURIER TRANSFORMS
ON REAL INPUT DATA

An Abstract of a Thesis
Presented to
the Faculty of the Department of Computer Science
University of Houston

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

By
Fredrick Mwandia
November, 2000
Abstract

In this thesis, we present an efficient methodology for architecture adaptable generation of code that can be constructed of components with well defined algebraic rules of composition. The code generator is written in C and generates a library of codelets in C. The code generator is shown to be flexible and extensible and the entire library can be generated in a matter of seconds.

We also present a software library for the Fast Fourier Transform (FFT) on real input data and show the results of adaptivity to different hardware architectures. The library consists of a number of composable blocks of code called codelets, each computing a part of the transform. The actual FFT algorithm used by the code is determined at run-time by selecting the fastest strategy among all possible strategies, given available codelets, for a given transform size and the distribution of data. The library takes into account the symmetries that occur when real input data is used and it takes advantage of these symmetries to produce more efficient codelets. We have evaluated the library for performance on the IBM–Power3, SGI Origin–2000, and Intel Pentium–III systems. The library is shown to be portable, adaptive and flexible.

The optimization of the FFT library is performed on two levels. The low level optimization involves generation of highly efficient codelets for small transform sizes. These codelets are optimized for the number of arithmetic operations and memory accesses on each platform and are shown to perform well on all architectures. The high level optimization involves selection of an execution plan for different transform sizes. Large size transforms are created by using a computationally efficient combination of the codelets in the library. Performance data for both cases is presented.
## Contents

1 **Introduction**  
  1.1 History of FFT Algorithms .................................................. 2  
  1.2 Related Work ................................................................. 4  
  1.3 Motivation for this Work ..................................................... 7  
  1.4 Organization of the Thesis .................................................... 8  

2 **Mathematical Background** .......................................................... 9  
  2.1 Introduction ........................................................................ 10  
    2.1.1 Notation ....................................................................... 11  
  2.2 Properties of the DFT Matrix .................................................. 12  
  2.3 The Cooley-Tukey Factorization .............................................. 14  
  2.4 Sparse factorizations ................................................................ 16  
    2.4.1 Prime Factor Algorithms ............................................... 17  
    2.4.2 Rader’s Algorithm ......................................................... 18  
    2.4.3 Split–Radix Algorithm .................................................... 22  
  2.5 Properties Resulting from Real Input Data  
    2.5.1 The Conjugate–Even Butterfly ....................................... 28  
    2.5.2 The Split–Radix Algorithm for Real Data  

3 **Methodology** .................................................................................... 37  
  3.1 Approach to the Problem ......................................................... 37  
  3.2 Execution Plan Generation ....................................................... 39  
  3.3 Library of FFT Codelets ............................................................. 42  
  3.4 Description of the Code Generator ............................................ 47  
    3.4.1 Basic Structures ............................................................ 47  
    3.4.2 Basic Operators ............................................................ 49  
    3.4.3 Derived Structures and Operations ................................... 50  
    3.4.4 FFT Operations ............................................................. 51

vii
4 Accuracy Tests and Performance Analysis 54
  4.1 Accuracy Tests .................................................. 55
  4.2 Target Hardware Architectures .................................. 57
     4.2.1 The SGI R10000 Processor ................................ 57
     4.2.2 The IBM Power3 Processor ............................... 58
     4.2.3 The Intel Pentium-III Processor ......................... 59
  4.3 Analysis of FFT Codelet Algorithms ........................... 60
  4.4 Compiler Analysis ................................................. 64
     4.4.1 Compiler Performance Measures .......................... 64
     4.4.2 Compiler Performance Results ............................ 66
  4.5 Performance of the FFT Codelets ............................... 74
     4.5.1 The SGI R10000 ........................................... 75
     4.5.2 The Intel Pentium ......................................... 83
     4.5.3 The IBM Power3 ......................................... 98
     4.5.4 UHRFFT Codelet Efficiency comparison with UHFFT .... 102
  4.6 Analysis of Execution Strategy ................................. 105
     4.6.1 The SGI R10000 ........................................... 106
     4.6.2 The IBM Power3 ......................................... 113
     4.6.3 The Intel Pentium-III 500 ............................... 119
     4.6.4 The Intel Pentium-III 550 Xeon ......................... 126

5 Interfaces and Library Packaging 132
  5.1 Motivation ....................................................... 132
  5.2 UHRFFT’s API ................................................... 134
  5.3 Usage ........................................................... 139
  5.4 Installation ..................................................... 141

6 Conclusion and Future Work 144
  6.1 Summary ........................................................ 144
  6.2 Future Work ..................................................... 145

References 147
List of Figures

3.1 UHFFT Library Organization. ........................................ 46
4.1 Option Performance Index \( IO_k \) on the Intel Pentium III’s. 67
4.2 Performance for 2-Point Transform. ............................... 67
4.3 Performance for 4-Point Transform. ............................... 68
4.4 Performance for 8-Point Transform. ............................... 68
4.5 Performance for 16-Point Transform. ............................. 69
4.6 Performance for 32-Point Transform. ............................. 69
4.7 Performance for 64-Point Transform. ............................. 70
4.8 Performance for 128-Point Transform. ............................ 70
4.9 Option Performance Index \( IO_k \) on the SGI Origin 2000... 71
4.10 Option Performance Index \( IO_k \) on the IBM Power3 Processor. 73
4.11 Performance for Codelets of Size 2 and Size 3 on a 250 MHz SGI R10000. 76
4.12 Performance for Codelets of Size 4 and Size 5 on a 250 MHz SGI R10000. 76
4.13 Performance for Codelets of Size 6 and Size 7 on a 250 MHz SGI R10000. 77
4.14 Performance for Codelets of Size 8 and Size 9 on a 250 MHz SGI R10000. 77
4.15 Performance for Codelets of Size 10 and Size 11 on a 250 MHz SGI R10000. 78
4.16 Performance for Codelets of Size 12 and Size 13 on a 250 MHz SGI R10000. 78
4.17 Performance for Codelets of Size 14 and Size 16 on a 250 MHz SGI R10000. 79
4.18 Performance for Codelets of Size 17 and Size 19 on a 250 MHz SGI R10000. 79
4.19 Performance for Codelets of Size 32 and Size 64 on a 250 MHz SGI R10000. 80
4.20 Performance for Codelet of Size 128 on a 250 MHz SGI R10000. 80
4.21 UHRFFT Codelet Efficiency on a 250 MHz SGI R10000. 81
4.22 UHRFFT Codelet Timings on a 250 MHz SGI R10000. 81
4.23 Performance for Codelets of Size 2 and Size 4 on a 500 MHz Pentium III. 84
4.24 Performance for Codelets of Size 8 and Size 16 on a 500 MHz Pentium III ......................................................... 84
4.25 Performance for Codelets of Size 32 and Size 64 on a 500 MHz Pentium III .......................................................... 85
4.26 Performance for Codelet of Size 128 on a 500 MHz Pentium III ................................................................. 85
4.27 UHRFFT Codelet Efficiency on a 500 MHz Pentium III ......................................................................................... 86
4.28 UHRFFT Codelet Timings on a 500 MHz Pentium III ...................................................................................... 86
4.29 Performance for Codelets of Size 2 and Size 4 on a 550 MHz Pentium III Xeon ........................................... 89
4.30 Performance for Codelets of Size 8 and Size 16 on a 550 MHz Pentium III Xeon ............................................. 89
4.31 Performance for Codelets of Size 32 and Size 64 on a 550 MHz Pentium III Xeon .............................................. 90
4.32 Performance for Codelet of Size 128 on a 550 MHz Pentium III Xeon .............................................................. 90
4.33 UHRFFT Codelet Efficiency on a 550 MHz Pentium III Xeon ................................................................. 91
4.34 UHRFFT Codelet Timings on a 550 MHz Pentium III Xeon ................................................................. 91
4.35 Performance for Codelets of Size 2 and Size 4 on a 600 MHz Pentium III ...................................................... 94
4.36 Performance for Codelets of Size 8 and Size 16 on a 600 MHz Pentium III ...................................................... 94
4.37 Performance for Codelets of Size 32 and Size 64 on a 600 MHz Pentium III ...................................................... 95
4.38 Performance for Codelet of Size 128 on a 600 MHz Pentium III .............................................................. 95
4.39 UHRFFT Codelet Efficiency on a 600 MHz Pentium III ................................................................. 96
4.40 UHRFFT Codelet Timings on a 600 MHz Pentium III ......................................................................................... 96
4.41 Performance for Codelets of Size 2 and Size 4 on a 222 MHz Power3 ...................................................... 98
4.42 Performance for Codelets of Size 8 and Size 16 on a 222 MHz Power3 ...................................................... 99
4.43 Performance for Codelets of Size 32 and Size 64 on a 222 MHz Power3 ...................................................... 99
4.44 Performance for Codelet of Size 128 on a 222 MHz Power3 .............................................................. 100
4.45 UHRFFT Codelet Efficiency on a 222 MHz Power3 .............................................................................. 100
4.46 UHRFFT Codelet Timings on a 222 MHz Power3 ....................................................................................... 102
4.47 UHRFFT vs. UHFFT Codelet Efficiency comparison on a 222 MHz Power3 ........................................... 104
4.48 UHRFFT vs. UHFFT Codelet Efficiency comparison on a 550 MHz Xeon Pentium III ........................................... 104
4.49 Execution Plan Performance for Size 8 and Size 16 FFTs on a 250 MHz SGI R10000 ........................................... 108
4.50 Execution Plan Performance for Size 32 and Size 64 FFTs on a 250 MHz SGI R10000 ........................................... 108
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.72</td>
<td>Execution Plan Performance for Size 512 and Size 1024 FFTs on a 500 MHz Pentium–III.</td>
<td>122</td>
</tr>
<tr>
<td>4.73</td>
<td>Execution Plan Performance for Size 2048 and Size 4096 FFTs on a 500 MHz Pentium–III.</td>
<td>123</td>
</tr>
<tr>
<td>4.74</td>
<td>Execution Plan Performance for Size 8192 and Size 16384 FFTs on a 500 MHz Pentium–III.</td>
<td>123</td>
</tr>
<tr>
<td>4.75</td>
<td>Execution Plan Performance for Size 32768 and Size 65536 FFTs on a 500 MHz Pentium–III.</td>
<td>124</td>
</tr>
<tr>
<td>4.76</td>
<td>Execution Plan Performance for Size 131072 and Size 262144 FFTs on a 500 MHz Pentium–III.</td>
<td>124</td>
</tr>
<tr>
<td>4.77</td>
<td>Execution Plan Performance for Size 524288 and Size 1048576 FFTs on a 500 MHz Pentium–III.</td>
<td>125</td>
</tr>
<tr>
<td>4.78</td>
<td>500 MHz Pentium–III Execution Plan Performance.</td>
<td>125</td>
</tr>
<tr>
<td>4.79</td>
<td>Execution Plan Performance for Size 8 and Size 16 FFTs on a 550 MHz Xeon.</td>
<td>126</td>
</tr>
<tr>
<td>4.80</td>
<td>Execution Plan Performance for Size 32 and Size 64 FFTs on a 550 MHz Xeon.</td>
<td>127</td>
</tr>
<tr>
<td>4.81</td>
<td>Execution Plan Performance for Size 128 and Size 256 FFTs on a 550 MHz Xeon.</td>
<td>128</td>
</tr>
<tr>
<td>4.82</td>
<td>Execution Plan Performance for Size 512 and Size 1024 FFTs on a 550 MHz Xeon.</td>
<td>128</td>
</tr>
<tr>
<td>4.83</td>
<td>Execution Plan Performance for Size 2048 and Size 4096 FFTs on a 550 MHz Xeon.</td>
<td>129</td>
</tr>
<tr>
<td>4.84</td>
<td>Execution Plan Performance for Size 8096 and Size 16384 FFTs on a 550 MHz Xeon.</td>
<td>129</td>
</tr>
<tr>
<td>4.85</td>
<td>Execution Plan Performance for Size 32768 and Size 65536 FFTs on a 550 MHz Xeon.</td>
<td>130</td>
</tr>
<tr>
<td>4.86</td>
<td>Execution Plan Performance for Size 131072 and Size 262144 FFTs on a 550 MHz Xeon.</td>
<td>130</td>
</tr>
<tr>
<td>4.87</td>
<td>Execution Plan Performance for Size 524288 and Size 1048576 FFTs on a 550 MHz Xeon.</td>
<td>131</td>
</tr>
<tr>
<td>4.88</td>
<td>550 MHz Xeon Execution Plan Performance.</td>
<td>131</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Comparison of Operation count Between Complex and Real FFT's 29
3.1 Number of Arithmetic Operations for a Codelet of Size 16 43
4.1 SGI R10000 Processor Characteristics 58
4.2 IBM Power3 Processor Characteristics 59
4.3 Intel Pentium-III Processor Characteristics 60
4.4 Arithmetic Operations Count for UHRFFT Generated Codelets (Forward) 61
4.5 Instruction and Cycle Counts for UHRFFT Generated Codelets on Intel Pentium-III 63
4.6 Compiler Performance on the SGI Origin 2000 72
4.7 Compiler Performance on the IBM NightHawk Nodes 73
4.8 Peak Performance of UHRFFT Codelets on a 250 MHz SGI R10000 82
4.9 Peak Performance of UHRFFT Codelets on a 500 MHz Intel Pentium III 87
4.10 Peak Performance of UHRFFT Codelets on a 550 MHz Intel Pentium III Xeon 92
4.11 Peak Performance of UHRFFT Codelets on a 600 MHz Intel Pentium III 97
4.12 Peak Performance of UHRFFT Codelets on a 222MHz Power3 101
4.13 Execution Plan Performance for Size 256 FFTs on a 250 MHz SGI R10000 107
4.14 Execution Plan Performance for Size 256 FFTs on a 222 MHz Power3 114
4.15 Execution Plan Performance for Size 256 FFTs on a 500 MHz Pentium-III 120
4.16 Execution Plan Performance for Size 256 FFTs on a 550 MHz Xeon 126
5.1 Compiler Flags for the UHRFFT on Some Architectures 142

xiii
Chapter 1

Introduction

The Fast Fourier Transform (FFT) is one of the most widely used algorithms in science and engineering. In fact, any field in physical science that utilises sinusoidal signals in theory makes use of the FFT. Some of the most common applications that make extensive use of FFTs are:

- Signal and Data processing/analysis
- Data compression
- Solution of Partial Differential Equations
- Seismic Data Analysis
- Image Processing

It’s importance and use in a vast range of applications has created a great demand for efficient implementations of the FFT algorithm on different hardware platforms.
1.1 History of FFT Algorithms

Considerable research efforts have been devoted to developing efficient Fourier transform algorithms over the past forty years but the history of the algorithm dates back to the early 19th century when Gauss described efficient algorithms for length $2^N$ transforms [1]. The turning point for Fourier transform applications came when Cooley and Tukey published their work [2] in 1965. The research that followed produced a number of important algorithms which focused on designs that either minimized the total number of arithmetic operations or the number of multiplications involved in the calculation of the transform by constant factors. The Cooley and Tukey algorithm (as described in [2]), reduced the number of arithmetic operations required for the Discrete Fourier Transform (DFT) from $O(N^2)$ to $O(N\log_2 N)$. Gentleman and Sande described a similar algorithm [4] in 1966. The algorithm described by C. M. Rader [5] in 1968 was designed for lengths of $N$ that are prime and involves the conversion of the given DFT to a convolution. In more recent years, the Split–Radix algorithm, designed for length $2^N$, was described by P. Duhamel and H. Hollmann [6]. The Prime Factor Algorithm (PFA), which was first described by Winograd [3], and more recently by C. Temperton [7, 8] and C. S. Burrus [9], is based on an index map originally developed by Thomas and Good [10, 11]. All these algorithms have an asymptotic complexity of $O(N\log_2 N)$.

The next step in the development of FFT algorithms was focused on the performance optimization of implementations on particular computer architectures. The initial development focused on the efficient use of arithmetic units, while later development focused on the efficient use of the memory hierarchy. Some of this work focused on reducing the number of multiplications even at the expense of increasing
the number of additions or subtractions. The justification was that multiplication is inherently a more complex operation than addition/subtraction. Today, modern processors are typically equipped with an equal number of units for multiplication and addition. Thus, with respect to the code execution environment, saving multiplications at the expense of increasing the number of additions, in general, does not improve the performance once the number of multiplications and additions are balanced. Moreover, many processor architectures place further restrictions on the ideal relationship between additions and multiplications by having floating-point units that can execute either a multiplication, an addition, or a multiplication of a pair of operands followed by the addition of the result to a third operand, a fused multiplication and addition, but not allowing for multiplication of two operands concurrently with the addition of a different pair of operands.

Current research efforts are focused on developing packages that are portable and adaptively tune themselves to the architectures on which they are installed. An early example of code adaptivity in scientific software packages is the Connection Machine Scientific Software Library (CMSSL) [12] in which adaptivity to problem size and data distribution was made by selecting at runtime among optimized codes for functions such as the BLAS (Basic Linear Algebra Subprograms) [13] and the FFT. Current state-of-the-art codes for the FFT adapt themselves to the computer architecture and transform size by using a dynamic construction of the FFT algorithm depending on the size of the transform, the memory architecture and the instruction set. The adaptability is accomplished by using a library of composable blocks of code, each computing a part of the transform, and by selecting the optimal combination of these blocks at runtime. The blocks of code, called codelets, are highly optimized and usually generated by a special-purpose compiler. Excellent examples that uses
this approach are the FFTW [14], developed at MIT and the FFT routines in the CWP numerical library developed by David Hale of the Colorado School of Mines. A similar approach has been adopted for the UHFFT library [15, 16]. In this thesis, we describe the optimization procedure for the real FFT library (hereafter referred to as UHRFFT), which is a part of the UHFFT. First, a brief overview of the efforts that have been directed towards the optimization of FFT codes is given.

1.2 Related Work

Ever since the FFT algorithm was first published by Cooley and Tukey [2] in 1965, a considerable research effort has been devoted to the problem of optimizing the FFT algorithm on different architectures. As a result, large number of FFT implementations are currently available. Due to the significance of the FFT in many applications, most systems have their own, often highly optimized implementation of the FFT. These implementations are marginally portable since the performance optimizations are highly tuned to the specific architectures on which they are implemented.

One of the early attempts to create a portable FFT code was done by P. N. Swartztrauber in FFTPACK [17]. FFTPACK contains Fortran code for one-dimensional complex–complex, real–complex, complex–real, sine, and cosine transforms. The routines in this package are non-adaptive, but the optimizations were good enough for the code to perform reasonably well on most platforms. This is because the codes were based on a fixed factorization of the DFT matrix that worked well on many platforms.

The Fortran FFT codes by C. Temperton [7], R. C. Singleton [18] and the Fortran Split–Radix FFT code by H. V. Sorensen [19] are other popular public domain codes
that are highly portable. The Temperton codes are designed for any powers of 2, 3, and 5, i.e., for $N = 2^p 3^q 5^r$ and uses the Prime Factor FFT algorithm [8, 20]. They perform both one and three-dimensional transforms. These codes were designed to perform well on vector architectures, and for many years were the fastest codes on such architectures. The Singleton code is based on the Mixed-Radix algorithm and handles both one and multi-dimensional complex transforms. The Sorensen code uses the Split-Radix algorithm to compute real and complex transforms for sizes that are powers of two. [6, 21].

The FFT optimization strategy has gone through many stages. The initial research efforts mentioned above, which were focused on the design of algorithms that minimized the number of arithmetic operations, resulted in a large number of very efficient FFT algorithms being developed over the years. Some of the most popular algorithms are:

- The Mixed-Radix algorithm,
- The Split-Radix algorithm,
- The Prime Factor algorithm,
- Rader’s algorithm.

The split-radix algorithm [6] yields a lower number of arithmetic operations than the Mixed-Radix algorithm [2] for transform sizes that are powers of 2, whereas the Prime Factor algorithm [8] is the most efficient for transform sizes that can be factored into mutually prime factors. Rader’s algorithm [5] is used for prime sized transforms. Since no single algorithm has proven to yield the lowest execution times for all architectures and all transform sizes, the efficient implementation of these
algorithms on modern microprocessor architectures is a challenge in itself. Hence, the current research focus is on adaptive algorithms that tune themselves to perform well on different architectures and for different sizes of transforms.

The main idea in this approach is to dynamically construct the factorization of the DFT matrix using a combination of one or more of the above mentioned algorithms depending on the size of the transform. This is accomplished by using a library of composable blocks of code, each computing a part of the transform, and by selecting the optimal combination of these blocks at runtime. One of the first public domain codes to use a similar adaptive approach to optimize the FFT over multiple platforms was the CWP numerical library by David Hale of the Center for Wave Phenomena at the Colorado School of Mines. This library has a fixed number of codelets and works only on a limited set of array sizes. In order to compute a desired transform, the CWP library changes the transform size to one that has the best performance in computing the desired transform size, if the desired size is one that is not supported. The library is based on DFT codelets based on the Prime Factor algorithm which was proposed by C. Temperton [20, 22]

The FFTW program [14], developed at MIT by Matteo Frigo and Steven G. Johnson is another example that uses this more elaborate adaptive approach. The codelets in this package are generated by a special-purpose compiler and perform well on most architectures. The FFTW library performs well for most sizes of transforms, but is most efficient for sizes of the form \( N = 2^e3^b5^c7^d11^e13^f \), where \( e + f \) is either 0 or 1, and other exponents are arbitrary. Other size transforms are computed by a slow, general purpose routine. The approach used to build the UHFFT library is very similar to the one used in FFTW.
1.3 Motivation for this Work

The main objective of this effort is to develop a portable and efficient FFT library suitable for parallel and distributed execution environments. Although there are many public domain codes for the FFT, many of them have serious limitations in terms of portability and flexibility. For these reasons, no good adaptable parallel FFT libraries exist. UHRFFT is designed to overcome these limitations. A good parallel library has to be based on an efficient serial library which in turn needs to be based on automatic generation of code for flexibility and adaptability. UHRFFT takes this adaptive approach to building a library of efficient algorithms for real input data. This work, therefore, builds the necessary infrastructure for the UHRFFT which includes a code generator and execution routines for real transforms.

The main advantage of this approach is the ease with which the library can be extended and improved when new low level optimizations are made or when new algorithms are added. The regeneration of the library is highly automated. Although FFTW uses this adaptive approach, it has certain shortcomings which give us motivation for our work on UHFFT. The UHFFT code generator is written in C, unlike FFTW which is written in Objective Caml, and this makes the UHFFT easier to modify and extend for the users of the libraries.

Another advantage of the approach used to build the UHRFFT library is that the code is generated (i.e codelets) on each installation unlike FFTW which uses pre-generated files. This allows for more architecture specific implementation and also enables installation tuning to determine which size codelets are needed for the specific architecture.
1.4 Organization of the Thesis

We first give a brief overview of the mathematical concepts behind the Discrete Fourier Transform in Chapter 2. The mathematical foundation for some of the popular FFT algorithms is given in this chapter with emphasis on FFT’s with real input data. Chapter 3 explains in detail the two-level optimization strategy of the library which enables the code produced by the generator to be adaptable over multiple hardware platforms. This chapter also give a brief description of the code generator explaining its most important features. In Chapter 4, we give a description of the hardware architecture that are used for the performance analysis and also describe the accuracy tests used to validate the correctness of the library of codelets. The bulk of the chapter is dedicated to the analysis of the the performance of the UHRFFT library. Results of both levels of optimization are presented. In Chapter 5, we describe the UHRFFT library interfaces and provide some details of how to use it to compute FFTs. A brief discussion is also given on how to install the library. The final chapter gives a summary of our work and gives suggestions of future work.
Chapter 2

Mathematical Background

The Fast Fourier Transform (FFT) is a method used for the fast evaluation of the Discrete Fourier Transform (DFT). The DFT is a matrix–vector product that requires $O(n^2)$ arithmetic operations to compute. Using the FFT to evaluate the DFT reduces the number of operations required to $O(n \log_2 n)$. For real data sequences, properties of the FFT allow a further reduction in work and storage by a factor of two. In this chapter, we

- introduce notation which is customary for describing FFT algorithms,
- give a brief mathematical background,
- describe the set of FFT algorithms used in the UHFFT and that are also used in UHRFFT,
- describe the algorithms used uniquely in UHRFFT.
2.1 Introduction

Let \( \mathbb{C}^n \) denote the vector space of complex \( n \)-vectors with components indexed from zero to \( n - 1 \). The Discrete Fourier Transform (DFT) of \( x \in \mathbb{C}^n \) is defined by

\[
X_l = \sum_{j=0}^{n-1} \omega_n^{lj} x_j, \quad l = 0, \ldots, n - 1,
\]

where

\[
\omega_n = e^{-2\pi i/n} = \cos(2\pi/n) - i\sin(2\pi/n),
\]

and \( i = \sqrt{-1} \). Note that \( \omega_n \) is an \( n \)th root of unity: \( \omega_n^n = 1 \).

The Inverse Discrete Fourier Transform (IDFT) is defined by

\[
x_j = \frac{1}{n} \sum_{l=0}^{n-1} \omega_n^{-lj} X_l.
\]

Both the forward and inverse DFTs constitute matrix–vector multiplications with special matrices called the DFT and inverse DFT matrix respectively. The matrix entries, or coefficients \( (\omega_n^{lj}) \) are known as twiddle factors. In matrix vector terms, the DFT is written as

\[
\begin{pmatrix}
X_0 \\
X_1 \\
X_2 \\
X_3 \\
\vdots \\
X_{n-1}
\end{pmatrix} =
\begin{pmatrix}
1 & 1 & 1 & 1 & \ldots & 1 \\
1 & \omega_n^1 & \omega_n^2 & \omega_n^3 & \ldots & \omega_n^{n-1} \\
1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \ldots & \omega_n^{2(n-1)} \\
1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \ldots & \omega_n^{3(n-1)} \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \omega_n^{(n-1)} & \omega_n^{(n-1)2} & \omega_n^{(n-1)3} & \ldots & \omega_n^{(n-1)(n-1)}
\end{pmatrix}
\begin{pmatrix}
x_0 \\
x_1 \\
x_2 \\
x_3 \\
\vdots \\
x_{n-1}
\end{pmatrix}
\]

or

\[
X = W_n x,
\]

where \( W_n \) is the DFT matrix.
2.1.1 Notation

All of the FFT algorithms are in essence sparse factorizations in which the action of the DFT matrix of size \( n \) on a vector \( \mathbf{x} \) is calculated through smaller DFT matrices acting on subvectors of \( \mathbf{x} \). A simple way to specify a subvector of a vector is by using a colon notation [30]. If \( \mathbf{x} \in \mathbb{C}^n \) then we can write a subvector \( \mathbf{y} \in \mathbb{C}^m, m < n \) consisting of components of \( \mathbf{x} \) beginning with the \( x_b \), ending with \( x_e \) and with a fixed stride \( s \) as \( \mathbf{y} = \mathbf{x}(b : s : e) \).

We will also use a special notation for block matrices and block-diagonal matrices. Let \( A_{ij} \) be an \( m \times n \) matrix. We denote by

\[
\text{block} (A_{ij}) = \begin{pmatrix}
A_{00} & A_{01} & \cdots & A_{0p} \\
A_{10} & A_{11} & \cdots & A_{1p} \\
\vdots & \vdots & \ddots & \vdots \\
A_{q0} & A_{q1} & \cdots & A_{pq}
\end{pmatrix}
\]

an \( mp \times nq \) block matrix. Similarly, if \( A_i \) is an \( m \times n \) matrix, we will denote by

\[
\text{diag} (A_i) = \begin{pmatrix}
A_0 \\
A_1 \\
\vdots \\
A_q
\end{pmatrix}
\]

an \( mq \times nq \) block diagonal matrix. In order to clearly explain the different factorization schemes used for the FFT we will use the tensor product notation, also known as the Kronecker product. The advantage of this notation is that it provides a more compact and precise description for operations on highly structured matrices. A detailed review of the Kronecker product is given in [23], [24] and [25].
**Definition 1** For $A \in \mathbb{C}^{p \times q}$ and $B \in \mathbb{C}^{k \times l}$ the Kronecker product, $A \otimes B \in \mathbb{C}^{pk \times ql}$ is defined by

$$A \otimes B = \begin{pmatrix}
a_{00}B & a_{01}B & \cdots & a_{0,l-1}B \\
a_{10}B & a_{11}B & \cdots & a_{1,l-1}B \\
\vdots & \vdots & \ddots & \vdots \\
a_{k-1,0}B & a_{k-1,1}B & \cdots & a_{k-1,l-1}B
\end{pmatrix}.$$ 

We see that $A \otimes B = \text{block}(a_{ij}B)$. The tensor product of an identity matrix with a dense matrix is block diagonal, i.e.,

$$I_l \otimes B = \begin{pmatrix}B & 0 & \cdots & 0 \\
0 & B & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & B
\end{pmatrix} = \text{diag}(B, \ldots, B),$$

where $I_l$ is the identity matrix of size $l$. For general block diagonal matrices we will also use a *direct sum* notation

$$\text{diag}(A_0, \ldots, A_{l-1}) = \begin{pmatrix}A_0 & 0 & \cdots & 0 \\
0 & A_1 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & A_{l-1}
\end{pmatrix} = A_0 \oplus \cdots \oplus A_{l-1} = \bigoplus_{k=0}^{l-1} A_k.$$

### 2.2 Properties of the DFT Matrix

The periodicity of $\omega_n$ introduces an extensive structure into $W_n$ that is easily exploitable to efficiently compute the DFT. Many coefficients of $W_n$ are either 1, −1,
$i$ or $-i$. For example

$$W_1 = (1), \quad W_2 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}, \quad W_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}. $$

All fast algorithms are based on taking advantage of this DFT structure in one way or another.

Important properties of the DFT matrix $W_n$ are:

- The matrix $W_n$ is symmetric, i.e., $W_n^T = W_n$.

- Let $W_n^H$ denote the conjugate transpose (Hermitian) of $W_n$. Then

$$W_n^H W_n = n I_n. \quad (2.4)$$

Hence, $W_n$ is a scaled unitary transformation on $\mathbb{C}^n$. It follows that the Inverse DFT is given by

$$x = W_n^{-1}X = \frac{1}{n}W_n^HX. \quad (2.5)$$

- The columns of $W_n$ are orthogonal with respect to the inner product $x \cdot y = x^H y$.

- The DFT solves the problem of trigonometric interpolation. If the function $f(x)$ is known on an interval $0 \leq x < L$ only at a discrete set of points $x_l = \frac{L}{n}l$, $l = 0, \ldots, n-1$, then it follows that the trigonometric polynomial

$$T_n(x) = \frac{1}{n} \sum_{l=0}^{n-1} c_l t_l(x) = \sum_{l=0}^{n-1} c_l e^{2\pi ilx/L}, \quad (2.6)$$

13
has the values \( f_l = f(x_l) \) at the points \( x_l \). The coefficients \( c_l \) are given as a DFT of \( f_l = f(x_l) \),

\[
c_l = \sum_{j=0}^{n-1} f(x_j)e^{-2\pi i lj/n} = \sum_{j=0}^{n-1} f_j e^{-2\pi ilj/n},
\]

This is also the best approximation of \( f(x) \) in the sense of the metric induced by the inner product \( \mathbf{x} \cdot \mathbf{y} = \mathbf{x}^H \mathbf{y} \).

### 2.3 The Cooley-Tukey Factorization

The fast Fourier transform algorithms are based upon sparse factorizations of the DFT matrix. Computation of \( W_n \mathbf{x} \) as a matrix–vector multiplication involves \( O(n^2) \) complex multiplications and summations. The FFT idea is to evaluate \( W_n \mathbf{x} \) as \( (A_1 \cdot A_2 \cdots \cdot A_r) \mathbf{x} \), where the matrices \( A_i \) are sparse and \( A_i \mathbf{x} \) involves \( O(n) \) operations. Here we illustrate the divide-and-conquer method known as the \( rdix-r \) splitting. If \( n = rq \), we can write \( j = mr + k \) and \( l = pq + s \), where \( 0 \leq k, p < r \) and \( 0 \leq m, s < q \). Also, let \( x_j = x_{mr+k} = x_k^m \) and \( X_l = X_{pq+s} = X_s^p \). Then,

\[
X_s^p = \sum_{k=0}^{r-1} \sum_{m=0}^{q-1} \omega_n^{mp} \omega_n^{sk} x_k^m.
\]

(2.7)

First, we note that \( \omega_n/(rq) = \omega_n/n = \omega_1 = 1 \) to any power is equal to one.

Next,

\[
\forall k \in [0, r - 1], \quad \sum_{m=0}^{q-1} \omega_q^{sm} x_k^m
\]

is a DFT of size \( q \) applied to a subvector of \( \mathbf{x} \) with components \( x_{mr+k}, m = \)
0, \ldots, q - 1$. For each $k \in [0, r - 1]$, denote by $Y_s^k, (k, s = 0, \ldots, q)$ the components of this transform, i.e.,

$$Y_s^k = \sum_{m=0}^{q-1} \omega_{q}^{sm} x_k^m.$$ 

Therefore, the next factor in Eq. (2.7)

$$X_s^p = \sum_{k=0}^{r-1} \omega_{r}^{pk} \omega_{s}^{sk} Y_s^k,$$  

(2.8)

represents a multiplication by a diagonal matrix of twiddle factors. Denoting by $Z_s^k$ the quantity $\omega_{s}^{sk} Y_s^k$ in Eq. (2.8), it follows that, $\forall s \in [0, \ldots, q - 1]$,

$$X_s^p = \sum_{k=0}^{r-1} \omega_{r}^{pk} Z_s^k,$$  

(2.9)

which is a DFT of size $r$. Using the Kronecker product notation, the matrix–vector product described in Eq. (2.9) can be expressed as:

$$X = W_n x$$

$$= (W_r \otimes I_q) D_{r,q} (I_r \otimes W_q) \Pi_{n,r} x,$$  

(2.10)

where

$$D_{r,q} = \bigoplus_{k=0}^{r-1} \Omega_{n,q}^k.$$ 

In matrix–vector notation we have

$$X = W_n x$$

$$= \text{block}(\omega_{r}^{ij} I_q) \text{diag}(I_q, \Omega_{n,q}, \ldots, \Omega_{n,q}^{r-1}) \text{diag}(W_q, \ldots, W_q) \Pi_{n,r} x,$$ 

where

$$\text{block}(\omega_{r}^{ij} I_q) = \begin{pmatrix} I_q & I_q & \ldots & I_q \\ I_q & \omega_r I_q & \ldots & \omega_r^{r-1} I_q \\ \vdots & \vdots & \ddots & \vdots \\ I_q & \omega_r^{r-1} & \ldots & \omega_r^{(r-1)^2} I_q \end{pmatrix}.$$
is a \( r \times r \) block matrix with the \( ij \)th block equal to \( \omega_r^{ij} I_q \); \( \text{diag}(I_q, \Omega_{n,q}, \ldots, \Omega_{n,q}^{r-1}) \) is a diagonal twiddle factor matrix of the form

\[
\text{diag}(I_q, \Omega_{n,q}, \ldots, \Omega_{n,q}^{r-1}) = \begin{pmatrix}
I_q \\
\Omega_{n,q} \\
\ddots \\
\Omega_{n,q}^{r-1}
\end{pmatrix},
\]

where \( \Omega_{n,q} = \text{diag}(1, \omega_n, \ldots, \omega_n^{q-1}) \); \( \text{diag}(W_q, \ldots, W_q) \) is a block-diagonal matrix of DFTs of size \( q \),

\[
\text{diag}(W_q, \ldots, W_q) = \begin{pmatrix}
W_q \\
\ddots \\
W_q
\end{pmatrix},
\]

and \( \Pi_{n,r} \) is a mod-\( r \) sort permutation matrix.

Hence, the DFT can be calculated as \( r \) independent DFTs of size \( q \), plus one diagonal scaling, and one \( q \)-vector DFT of size \( r \). This idea is the basis for the Mixed-Radix splitting algorithms for FFTs. The number of operations is thus reduced to \( O(qr^2 + n + qr^2) = O(n^2/r) \) or \( O(n^2/q) \).

This example also shows that a notation based only on \( \text{diag} \) and \( \text{block} \) is somewhat cumbersome for sparse matrix structures that arise in FFT algorithms, and that a notation based on the kronecker product is much more compact and clear.

### 2.4 Sparse factorizations

If \( q \) in Eq. (2.10) is not a prime number the algorithm can be applied recursively. This is the heart of the fast Fourier transform. If \( n = r^k \), the DFTs used are always
the same and the FFT algorithm obtained in this way is called a radix-$r$ Cooley–Tukey FFT. Otherwise, the FFT algorithm is called a mixed-radix Cooley–Tukey FFT. The sparse factorization of $W_n$ is not unique. Hence, there exist many different FFT algorithms. For example, since $W_n$ is symmetric, we obtain from Eq. (2.10)

$$X = W_n x = W_n^T x$$

$$= \Pi_{n,r}(I_r \otimes W_q)D_{r,q}(W_r \otimes I_q)x. \tag{2.11}$$

In order to achieve the above factorizations, one of the axes need to be mod-$r$ sorted. In Eq. (2.10) the data (time) axis is sorted, and this is usually referred to as decimation-in-time sorting or splitting. Similarly, the factorization in Eq. (2.11) is obtained by sorting the transform (frequency) axis and this results in decimation-in-frequency splitting.

In UHRFFT, we will use the following sparse factorization algorithms:

- Mixed-radix Cooley–Tukey (above),
- Prime Factor (PFA),
- Rader ,
- Split–Radix.

In the following sections we give a brief description of these factorization schemes.

### 2.4.1 Prime Factor Algorithms

The mixed-radix splitting algorithms described earlier rely on the splitting:

$$X = (W_{n_1} \otimes I_{n_2})D_{n_1,n_2}(I_{n_1} \otimes W_{n_2})\Pi_{n,n_1}x,$$
where \( n = n_1 n_2 \). In this algorithm a non-trivial fraction of the computational work is associated with the twiddle-factor multiplication, i.e., the application of the diagonal scaling matrix \( D_{n_1,n_2} \). The prime factor FFT algorithm \([8, 20]\) removes the need for this scaling. These reductions are possible only if \( n_1 \) and \( n_2 \) are relatively prime, meaning that their greatest common divisor is 1, i.e., \( \text{gcd}(n_1, n_2) = 1 \). The algorithm is based upon splittings of the form:

\[
W_n = P_1 (W_{n_1} \otimes I_{n_2}) (I_{n_1} \otimes W_{n_2}) P_2 = P_1 (W_{n_1} \otimes W_{n_2}) P_2,
\]

where \( P_1 \) and \( P_2 \) are permutations.

### 2.4.2 Rader’s Algorithm

When \( n \) is a prime, there is a factorization of \( W_n \) proposed by Rader \([5, 26]\) involving a number-theoretic permutation of \( W_n \) that produces a circulant or a skew-circulant submatrix of order \( n - 1 \).

**Definition 2** For \( c \in \mathbb{C}^n \), a matrix \( C_n \) with the property

\[
C_n e_j = R_n^j c \quad j = 0, \ldots, n - 1,
\]

where \( R_n \) is a downshift permutation,

\[
R_n \mathbf{x} =
\begin{pmatrix}
  x_{n-1} \\
  x_0 \\
  \vdots \\
  x_{n-2}
\end{pmatrix}, \quad
\mathbf{x} =
\begin{pmatrix}
  x_0 \\
  x_1 \\
  \vdots \\
  x_{n-1}
\end{pmatrix},
\]

and \( e_j \) is the \( j \)th unit vector is called a circulant matrix.
The inverse of $R_n$ is the upshift permutation

$$R_n^{-1} \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_0 \end{pmatrix}.$$

**Definition 3** The matrix $S_n$ whose $j^{th}$ column $S_ne_j$ is given by

$$S_ne_j = R_n^{-j} \mathbf{c},$$

is called a skew-circulant matrix.

Both $C_n$ and $S_n$ can be diagonalized by $W_n$, i.e.,

$$W_n C_n W_n^{-1} = \text{diag}(W_n \mathbf{c})$$

$$W_n S_n W_n = \text{diag}(W_n^{-1} \mathbf{c}).$$

Hence, the action of $C_n$ or $S_n$ can be obtained in $O(n \log_2 n)$ operations. The permutation used in Rader factorization is based on the multiplicative structure of the indexing set of $W_n$.

Let $\omega_n = e^{-2\pi i/n}$. Then, $W_n = (\omega_n^{ij})_{i,j=0}^{n-1}$. We can use the following notation for the elementwise power of the matrix

$$W_n = \omega_n \cdot \w_n,$$

where $\w_n$ is given by

$$\w_n = (<ij> \w_{i,j=0})^{n-1}.$$
For example,

$$W_5 = \omega_5 \cdot \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 \\ 0 & 2 & 4 & 1 \\ 0 & 4 & 3 & 2 \\ \end{pmatrix}. $$

It is easy to show that each row (or column) in the lower \((n-1) \times (n-1)\) submatrix of \(W_5\) is a permutation of the first row, hence there exists a permutation of \(W_5\) that will produce a circulant or skew-circulant submatrix in \(W_5\). The same is true for all \(W_n\), where \(n\) is prime. The permutation needed is based on the following definitions.

**Definition 4** If \(n\) is prime, then there exists an integer \(r\), called the primitive root, where \(2 \leq r \leq n-1\) such that

$$2, \ldots, n-1 = \langle r \rangle_n, \langle r^2 \rangle_n, \ldots, \langle r^{n-2} \rangle_n.$$ 

**Definition 5** If \(n\) is prime and \(r\) is its primitive root, the permutation defined by

$$z = Q_{n,r} x, \quad z_k = \begin{cases} x_k & \text{if } k = 0, 1 \\ x_{<k,r^{-1}>n} & \text{if } 2 \leq k \leq n-1 \\ \end{cases}$$

is called the exponential permutation associated with \(r\).

It can be shown that

$$W_n = Q_{n,r}^T \begin{pmatrix} 1 & 1_n^T \\ 1_n & C_{n-1} \end{pmatrix} Q_{n,r}^{-1} = Q_{n,r}^T \bar{W}_n Q_{n,r}^{-1}$$

and

$$W_n = Q_{n,r}^T \begin{pmatrix} 1 & 1_{n-1}^T \\ 1_{n-1} & S_{n-1} \end{pmatrix} Q_{n,r} = Q_{n,r}^T \bar{W}_n Q_{n,r},$$

20
where $C_{n-1}$ is a circulant matrix generated by the vector $c = (\omega_n, \omega_n^r, \omega_n^{r^2}, \ldots, \omega_n^{r^{n-2}})^T$.

Similarly, $S_{n-1}$ is a skew–circulant matrix generated by the same vector.

It is easy to show that $\hat{W}_n$ admits the following factorization

$$\hat{W}_n = (1 \oplus S_{n-1})A_n,$$

where

$$A_n = \begin{bmatrix} 1 & 1^T_{n-1} \\ -1_{n-1} & I_{n-1} \end{bmatrix}.$$

The application of $A_n$ involves $2(n-1)$ additions. The different variations of the Rader factorization follow from the different factorizations of $C_{n-1}$ or $S_{n-1}$. We describe the factorizations for the skew–circulant version of the algorithm. Similar factorizations can be obtained for $C_{n-1}$.

First we consider a full diagonalization of $S_{n-1}$. We have seen that

$$S_{n-1} = W_{n-1} \text{diag}(d)W_{n-1},$$

where $d = W_{n-1}^{-1}c$. Hence,

$$W_n = Q^TVDAQ,$$

where

$$Q = Q_{n,r},$$

$$V = 1 \oplus W_{n-1},$$

$$D = 1 \oplus \text{diag}(d).$$

The number of additions could be reduced by the following transformation. Let $A = V^{-1}BV$, then

$$W_n = Q^TVDBVQ,$$
where
\[ B = V A V^{-1} = \begin{pmatrix} 1 & e_{n-1}^T \\ (1-n) e_{n-1} & I_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ (1-n) & 1 \end{pmatrix} \oplus I_{n-2}, \]
where \( e_{n-1} \) is the first unit vector of size \( n-1 \). Application of matrix \( B \) requires two additions and one multiplication. This additional multiplication can be eliminated by using the symmetry of \( W_n \). Since \( W_n = W_n^T \) and \( V \) and \( D \) are also symmetric, it follows that
\[ W_n = Q^T V B^T D V Q \]
and the 0th component of \( W_n x \) is calculated as \( x_0 + y_1 \), where \( y = VQx \) is calculated in the process of calculation \( W_n x \).

### 2.4.3 Split–Radix Algorithm

Standard radix–2 procedures are based upon the fast synthesis of two half–length DFTs. The split–radix [6, 21] algorithm is based upon a clever synthesis of one–half length DFT together with two quarter–length DFTs, i.e.,

\[ W_n x(0 : n - 1) = \begin{cases} W_{n/2} x(0 : 2 : n - 1) \\ W_{n/4} x(1 : 4 : n - 1) \\ W_{n/4} x(3 : 4 : n - 1). \end{cases} \tag{2.12} \]

It turns out that the resulting procedure involves less arithmetic than any of the standard radix–2, radix–4 or radix–8 procedures. The split–radix idea was proposed by Duhamel and Hollmann (1984).

To illustrate the split–radix splitting, we begin with the radix–2 splitting. Let \( n = 2q = 4p \) and let \( x \in \mathbb{C}^n \). By using the radix–2 splitting we obtain
\[ W_n = (W_2 \otimes I_q) D_{n,q} (I_2 \otimes W_q) \Pi_{n,2}. \]
The split–radix algorithm is obtained by using the same formula again on the second block of the block–diagonal matrix $I_2 \otimes W_q = W_q \oplus W_q$, and rearranging the terms such that the final factorization is of the form
\[
W_n = B (W_q \oplus W_p \oplus W_p) \Pi_{\text{split–radix}}.
\]

It follows that
\[
W_n = (W_2 \otimes I_q) D_{n,q}[W_q \oplus (W_2 \otimes I_p) D_{q,p}(I_2 \otimes W_p) \Pi_{n,2}] \Pi_{n,2}
\]
\[
= (W_2 \otimes I_q) D_{n,q}[I_q \oplus (W_2 \otimes I_p) D_{q,p}] \Pi_{n,2}
\]
\[
\cdots (W_q \oplus W_p \oplus W_p)[(I_q \oplus \Pi_{q,2}) \Pi_{n,2}]
\]
\[
= B (W_q \oplus W_p \oplus W_p) \Pi_{n,q,2},
\]

where $B$ is the split–radix butterfly matrix and $\Pi_{n,q,2}$ is the split–radix permutation matrix
\[
\Pi_{n,q,2} = (I_q \oplus \Pi_{q,2}) \Pi_{n,2}.
\]

The efficiency of the split–radix algorithm follows from the simplifications of the butterfly matrix
\[
B = (W_2 \otimes I_q) D_{n,q}[I_q \oplus (W_2 \otimes I_p) D_{q,p}].
\]

First, we note that
\[
D_{n,q} = I_q \oplus \Omega_{n,q}
\]
\[
= I_q \oplus \Omega_{n,p} \oplus -i\Omega_{n,p}
\]
\[
= I_q \oplus (I_p \oplus -iI_p)(I_2 \otimes \Omega_{n,p})
\]
\[
= I_q \oplus (S \otimes I_p)(I_2 \otimes \Omega_{n,p})
\]

and
\[
D_{q,p} = I_p \oplus \Omega_{q,p} = I_p \oplus \Omega_{n,p}^2,
\]

23
where $\Omega_{n,k} = \ldots$, and $S = 1 \oplus -i$. It follows that

$$B = (W_2 \otimes I_q)[I_q \oplus (S \otimes I_p)(I_2 \otimes \Omega_{n,p})(W_2 \otimes I_p)(I_p \oplus \Omega_{n,p}^2)].$$

Next, we can show that matrices $(I_2 \otimes \Omega_{n,p})$ and $(W_2 \otimes I_p)$ commute since

$$(I_2 \otimes \Omega_{n,p})(W_2 \otimes I_p) = W_2 \otimes \Omega_{n,p} = (W_2 \otimes I_p)(I_2 \otimes \Omega_{n,p}).$$

Hence,

$$B = (W_2 \otimes I_q)[I_q \oplus (S \otimes I_p)(W_2 \otimes I_p)(I_q \oplus \Omega_{n,p} \oplus \Omega_{n,p}^3)]$$

$$= (W_2 \otimes I_q)[I_q \oplus (\tilde{W}_2 \otimes I_p)](I_q \oplus \Omega_{n,p} \oplus \Omega_{n,p}^3)$$

$$= B_a B_m,$$

where

$$\tilde{W}_2 = SW_2 = \begin{pmatrix} 1 & 1 \\ -i & i \end{pmatrix},$$

is the scaled radix-2 DFT matrix and $B_a = (W_2 \otimes I_q)[I_q \oplus (\tilde{W}_2 \otimes I_p)]$ and $B_m = (I_q \oplus \Omega_{n,p} \oplus \Omega_{n,p}^3)$ are respectively the additive and multiplicative parts of the butterfly matrix $B$. The matrix representation of the components of $B$ is

$$B_a = \begin{pmatrix} I_q & I_q \\ I_q & -I_q \end{pmatrix} \begin{pmatrix} I_q & 0 \\ 0 & C \end{pmatrix} = \begin{pmatrix} I_q & C \\ I_q & -C \end{pmatrix}$$

and $B_m = \begin{pmatrix} I_q & 0 \\ 0 & \Omega_{sr} \end{pmatrix}$,

where $C = \tilde{W}_2 \otimes I_p = \begin{pmatrix} I_p & I_p \\ -iI_p & iI_p \end{pmatrix}$ and $\Omega_{sr} = \begin{pmatrix} \Omega_{n,p} & 0 \\ 0 & \Omega_{n,p}^2 \end{pmatrix}$.

### 2.5 Properties Resulting from Real Input Data

The FFT algorithms described in section 2.4 revolve around recurring block structures that surface when the columns or rows of the DFT matrix are permuted. The structure of the DFT permits even more efficient evaluation of the FFT when the
input data is real. We will look into this structure as described in [26] and also explain how it is exploited in the code generator.

Let $\mathbb{R}^n$ denote the vector space of real $n$-vectors with components indexed from zero to $n - 1$. If $x \in \mathbb{R}^n$, then $y = W_n x$ is conjugate-even. We say that $y \in \mathbb{C}^n$ is conjugate-even if $\bar{y} = T_n y$, where $T_n$ is the reflection matrix given by

$$T_n = \begin{pmatrix} 1 & 0 \\ 0 & E_{n-1} \end{pmatrix}$$

and $E_n$ is the exchange matrix defined as follows:

$$E_n = I_n(:, n - 1 : -1 : 0), \text{ e.g., } E_3 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$ 

If $n = 2m$, the conjugate-even vector $y$ has the form

$$y = \begin{pmatrix} a \\ b + ic \\ d \\ E b - iEc \end{pmatrix},$$

where $a$ and $d$ are real scalars, $b$ and $c$ are real vectors of length $m - 1$ and $E = E_{m-1}$.

If $n = 2m + 1$, then the conjugate-even vector $y$ has the form

$$y = \begin{pmatrix} a \\ b + ic \\ E b - iEc \end{pmatrix},$$

where $a$ is a real scalar, $b$ and $c$ are real vectors of length $m$ and $E = E_m$. Vector $y$
can be represented as a single real $n$-vector

$$
y^{(ce)} = \begin{pmatrix}
a \\
b \\
c
\end{pmatrix},
$$

when $n$ is even and as

$$
y^{(ce)} = \begin{pmatrix}
a \\
b \\
c
\end{pmatrix},
$$

when $n$ is odd. This amount to the non-redundant stacking of the real and imaginary parts of $\mathbf{y}$. $\mathbf{y}^{(ce)}$ is a linear transformation of $\mathbf{y}$ which is easily seen by defining the permutation matrix

$$
U_n = \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & I/2 & 0 & E/2 \\
0 & 0 & 1 & 0 \\
0 & -iI/2 & 0 & iE/2
\end{pmatrix},
$$

where $n$ is even, $I = I_{m-1}$ and $E = E_{m-1}$. The relationship between $\mathbf{y}$ and $\mathbf{y}^{(ce)}$ is thus

$$
\mathbf{y}^{(ce)} = \begin{pmatrix}
\text{Re}(\mathbf{y}(0: m)) \\
\text{Im}(\mathbf{y}(1: m - 1))
\end{pmatrix} = U_n \mathbf{y}.
$$

The columns of $U_n$ are orthogonal and it’s inverse is

$$
U_n^{-1} = \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & I & 0 & iI \\
0 & 0 & 1 & 0 \\
0 & E & 0 & -iE
\end{pmatrix}.
$$

(2.13)
In the case that \( n \) is odd, the permutation matrix \( U_n \) would be defined as

\[
U_n = \begin{pmatrix}
1 & 0 & 0 \\
0 & I/2 & E/2 \\
0 & -iI/2 & iE/2
\end{pmatrix},
\]

where \( n = 2m + 1 \), \( I = I_m \) and \( E = E_m \), and it’s inverse would be

\[
U_n^{-1} = \begin{pmatrix}
1 & 0 & 0 \\
0 & I & iI \\
0 & E & -iE
\end{pmatrix}.
\]

Hence, if \( x \in \mathbb{R}^n \), the DFT of \( x \) is complex, conjugate–even, and it requires the same amount of storage as the original real vector. Moreover, due to symmetry, only one half of the components of the transform need to be computed. In the following we will first show a general procedure that can be used to derive factorization formulas for real data and then illustrate this procedure for two different factorization algorithms.

First, we note that when \( x \in \mathbb{R}^n \), the DFT of \( x \), \( y = W_n x \) can be stored as \( y^{(ce)} = U_n y \). If we apply this observation to the Mixed–radix factorization we have

\[
y^{(ce)} = U_n y = U_n W_n x = U_n (W_r \otimes I_q) D_{r,q} (I_r \otimes W_q) \Pi_{n,r} x.
\]

Next, the multiplication with \((I_r \otimes W_q)\) produces \( r \) conjugate even subvectors of size \( q \), and we can store those also using the nonredundant storage scheme by inserting the factor \((I_r \otimes U_q)^{-1}(I_r \otimes U_q)\) into Eq. (2.14)

\[
y^{(ce)} = U_n (W_r \otimes I_q) D_{r,q} (I_r \otimes U_q)^{-1}(I_r \otimes U_q)(I_r \otimes W_q) \Pi_{n,r} x.
\]

This implies that only half of the components of each size \( q \) transform needs to be computed. The final expression for the real mixed radix formula can be obtained by
simplifying Eq. (2.15). Similarly, for the PFA we have

\[ y^{(\infty)} = U_n P_1(W_r \otimes I_q)(I_r \otimes U_q)^{-1} (I_r \otimes U_q)(I_r \otimes W_q)P_2x. \]

In the following two subsections we show detailed derivations for the radix–2 Cooley–Tukey splitting and for the Split–radix algorithm.

It is to be noted that there are simpler methods that can be used to calculate the FFT’s on real input data. One such method is to use a complex FFT algorithm of size \( n \) to calculate 2 real FFT’s of size \( n \). This method, however does not produce the most efficient algorithms. This is because the FFT algorithms that take advantage of the symmetries that arise when real input data is used generally require less than half the number of arithmetic operations than those algorithms designed for complex data. For this reason, it is more expensive to do 2 real FFT’s of size \( n \) with a complex algorithm as opposed to using an FFT of size \( n \) twice, which takes advantage of the symmetries that arise due to the nature of the real input data. This can be seen in Table 2.1. A further disadvantage of using this method is that there is a post processing stage that is associated with de-interleaving the output. This post processing stage adds an overhead of \( 4 \times n \) (i.e., \( 2 \times n \) additions and \( 2 \times n \) multiplications).

### 2.5.1 The Conjugate–Even Butterfly

If \( x \in \mathbb{R}^n \), then the DFT of any subvector of \( x \) is conjugate–even. This implies that during the production of intermediate DFT’s in the computation of a radix–2 FFT, the butterflies combine two conjugate–even vectors to produce a single double length conjugate even vector. In order to take advantage of conjugate–even storage and thus avoid redundant computation, it is appropriate to examine the action of
### Table 2.1: Comparison of Operation count Between Complex and Real FFT's.

<table>
<thead>
<tr>
<th>Transform Size</th>
<th>No. of Operations (CPLX)</th>
<th>No. of Operations (real)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
<td>2.000</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>6</td>
<td>2.667</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>6</td>
<td>2.667</td>
</tr>
<tr>
<td>5</td>
<td>44</td>
<td>18</td>
<td>2.444</td>
</tr>
<tr>
<td>6</td>
<td>44</td>
<td>18</td>
<td>2.444</td>
</tr>
<tr>
<td>7</td>
<td>96</td>
<td>42</td>
<td>2.286</td>
</tr>
<tr>
<td>8</td>
<td>56</td>
<td>22</td>
<td>2.545</td>
</tr>
<tr>
<td>9</td>
<td>120</td>
<td>66</td>
<td>1.818</td>
</tr>
<tr>
<td>10</td>
<td>108</td>
<td>46</td>
<td>2.348</td>
</tr>
<tr>
<td>12</td>
<td>112</td>
<td>54</td>
<td>2.074</td>
</tr>
<tr>
<td>14</td>
<td>220</td>
<td>98</td>
<td>2.245</td>
</tr>
<tr>
<td>15</td>
<td>212</td>
<td>118</td>
<td>1.797</td>
</tr>
<tr>
<td>16</td>
<td>168</td>
<td>70</td>
<td>2.400</td>
</tr>
<tr>
<td>32</td>
<td>456</td>
<td>198</td>
<td>2.303</td>
</tr>
<tr>
<td>64</td>
<td>1160</td>
<td>518</td>
<td>2.239</td>
</tr>
</tbody>
</table>

the radix-2 butterfly and formulate a different butterfly, for real input data, called the conjugate–even butterfly.

Assume that $\mathbf{u}, \mathbf{v} \in \mathbb{C}^m$ are $m$ point DFT’s of real data. Also, assume that $n = 2m = 4p$ and consider the radix-2 synthesis,

$$
\mathbf{z} = B_n \begin{pmatrix} \mathbf{u} \\ \mathbf{v} \end{pmatrix} = \begin{pmatrix} I_m & \Omega_m \\ I_m & -\Omega_m \end{pmatrix} \begin{pmatrix} \mathbf{u} \\ \mathbf{v} \end{pmatrix},
$$

where $\Omega_n = \text{diag}(1, \omega_n, \ldots, \omega_n^{m-1})$. Since $\mathbf{z}$ is an $n$-point DFT of real data, it is conjugate–even and it follows from

$$
U_n \mathbf{z} = (U_n B_n (I_2 \otimes U_n^{-1}))(I_2 \otimes U_m) \begin{pmatrix} \mathbf{u} \\ \mathbf{v} \end{pmatrix}
$$

29
that

\[ z^{(ce)} = B_n^{(ce)} \begin{pmatrix} u^{(ce)} \\ v^{(ce)} \end{pmatrix}, \]

where

\[ B_n^{(ce)} = U_n \begin{pmatrix} U_m^{-1} & \Omega_m U_m^{-1} \\ U_m^{-1} & -\Omega_m U_m^{-1} \end{pmatrix}. \]  \hspace{1cm} (2.16)

To derive the block structure for \( U_n \), set \( I = I_{p-1}, E = E_{p-1} \) and \( \Delta = \text{diag}(1, \omega_n, \ldots, \omega_n^{p-1}) \).

Since \( \omega_n^{m-j} = \omega_n^m \omega_n^{-j} = -\omega_n^{-j} \), \( \Omega_m \) can be written as

\[ \Omega_m = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \Delta & 0 & 0 \\ 0 & 0 & -i & 0 \\ 0 & 0 & 0 & -E \Delta E \end{pmatrix}. \]

Then, from Eq. (2.13), it follows that

\[ \Omega_m U_m^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \Delta & 0 & i\Delta \\ 0 & 0 & -i & 0 \\ 0 & -E \Delta & 0 & iE \Delta \end{pmatrix}. \]

By substituting this result and

\[
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & I/2 & 0 & 0 & 0 & 0 & 0 & E/2 \\
0 & 0 & 0 & I/2 & 0 & E/2 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & -iI/2 & 0 & 0 & 0 & 0 & 0 & iE/2 \\
0 & 0 & -i/2 & 0 & 0 & 0 & i/2 & 0 \\
0 & 0 & 0 & -iI/2 & 0 & iE/2 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{pmatrix}
\]

\[ U_n = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & I/2 & 0 & 0 & 0 & 0 & 0 & E/2 \\
0 & 0 & 0 & I/2 & 0 & E/2 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & -iI/2 & 0 & 0 & 0 & 0 & 0 & iE/2 \\
0 & 0 & -i/2 & 0 & 0 & 0 & i/2 & 0 \\
0 & 0 & 0 & -iI/2 & 0 & iE/2 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{pmatrix}
\]

30
into Eq. (2.16), the following block representation for the conjugate–even butterfly is obtained:

\[
B_n^{(\omega)} = \begin{pmatrix}
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & I & 0 & 0 & 0 & C & 0 & -S \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & E & 0 & 0 & 0 & -EC & 0 & ES \\
1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & I & 0 & S & 0 & C \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & -E & 0 & ES & 0 & EC \\
\end{pmatrix},
\]

where \( C = \text{diag}(\cos(\theta), \ldots, \cos((p-1)\theta)) \), \( S = \text{diag}(\cos(\theta), \ldots, \cos((p-1)\theta)) \) and \( \theta = -2\pi/n \). The total number of computations required by the conjugate–even butterfly is half of that which is required by a fully complex butterfly of the same length.

A version of the Cooley-Tukey algorithm can be built upon this conjugate–even butterfly. This algorithm, known as the Edson Factorization, is described in detail in [27] and [28] and is implemented in our code generator described in Chapter 3.

### 2.5.2 The Split–Radix Algorithm for Real Data

We have seen that for \( x \in \mathbb{C}^n \), we have

\[
y = W_n x = B \begin{pmatrix}
x_1 \\
x_2 \\
x_3
\end{pmatrix},
\]
where
\[
\begin{align*}
x_1 &= W_{n/2} x(0:2:n-1) \\
x_2 &= W_{n/4} x(1:4:n-1) \\
x_3 &= W_{n/4} x(3:4:n-1).
\end{align*}
\]

and \( B \) is the split-radix butterfly

\[
B = B_d B_m = (W_2 \otimes I_{n/2})[I_{n/2} \otimes (S \otimes I_{n/4})(W_2 \otimes I_{n/4})] (I_{n/4} \oplus \Omega_{n,n/4} \oplus \Omega_{n,n/4}^3) .
\]

If \( x \in \mathbb{R}^n \), then \( T_n y = \tilde{y} \) and \( y^{ce} = U_n y \) and

\[
y^{(ce)} = U_n B(U_{n/2}^{-1} \oplus U_{n/4}^{-1} \oplus U_{n/4}^{-1}) \begin{pmatrix} x_1^{(ce)} \\ x_2^{(ce)} \\ x_3^{(ce)} \end{pmatrix} .
\]

We would like to simplify this expression to obtain a formula of the form

\[
y^{(ce)} = B^{(ce)} \begin{pmatrix} x_1^{(ce)} \\ x_2^{(ce)} \\ x_3^{(ce)} \end{pmatrix} .
\]

Let \( B_m^{(ce)} \) be defined as

\[
B_m^{(ce)} = B_m(U_{n/2}^{-1} \oplus U_{n/4}^{-1} \oplus U_{n/4}^{-1}) .
\]

In order to simplify \( B_m^{ce} \), we first observe that \( \Omega = \Omega_{n,n/4} \) has a quarter-wave symmetry

\[
(1 \oplus i E) \Omega = \tilde{\Omega}
\]

and it can be written as

\[
\Omega = \begin{cases} 
1 \oplus \Delta \oplus \delta \oplus -iE \Delta & \text{for } n/4 \text{ even} \\
1 \oplus \Delta \oplus -iE \Delta & \text{for } n/4 \text{ odd}
\end{cases}
\]

32
Since
\[
U_{n/4}^{-1} = \begin{cases}
1 \oplus \begin{pmatrix} I & 0 & iI \\ 0 & 1 & 0 \\ E & 0 & -iE \end{pmatrix} & \text{a for } n/4 \text{ even} \\
1 \oplus \begin{pmatrix} I & iI \\ E & -iE \end{pmatrix} & \text{for } n/4 \text{ odd}
\end{cases}
\]
it follows that
\[
\Omega U_{n/4}^{-1} = \begin{cases}
1 \oplus \begin{pmatrix} \Delta & 0 & i\Delta \\ 0 & \delta & 0 \\ -i\hat{\Delta} & 0 & -\hat{\Delta} \end{pmatrix} & \text{for } n/4 \text{ even}, \\
1 \oplus \begin{pmatrix} \Delta & i\Delta \\ -i\hat{\Delta} & -\hat{\Delta} \end{pmatrix} & \text{for } n/4 \text{ odd},
\end{cases}
\]
where
\[
\Delta = \text{diag}(d_k), \quad d_k = e^{-2\pi ik/n}, \quad k = 1, \ldots, m,
\]
\[
m = \begin{cases}
(p - 2)/n, & \text{for } p \text{ even,} \\
(p - 1)/n, & \text{for } p \text{ odd,}
\end{cases}
\]
\[
p = n/4,
\]
\[
\delta = \sqrt{2}/(1 - i),
\]
\[
\hat{\Delta} = E\bar{\Delta}E.
\]
Now, we write \( \Delta = C - iS \), where
\[
C = \text{diag}(c_k), \quad c_k = \cos 2\pi k/n, k = 1, \ldots, m, \text{ and}
\]
\[
S = \text{diag}(s_k), \quad s_k = \sin 2\pi k/n, k = 1, \ldots, m.
\]
Let $\hat{\mathcal{C}} = ECE$ and $\hat{S} = ESE$. Hence,

$$\Omega U_p^{-1} = 1 \oplus \begin{bmatrix} \begin{pmatrix} C & 0 & S \\ \sqrt{2}/2 & 0 \\ \hat{S} & 0 & -\hat{C} \end{pmatrix} - i \begin{pmatrix} S & 0 & -C \\ 0 & \sqrt{2}/2 & 0 \\ \hat{C} & 0 & -\hat{S} \end{pmatrix} \end{bmatrix}.$$ 

Now, let $z_2 = \Omega U_p^{-1} x_2^{(ce)}$ and for $x_2^{(ce)} = (a, b, c, d)^T$, we obtain

$$z_2 = \Omega U_p^{-1} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} a \\ Cb + Sd \\ \sqrt{2}/2c \\ \hat{S}b - \hat{C}d \end{pmatrix} - i \begin{pmatrix} 0 \\ Sb - Cd \\ \sqrt{2}/2c \\ \hat{C}b + \hat{S}d \end{pmatrix}.$$ 

It is easy to show that $z_2$ also has a quarter-wave symmetry

$$(1 \oplus iE)z_2 = \begin{pmatrix} a \\ Cb + Sd \\ \sqrt{2}/2c \\ \hat{S}b - \hat{C}d \end{pmatrix} + i \begin{pmatrix} 0 \\ Sb - Cd \\ \sqrt{2}/2c \\ \hat{C}b + \hat{S}d \end{pmatrix} = \hat{Z}.$$ 

Hence, the non-redundant stacking of the real and imaginary parts of $z_2$ leads to

$$z_2^{(cc)} = \begin{pmatrix} a \\ Cb + Sd \\ \sqrt{2}/2c \\ Sb - Cd \end{pmatrix} = \hat{U}_p z_2,$$

where

$$\hat{U}_p^{-1} = 1 \oplus \begin{pmatrix} I & 0 & -iI \\ 0 & (1 - i) & 0 \\ -iE & 0 & E \end{pmatrix}.$$
By taking a cubic of both sides of Eq. (2.17), we obtain

\[ \Omega^3 = (1 \oplus iE)^3 \bar{\Omega}^3 = (1 \oplus -iE)\bar{\Omega}^3. \]

We see that \( \Omega^3 \) has a similar symmetry as \( \Omega \) and we will refer to it as a cubic quarter-wave symmetry. By taking the same steps as before, we obtain

\[ \Omega^3 = 1 \oplus \Delta^3 \oplus \delta^3 \oplus iE\overline{\Delta^3} \]

and

\[ \Omega^3 U_p^{-1} = 1 \oplus \begin{pmatrix} \Delta^3 & 0 & i\Delta^3 \\ 0 & \delta^3 & 0 \\ -i\bar{\Delta}^3 & 0 & \bar{\Delta}^3 \end{pmatrix}, \]

where

\[ \Delta^3 = \text{diag}(d_k), \quad d_k = e^{-6\pi ik/n}, \quad k = 1, \ldots, m, \]

\[ \delta^3 = [\sqrt{2}/2(1 - i)]^3 = -\sqrt{2}/2(1 + i), \quad \text{and} \]

\[ \overline{\Delta^3} = E\overline{\Delta^3}E. \]

Let \( \Delta^3 = C_3 - iS_3 \) and \( \bar{C}_3 = EC_3E \), then

\[ \Omega^3 U_p^{-1} = 1 \oplus \begin{pmatrix} C_3 & 0 & S_3 \\ 0 & -\sqrt{2}/2 & 0 \\ -\bar{S}_3 & 0 & \bar{C}_3 \end{pmatrix} \quad -i \begin{pmatrix} -S_3 & 0 & C_3 \\ 0 & -\sqrt{2}/2 & 0 \\ \bar{C}_3 & 0 & \bar{S}_3 \end{pmatrix} \]

and

\[ z_3 = \Omega^3 U_p^{-1} x_3^{(ce)} = \Omega^3 U_p^{-1} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} a \\ C_3 b + S_3 d \\ -\sqrt{2}/2c \\ -\bar{S}_3 b + \bar{C}_3 d \end{pmatrix} + i \begin{pmatrix} 0 \\ -S_3 b + C_3 d \\ -\sqrt{2}/2c \\ \bar{C}_3 b + \bar{S}_3 d \end{pmatrix}. \]

It is easy to show that \( z_3 \) has the same symmetry as \( \Omega^3 \), namely \((1 \oplus -iE)z_3 = z_3\).
Again, the nonredundant stacking leads to

\[
    z_3^{(\alpha)} = \begin{pmatrix}
    a \\
    C_3 b + S_3 d \\
    -\sqrt{2}/2c \\
    -S_3 b + C_3 d
    \end{pmatrix} = \tilde{U}_p z_3,
\]

where

\[
    \tilde{U}_p^{-1} = 1 \oplus \begin{pmatrix}
    I & 0 & iI \\
    0 & 1 + i & 0 \\
    iE & 0 & E
    \end{pmatrix}
\]

for \( p \) even.
Chapter 3

Methodology

In this chapter, we describe the methodology we have used to address the problem of optimizing an FFT library. First we explain the general methodology used, which consists of two levels of optimization. Then, we go on to explain both the higher level optimization and the lower level optimization methods used.

3.1 Approach to the Problem

There are many different approaches to the problem of FFT code optimization. The two main strategies that interest us are:

- Adaptive and portable methods: The library is designed to perform well on all architectures. The concentration is on size and architecture dependent selection of algorithms, such that the code performs very well on all architectures rather than just on one platform.

- Architecture specific methods: Vendor libraries are usually optimized for a particular processor architecture to achieve the best possible performance on
that particular machine. Usually some low level optimizations involving the processor architecture are used, thereby limiting the portability of the code.

Our FFT library uses an adaptive approach similar to the one used by the FFTW library [14] from MIT. The main idea is to develop a library which can be used over many different platforms. We further extend the portability by providing interfaces for both C and Fortran. The optimization of the FFT routines in our library is performed on two levels. The reason we have taken this approach is the complexity of the global optimization problem for an arbitrary size FFT. Hence, instead of trying to build an optimal code for every possible transform size, a library of relatively small FFT codelets is built, and the code for all other sizes is obtained by optimally combining the codelets from the library for supported factorizations of the DFT matrix. The problem of finding the optimal factorization and codelet scheduling for a given processor with a specific memory configuration is difficult, if not impossible. The best that we can hope for is to find an algorithm among all known versions of FFT algorithms for a particular transform size that performs the best on a specific machine.

In our approach, the first (high) level optimization consists of selecting the optimal factorization of the FFT of a given size for which efficient DFT codelets exist in our library. The optimization on this level is performed during the initialization phase of the procedure, which makes the code adaptive to the architecture it is running on. A detailed explanation of the factorization schemes and the associated performance optimization strategy is given in section 3.2.

The second (low) level optimization involves generating a library of efficient, small size DFT codelets. Since the efficiency of the code depends strongly on the efficiency of the codelets themselves, it is important to have the best possible performance for
the codelets to be able to build an efficient library. The methods used to generate these codelets and optimize them are discussed in section 3.3.

This layered approach to optimization makes future improvements of the library performance through inclusion of additional or revised codelets simple and convenient. The methodology allows for the whole library to be regenerated in a few seconds with all codelets optimized using a new set of optimization rules. The methodology also allows for modification of execution strategies to include new factorization algorithms, which may improve the overall performance for a particular size FFT.

3.2 Execution Plan Generation

We use an adaptive procedure for the selection of DFT codelets. The objective is to select the computationally most efficient factorization of the FFT based on small factors for which codelets already exist in the library. This selection step is performed during the initialization phase of the procedure for a given FFT size. Given the parameters of the problem, the initialization routine selects the strategy in terms of execution time on the given architecture. This selection involves two steps.

We use the Split–Radix Algorithm to generate a large number of possible factorizations for a given transform size. Next, we seek to select the fastest factorization in terms of the actual execution time on the given architecture. There are essentially two methods used in choosing the set of codelets from among the various possible factorization schemes. The first method involves measuring the actual time taken for each factorization scheme and choosing the fastest one. This method is time consuming and may not be feasible for all applications, especially if the transform
size is large since there are a large number of possible factorizations. The second method uses data collected and stored in two databases. These are

- The **codelet database** which stores information about codelet execution times
- The **transform database** which stores information about the execution times for entire transforms.

The codelet database is initialized during installation of the library as a part of the benchmarking routine. It is then updated every time a new codelet is added to the library. The transform database stores the best execution plan for different size transforms along with best, worst and average execution times for that particular plan and the number of times the particular plan has been executed. The transform database is initialized for some of the popular FFT sizes during installation (such as power of 2 sizes) and is updated everytime a transform is executed. If the transform size already exists in the database, then the current execution time for that particular transform is added to the transform database to get an average over all past executions. It also updates the best or worst execution times if the current execution time happens to be the best or the worst. If a new transform size is executed, data from that execution is added to the transform database and these data can be used by the library for later executions of the same size transform. Hence, in a sense, the estimation algorithm **learns** from each execution.

For transform sizes that are not in the database, an execution plan is created based on supported factorizations for the given transform size. A plan that either minimizes the real or estimated execution time is then selected and code is generated for the selected plan.

To find a plan that minimizes the real execution time, all supported plans for
the given size are executed and the plan with the best performance is chosen. This method ensures that the plan selected will indeed result in the smallest execution time for all choices possible within the UHRFFT library. This approach only makes sense when a large number of executions are expected. A major drawback to using this approach is that the time required to find the execution plan may be quite large for large size FFTs.

To find a plan that minimizes the estimated execution time, the information in the codelet database is used. For each execution plan feasible with the codelets in the library, the expected execution time is derived based on the performance of the codelets being used in the plan and the number of calls to each codelet. The estimation algorithm also takes into account the input and output strides and transform direction (forward or inverse). It also accounts for the twiddle factor multiplications for each plan as the number of such multiplications depend on the execution plan. The estimation algorithm essentially *mimics* the actual execution of the transform. An execution plan estimated to require the least execution time is then selected for code generation. In case more then one plan is estimated to yield the minimum execution time, then the choice of execution plan for code generation is based on the factorization which has the largest size codelets in it. This ensures that more straight line code is included in the execution plan using this method.

For large transform sizes with many factorizations, the estimation method is considerably faster than the empirical method. The quality of the execution plan based on the estimation approach clearly relies heavily on the assumption that codelet timings can be used to predict transform execution times, and that the memory system will have a comparable impact on all execution plans. We plan to improve this simplistic model, but the performance of the execution plans selected by this
simple estimation method seem to be modestly suboptimal (typically about 10 – 20%), as shown in Chapter 4.

The list of codelets, execution strategy, twiddle factors and other information needed by the application to call the FFT routine are stored in a special structure called the *FftPlan*. Once the execution plan for a given transform size is generated, the application can use the structure to compute any number of FFTs of the given size. The execution of the final transform is based on a recursive algorithm. The execution routine uses the *FftPlan* structure to recursively call the different codelets in the plan with the corresponding data and strides to compute the transform of the particular size.

### 3.3 Library of FFT Codelets

The FFT library contains a number of composable blocks of code, called *codelets*, each computing a part of the transform. The overall efficiency of the code depends strongly on the efficiency of these codelets. Therefore, it is essential to have a highly optimized set of DFT codelets in the library. There are many methods that are used in optimizing our library of codelets. One optimization involves reductions in the number of arithmetic operations for each DFT codelet. Currently, the codelet algorithm is chosen such that the number of arithmetic operations for a given codelet size is minimized. Though, in general, the execution time is not necessarily minimized by minimizing the number of arithmetic operations since the memory architecture, in particular cache behavior, tend to be at least as important in modern processor architectures, we have found that for the codelets minimizing the number of arithmetic operations in most cases does minimize the execution time as well with carefully
selected scheduling of the operations. For example, the Mixed-Radix algorithm requires a larger number of arithmetic operations than the PFA for a codelet of size 6, as shown in Table 3.1. Hence, the PFA is chosen over the Mixed-Radix algorithm for codelets of size 6.

<table>
<thead>
<tr>
<th>Transform Size</th>
<th>UH Mixed-Radix</th>
<th>UH PFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>16</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 3.1: Number of Arithmetic Operations for a Codelet of Size 16.

The next level of optimization involves the memory hierarchy. In current processor architectures, memory access time is of prime concern for performance. Hence, it is essential to make use of the memory hierarchy in such a way as to minimize the impact of memory systems not capable of delivering data and instructions at a rate needed for full utilization of functional units. Optimizations involving memory accesses are architecture dependent and are performed only once during the installation of the library. The configuration of the machine, memory size and cache size are important factors in determining this optimization process. The main idea is to reduce memory access time by proper alignment of data and effective utilization of the cache. Small codelets for which the data fits entirely inside the data cache are seen to be very effective since access time from the cache is very low. For larger codelets, the scheduling of memory accesses during computations is very crucial. It is important to make effective use of data already residing in the cache so that fewer number of memory references occur. We use blocking techniques to ensure this and also use register variables to hold temporary data during computations. This ensures that temporary data are written to and read from the cache or registers, and
not from the main memory. A considerable performance boost is achieved by such effective utilization of the cache.

Some processor architectures have a reasonably large instruction cache along with the data cache. Larger codelets can easily fit into the instruction cache, thereby reducing the \textit{fetch, decode and execute} cycle time for each instruction. The UHRFFT codelets make effective use of large instruction caches if available, as the codelets are entirely straight line codes. Straight line codes are also effective on architectures without instruction caches. By eliminating loops the chances for instruction pipelining and a predictable control flow is increased. The codelet features are such that they can be expected to perform well on most architectures.

Our library consists of a number of highly optimized codelets that are generated using a special purpose compiler that we have developed. We used this automatic code generation approach because the actual coding and optimization of these codelets become very tedious and difficult for transform sizes greater than five. For this reason many authors have found it convenient to build the codelets by using different ways of automatic code generation. Our code generator is written in C. It can produce DFT codelets of arbitrary size, direction (forward or inverse), and rotation (for PFA). It first generates an abstraction of the FFT algorithm by using a combination of Rader’s algorithm [5], the Mixed–Radix algorithm [25] and the PFA. The next step is the scheduling of the arithmetic operations such that memory accesses are minimized. We make effective use of temporary variables so that intermediate writes use the cache instead of writing directly to memory. We also use blocking techniques so that data residing in the cache is reused the maximum possible number of times without being written and re-read from main memory.
Finally, the abstract code is unparsed to produce the desired C code. The output of the code-generator is then compiled to produce the executable version of the library. Any ANSI C compliant compiler can be used for this compilation. The strategy to produce codelets that can be compiled with any ANSI C compliant compiler enables the ultimate portability, since there is no need to install any particular compiler on the platform targeted for the UHFFT library installation. However, the strategy does introduce a degree of uncertainty with respect to the performance that will be achieved, and stability of the optimizations being used in generating the codelets. Though we have observed that the output generated by different compilers on different processor architectures vary greatly, the optimizations nevertheless seem to be quite stable in the sense that the choices for code arrangement of a particular codelet that we expect to perform best on a particular architecture in most cases we have studied is independent of the compiler used. As for compiler optimization options, we have noticed that for most compilers and most architectures, the optimization level "02" produces the best performance for most codelets. Since all the codelets are straight line codes without loops, higher level optimizations generally tend to affect the performance of the codelets in a negative way. Some architectures, such as the Intel Pentium-II and III processors and the IBM RS 6000s, have features that allow alignment of data with the cache lines. This option for the compilers is seen to slightly improve the performance on these architectures. Most other compiler options do not seem to enhance the performance in any significant manner. The different compiler options we have used to achieve good performance on all evaluated platforms are listed in Table 5.1.

Once the executables for the library are ready, we benchmark the codelets to test their performance. These benchmark tests are conducted for various input and
output strides of data. The results of these performance tests are then stored in a database that is used later by the execution plan generator algorithm during the initialization phase of an FFT computation. The structure of the library is given in Figure 3.1. We have compared the performance and functionality of our library of DFT codelets to similar, already existing libraries. For the PFA codelets we compared our library with the pfafft code from the CWP library. For all other codelets we compared our library with the FFTW library codelets. Our library can also accommodate codelets developed by other authors with minor modifications. Analysis of the performance of these codelets is given in Chapter 4.
3.4 Description of the Code Generator

In this chapter, a description of the structures that define the code generator is given as well as the operators used on these structures. The latter part of this chapter describes how the different FFT algorithms use these structures in the generation of the codelets. The main idea is to obtain an abstraction for a given algorithm. This abstraction is given in the form of a list of expressions.

3.4.1 Basic Structures

The basic structure in the code generator is the Expression. An expression is a polymorphic structure that takes one of the following forms:

- **Constant Expression**: A constant can be a real constant (REALCONST), an integer constant (INTEGERCONST), a Zero constant (ZEROCONST) or a constant with a value equal to one (ONECONST). All constants are stored as doubles.

- **Variable Expression**: This is an abstraction for the variables used in the FFT algorithms. The interpretation of the variable depends on its type, which is a composition of two attributes:
  
  o relevance to the algorithm, i.e., input, output or temporary.
  o type of data stored, i.e., real, imaginary or complex.

Hence, the complete list of variables is:

- Real input variable (INPUTREVAR)
- Imaginary input variable (INPUTIMVAR)
o Complex input variable (CPLEXINVAR)
o Real output variable (OUTPUTREVAR)
o Imaginary output variable (OUTPUTIMVAR)
o Complex output variable (CPLEXOUTVAR)
o Temporary real variable (TEMPREVAR)
o Temporary imaginary variable (TEMPIMVAR)
o Temporary complex variable (CPLEXTMPVAR)

• **Sum Expression**: This is an abstraction form for composite expressions of the form:

\[ \sum_{i=0}^{N} e_i, \]

where \( e_i \) are expressions.

• **Product Expression**: This is an abstraction form for composite expressions of the form:

\[ \prod_{i=0}^{N} e_i, \]

where \( e_i \) are expressions.

• **Negative Expression**: This is an abstraction for the change of sign. It is also used for subtraction e.g. the statement `SumExpr(expr1, NegExpr(expr2))` amounts to subtracting `expr2` from `expr1`.

• **Assign Expression**: This is an abstraction for expressions of the form \( v = e \), where \( v \) is a variable and \( e \) is an expression.
• **Complex Expression**: This is the abstraction for complex arithmetic. It consists of two expressions, one for the real part and one for the imaginary part.

### 3.4.2 Basic Operators

The three basic operations on Expressions are the sum (**SumExpr**), the product (**MultExpr**) and the sign change (**NegExpr**). These three operations are used in the generation of Expression Lists. The derived expressions may contain some redundancies which are removed to reduce the total number of arithmetic operations in the expression list. Some of the simplifications include:

  - Cancellation of terms in sums, e.g., \((\text{tmp}3 + \text{tmp}5) + (\text{tmp}6 - \text{tmp}3)\) would yield \(\text{tmp}5 + \text{tmp}6\).
  
  - Trivial multiplications are eliminated, i.e., multiplications by \(\pm 1, \pm i\) and \(0\).
  
  - Common constants are factored out, e.g., \(C2 * \text{tmp}5 + C2 * \text{tmp}7\) would yield \(C2 * (\text{tmp}5 + \text{tmp}7)\).
  
  - Constants are folded, e.g., \(C2 * C3 * C4 * \text{tmp}5\) would yeild \(C5 * \text{tmp}5\) where \(C5 = C2 * C3 * C4\).

  - Multiple sign changes are simplified, e.g., \(-(\text{tmp}2)\) would yield \(\text{tmp}2\).

  - Complete complex algebra is included, i.e., the generator is able to deal with complex expressions.
3.4.3 Derived Structures and Operations

From the basic structures described above, other more complex structures can be derived. The Expression Vector (ExprVec) is a vector of expressions of which each expression may be of one of the defined types for an expression. The Expression Matrix (ExprMat) is a two dimensional structure of expressions. There are a number of operations that are used to manipulate these structures. These operations obey the rules of vector addition and multiplication, e.g., to add two vectors together, they must be of the same size. These operations include but are not restricted to the following:

- Multiplication by a Constant. **ScaleExprVec**: This multiplies each component of a vector by a scalar expression.

- Sum of Vectors. **SumExprVec**: This adds corresponding components of two vectors and assigns it to a third vector.

- Matrix–Vector Product. **GetExprMatVec**: This multiplies a given matrix with a vector and returns a vector of appropriate size.

- Componentwise Product of Two Vectors. **MultExprVec**: This multiplies corresponding components of two vectors and assigns it to a third vector.

- Subvector operators. There are two types:
  - **GetSubExprVec**: This operator extracts a vector \( x \) from another vector \( y \) where \( x \subseteq y \). It takes a beginning index (from where extraction should start), an ending index (to where extraction should end) and a stride (which determines the elements to be extracted between the beginning and
ending indicies). The size of the vector \( x \) is

\[
( \text{Ending Index} - \text{Starting Index} + 1) / \text{Stride}.
\]

- PutSubExprVec: This operator assigns the elements of one vector to specific elements in another vector. It takes a beginning index (from where assignment should start), an ending index (to where assignment should end) and a stride (which determines the elements to be replaced in the assignment between the beginning and ending indicies).

- Concatination Operator. ConcatExprVec: This operator takes as input two vectors and assigns the first to the leading part of the resultant vector and the second to the tail part of the resultant vector. The size of the resultant vector is the sum of the sizes of the two input vectors.

An object oriented approach was taken in building the code generator. This implies that structures that are built on the abstraction of the basic structures inherit all the properties of the basic structures. It also implies that simplifications that are done on the simple structures are propagated to the derived structures.

### 3.4.4 FFT Operations

The FFT operations generate lists of expressions that are used to form the abstractions for the different FFT algorithms. They include:

- GenDFTExprMat: This function returns the DFT expression matrix that consists of complex constant expressions. It takes in the DFT size and the transform direction as its input arguments.
• **GenFFTExprList**: This function generates the expression list for the FFT of size \( n \). It takes in as arguments the input vector (to be transformed) and the direction of the required transform and then calls the appropriate FFT algorithm, depending on the size of the input vector. Pseudo code for this function is as follows:

\[
\text{GenFFTExprList( ExprVec *u) :} \\
\text{n = Length of vector u} \\
\text{if n = 2} \\
\text{apply DFT to the vector u} \\
\text{else if n is prime} \\
\text{call RADER’S algorithm} \\
\text{else} \\
\{ \\
\text{choose factor r of n} \\
\text{if r and n/r are co-prime} \\
\text{call PRIME FACTOR algorithm} \\
\text{else if n > 8 and n mod 4 = 0} \\
\text{call SPLIT-RADIX algorithm} \\
\text{else} \\
\text{call MIXED-RADIX algorithm} \\
\}\]

• \( I_r \times FFT_m \): This function evaluates the tensor (kroneker) product of the identity matrix of size \( r \) with the FFTExprList of size \( m \). This operation is also known as the parallel FFT.

• \( FFT_r \times I_m \): This function evaluates the tensor (kroneker) product of the FFT-ExprList of size \( r \) with the identity matrix of size \( m \). This operation is also known as the vector FFT.

• **Mixed–Radix Algorithm**: This returns a list of expressions which is an abstraction for the Mixed–Radix FFT algorithm, as described in Section 2.4.
• **Split–Radix Algorithm**: This returns a list of expressions which is an abstraction for the Split–Radix FFT algorithm, as described in Section 2.4.3.

• **Prime Factor Algorithm**: This returns a list of expressions which is an abstraction for the Prime Factor FFT algorithm, as described in Section 2.4.1.

• **Rader’s Algorithm**: This returns a list of expressions which is an abstraction for Rader’s FFT algorithm, as described in Section 2.4.2.
Chapter 4

Accuracy Tests and Performance Analysis

In this chapter we describe the tests that are used to verify the correctness and accuracy of the codelets generated and then analyze the performance and efficiency of our FFT library. Since the optimization is performed on two levels, we analyze the optimizations on the two levels separately. We first give a description of the accuracy tests and then compare the different algorithms for the FFT codelets and analyze the performance of the codelets on different platforms. Lastly, we analyze the high level optimization of the execution strategy and compare the performance of different strategies. A brief description of the different hardware platforms and environments that we used to test our library is also given.
4.1 Accuracy Tests

Four different tests are implemented in the code generator to validate the correctness of the generated code. The tests are based on the properties of the DFT. The tests include:

- **Constant Test**

  The constant test checks the accuracy of the row sums in the DFT matrix. It is a good check for the exactness of the constants used by the FFT algorithm. This test takes an input vector of all ones, i.e.,

  \[ x = (x_j)_{j=0}^{N-1}, \text{ where } x_j = 1, \]

  and multiplies the DFT matrix with this vector. This amounts to summing the individual rows of the DFT matrix and assigning the result to the corresponding components of the output vector. The expected result of this operation is

  \[ y_k = \sum_{j=0}^{N-1} e^{-2\pi j/N} = \begin{cases} N & \text{for } k = 0, \\ 0 & \text{otherwise}. \end{cases} \]

  Example of a size 4 DFT would yeild:

  \[
  \begin{pmatrix}
  1 & 1 & 1 & 1 \\
  1 & -i & -1 & i \\
  1 & -1 & 1 & -1 \\
  1 & i & -1 & -i
  \end{pmatrix}
  \begin{pmatrix}
  1 \\
  1 \\
  1 \\
  1
  \end{pmatrix}
  =
  \begin{pmatrix}
  4 \\
  0 \\
  0 \\
  0
  \end{pmatrix}.
  \]

  Although the constant test checks the validity of the DFT matrix, it does not check for errors that may arise from permutations that are used in calculating the FFT.
• **Square Test**

The square test takes in as input a square impulse and calculates the FFT of this impulse. It is easy to show that for \( x = (x_j)_{j=0}^{N-1} \), where \( x_j = (-1)^j \),

\[
y_k = \begin{cases} 
0 & \text{N is even, } k \neq N/2, \\
N & \text{N is even, } k = N/2, \\
1 + i \tan(\pi k / N) & \text{N is odd.}
\end{cases}
\]

• **Ramp Test**

The input for the ramp test is a constantly increasing sequence, i.e., \( x = (x_j)_{j=0}^{N-1} \), where \( x_j = j \). It has been shown that the expected result is:

\[
y_k = \begin{cases} 
N(N - 1)/2 = \sum_{j=0}^{N-1} j & \text{for } k = 0, \\
N/2(-1 + i \cot(\pi k / N)) & \text{otherwise.}
\end{cases}
\]

As the size of the FFT increases, the ramp test becomes more unstable. This is because there is an increased loss of precision when numbers of different precision are added and subtracted.

• **Inverse Test**

The inverse test takes a random input vector and passes it through a forward FFT algorithm and then takes the resultant vector and inputs it into the inverse FFT algorithm. The expected final output vector should be identical to the original input vector. Mathematically,

\[
x = F_N^{-1}(F_N(x)).
\]

It is important to note that any nonsingular linear function exhibits this property with its inverse function. It is therefore necessary to use a combination of
the previously described tests with the inverse test to verify that the function in question is in fact an FFT.

4.2 Target Hardware Architectures

Different hardware architectures were used to benchmark the performance of the generated library. The platforms chosen were different in the core processor architecture (RISC vs. CISC, number of Functional Units, clock rate) as well as the overall system design (cache size, cache type, size of main memory). This provided us with an opportunity to test for the robustness of the generated library seeing that one of our main goals was to have the ability to generate code that was portable and performed well on different hardware targets. The hardware architectures used were:

- The SGI R10000 Processor
- The IBM Power3 Processor
- The Intel Pentium-III Processor
  - The Intel Pentium-III 500
  - The Intel Pentium-III 550 (Xeon)
  - The Intel Pentium-III 600 (Coppermine)

4.2.1 The SGI R10000 Processor

The SGI Origin 2000 [29] at NCSA has 1528 MIPS R10000 64-bit processors [31] of which 760 operate at 195 MHz and the remaining operate at 250 MHz. We used the 250 MHz processors with the IRIX 6.5.1 operating system for our tests. The
SGI R10000 processor supports four instructions per cycle, i.e., two integer and two floating-point instructions plus one load/store per cycle. Thus, peak performance achievable is 500 MFlops (Flop = floating-point operations per second) per processor. The processor has as primary caches a 32 KB two-way set-associative on-chip instruction cache and a 32 KB two-way set-associative, two-way interleaved on-chip data cache with LRU replacement. It also has a 4 MB two-way set-associative L2 secondary cache per CPU. The SGI R10000 has 64 physical registers, each 64 bits wide. A summary of the SGI R10000 hardware is given in Table 4.1.

<table>
<thead>
<tr>
<th>Processor</th>
<th>SGI R10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock Speed</td>
<td>250 MHz</td>
</tr>
<tr>
<td>Peak Performance</td>
<td>500 MFlops</td>
</tr>
<tr>
<td>Primary Data Cache</td>
<td>32 KB on-chip(32 Bytes)</td>
</tr>
<tr>
<td>Secondary Data Cache</td>
<td>4 MB(128 Bytes)</td>
</tr>
<tr>
<td>Instruction Cache</td>
<td>32 KB on-chip</td>
</tr>
<tr>
<td>Cache Type</td>
<td>Two-Way Set Associative</td>
</tr>
<tr>
<td>Operating System</td>
<td>IRIX 6.5.1</td>
</tr>
</tbody>
</table>

Table 4.1: SGI R10000 Processor Characteristics.

4.2.2 The IBM Power3 Processor

The Blue Horizon cluster installed at NPACI/SDSC is a teraflop scale machine based on the Power3 processor built by IBM. The machine contains 1,152 processors and 512 GBytes of main memory, arranged as 144 Symmetric Multiprocessing (SMP) compute nodes. Each node is equipped with 4 Gbytes of memory shared among its eight 222MHz Power3 processors. The Power3 processors are super-scalar pipelined 64-bit RISC chips with two Floating-Point Units, three Integer Units and are capable of up to 8 instructions per cycle and up to 4 floating-point operations (two fused
multiply-adds) per cycle giving the processors a peak performance of 888 MFLOPS. Each Power3 processor has a 64KB L1 cache which is 128-way set associative and a 4MB L2 cache which is direct-mapped with its own private cache bus. A summary of the IBM Power3 hardware is given in Table 4.2.

<table>
<thead>
<tr>
<th>Processor</th>
<th>IBM Power3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock Speed</td>
<td>222 MHz</td>
</tr>
<tr>
<td>Peak Performance</td>
<td>888 MFlops</td>
</tr>
<tr>
<td>Primary Data Cache</td>
<td>64 KB (128-Way Set Associative)</td>
</tr>
<tr>
<td>Secondary Data Cache</td>
<td>4 MB (Direct Mapped)</td>
</tr>
<tr>
<td>Operating System</td>
<td>AIX 4.3.3</td>
</tr>
</tbody>
</table>

Table 4.2: IBM Power3 Processor Characteristics.

4.2.3 The Intel Pentium-III Processor

We used three Intel Pentium-III [32] PCs at the University of Houston to test the performance of our library on the Intel family of processors. The three processors used were the Pentium III–500, Pentium III–550 Xeon and the Pentium III–600 Coppermine processors. The Intel Pentium-III is a 32-bit CISC (Complex Instruction Set Computer) microprocessor which has a 32 KB non-blocking, level-one cache of which 16 KB is reserved as instruction cache and 16 KB is used as data cache. The processor has a dedicated 64-bit cache bus. It also has a 512 KB unified, non-blocking, level-two cache. The speed of the level-two cache scales with the processor core frequency. The Intel Pentium-III 500 has one pipelined FPU for supporting 32-bit and 64-bit arithmetic. The processor has 8 registers, 32 bit wide each. The processors we used had a memory of 128 MB and were using the Linux 2.2.14-5.0 operating system. A summary of the Intel Pentium-III 500 hardware is given in
Table 4.3.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel Pentium-III</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock Speed</td>
<td>500 MHz</td>
</tr>
<tr>
<td>Peak Performance</td>
<td>500 MFlops</td>
</tr>
<tr>
<td>Primary Data Cache</td>
<td>16 KB (32 Bytes)</td>
</tr>
<tr>
<td>Secondary Data Cache</td>
<td>512 KB</td>
</tr>
<tr>
<td>Instruction Cache</td>
<td>16 KB</td>
</tr>
<tr>
<td>Cache Type</td>
<td>8-Way Set-Associative</td>
</tr>
<tr>
<td>Operating System</td>
<td>Linux 2.2.14-5.0</td>
</tr>
</tbody>
</table>

Table 4.3: Intel Pentium-III Processor Characteristics.

4.3 Analysis of FFT Codelet Algorithms

In this section we present results for the selection of algorithms for the codelets of the UHRFFT library. Different algorithms are used for codelets of different sizes. The algorithm resulting in the fewest operations for a given codelet is chosen. Then, we verify that in fact the selected algorithm also result in the fewest instruction cycles when the codelets are compiled with the gnu C compiler version 2.7.23.

In Table 4.4 we compare the number of floating-point operations for the Mixed-Radix, Split-Radix and PFA algorithms for forward transforms on various transform sizes. In all of the cases, Rader’s algorithm is used to generate codelets of prime size transforms. The Split-Radix algorithm results in a reduction of operations for transform sizes that are a power of 2. The reductions can be seen for sizes 16 and above ($N = 16, 32, 64$ etc.). We have shown the number of arithmetic operations for each algorithm for the cases where they have been applied to generate the codelets. The reduction in arithmetic operations for the Mixed-Radix + PFA
and Mixed-Radix + PFA + Split-Radix compared to the Mixed-Radix alone is shown in parenthesis. The algorithms chosen for the UHRFFT library is specified in bold font in Table 4.4.

<table>
<thead>
<tr>
<th>Transform Size</th>
<th>Rader+Mix-Radix(MR)</th>
<th>Rader+MR+PFA</th>
<th>Rader+MR+PFA+Split-Radix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Adds</td>
<td>Mults</td>
<td>Adds(red.)</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>6</td>
<td>14( 2)</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>24</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>22</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>38</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>45</td>
<td>24</td>
<td>34( 11)</td>
</tr>
<tr>
<td>11</td>
<td>60</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>54</td>
<td>24</td>
<td>46( 8)</td>
</tr>
<tr>
<td>13</td>
<td>84</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>83</td>
<td>60</td>
<td>62( 21)</td>
</tr>
<tr>
<td>15</td>
<td>98</td>
<td>68</td>
<td>82( 16)</td>
</tr>
<tr>
<td>16</td>
<td>78</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>144</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>119</td>
<td>78</td>
<td>98( 21)</td>
</tr>
<tr>
<td>19</td>
<td>180</td>
<td>162</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>131</td>
<td>76</td>
<td>100( 31)</td>
</tr>
<tr>
<td>21</td>
<td>172</td>
<td>140</td>
<td>134( 38)</td>
</tr>
<tr>
<td>22</td>
<td>197</td>
<td>160</td>
<td>142( 55)</td>
</tr>
<tr>
<td>32</td>
<td>238</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>654</td>
<td>276</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>1678</td>
<td>788</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.4: Arithmetic Operations Count for UHRFFT Generated Codelets (Forward).

The codelets generated by the automatic code generator are converted to executable binary code by the compiler for the target platform. The performance of the codelets depend heavily on the output generated by the compiler on different architectures.

In Table 4.5 we compare the number of cycles required for different codelets when
compiled with the `gcc` compiler version 2.7.2.3 for Intel Pentium-III processors running Linux 2.2.14-5.0. Several compiler options were tested as described in section 4.4 to get the best result for the codelets. The `O1` `-fomit-frame-pointer` optimization option yielded the best results. The Performance Counter Library (PCL) [33] was used to evaluate the compiler output for the number of cycles, the number of floating-point instructions and the total number of instructions. The number of cycles are not exactly reproducible, most likely due to stalls that may vary from one execution to another depending on the system load at the time of execution. Each cycle count in Table 4.5 represents the average over 20 million executions, which gives fairly reasonable statistics for the number of cycles. The total number of instructions per floating-point instruction ranges between 2.5 – 3.5 while the total number of instructions per cycle is in the range 0.4 – 0.9, except for the codelet of size 64, for which the number of cycles per instruction is about 0.5 for the Split–Radix algorithm.

From Table 4.5 it is evident that the number of cycles required to execute codelets is reduced when the number of floating-point operations is reduced. Thus, for the codelets, minimizing the number of arithmetic operations indeed also maximizes the performance for the Intel Pentium-III processors. For the other architectures we studied the results in terms of the relationship between minimizing floating-point operations and minimizing the execution time were similar.
<table>
<thead>
<tr>
<th>Transform Size</th>
<th>Rader+Mix-Radix(MR)</th>
<th>Rader +MR+PFA</th>
<th>Rader+MR+PFA +Split-Radix</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>33</td>
<td>2</td>
<td>31</td>
</tr>
<tr>
<td>3</td>
<td>45</td>
<td>6</td>
<td>40</td>
</tr>
<tr>
<td>4</td>
<td>55</td>
<td>6</td>
<td>46</td>
</tr>
<tr>
<td>5</td>
<td>83</td>
<td>18</td>
<td>76</td>
</tr>
<tr>
<td>6</td>
<td>102</td>
<td>24</td>
<td>103</td>
</tr>
<tr>
<td>7</td>
<td>143</td>
<td>42</td>
<td>151</td>
</tr>
<tr>
<td>8</td>
<td>140</td>
<td>25</td>
<td>150</td>
</tr>
<tr>
<td>9</td>
<td>231</td>
<td>66</td>
<td>277</td>
</tr>
<tr>
<td>10</td>
<td>274</td>
<td>69</td>
<td>400</td>
</tr>
<tr>
<td>11</td>
<td>319</td>
<td>110</td>
<td>517</td>
</tr>
<tr>
<td>12</td>
<td>293</td>
<td>69</td>
<td>448</td>
</tr>
<tr>
<td>13</td>
<td>440</td>
<td>156</td>
<td>890</td>
</tr>
<tr>
<td>14</td>
<td>487</td>
<td>143</td>
<td>885</td>
</tr>
<tr>
<td>15</td>
<td>558</td>
<td>166</td>
<td>971</td>
</tr>
<tr>
<td>16</td>
<td>431</td>
<td>94</td>
<td>812</td>
</tr>
<tr>
<td>17</td>
<td>722</td>
<td>272</td>
<td>1787</td>
</tr>
<tr>
<td>18</td>
<td>710</td>
<td>197</td>
<td>1232</td>
</tr>
<tr>
<td>19</td>
<td>865</td>
<td>342</td>
<td>2335</td>
</tr>
<tr>
<td>20</td>
<td>723</td>
<td>207</td>
<td>1403</td>
</tr>
<tr>
<td>21</td>
<td>960</td>
<td>312</td>
<td>1802</td>
</tr>
<tr>
<td>22</td>
<td>1063</td>
<td>357</td>
<td>2549</td>
</tr>
<tr>
<td>32</td>
<td>1233</td>
<td>308</td>
<td>2267</td>
</tr>
<tr>
<td>64</td>
<td>3357</td>
<td>896</td>
<td>7590</td>
</tr>
<tr>
<td>128</td>
<td>8567</td>
<td>2395</td>
<td>27563</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Rader+MR+PFA</th>
<th>Rader+MR+PFA+Split-Radix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total Instr.</td>
<td>Total FP Instr.</td>
</tr>
<tr>
<td></td>
<td>833</td>
<td>198</td>
</tr>
</tbody>
</table>

Table 4.5: Instruction and Cycle Counts for UHRFFT Generated Codelets on Intel Pentium-III.
4.4 Compiler Analysis

As explained in the previous chapter, the main idea behind the code-generator is to develop a library which can be used over many different platforms. The output of the code-generator is C code which is subject to compilation by any ANSI C compliant compiler to produce the executable version of the library. This step can adversely affect the performance of the library due to the uncertainties introduced by the compiler. Ideally, squeezing the maximum performance from the compiler would yield the ‘best’ library of codelets. The recommended compiler options on each architecture may not be the most apt for generating the ‘best’ library of codelets and thus there is a need for compiler analysis where the compiler components (capabilities) are addressed. The aim is to have compiler analysis done automatically by the installer of the library. This would result in having a performance tuned installation of the library with the best set of options over the entire domain of parameters, or a subset of the parameters, being defined.

4.4.1 Compiler Performance Measures

We start by defining a 4-dimensional performance matrix for the library of FFT codelets $P = (p(i, o, n, k))$. The four parameters used are:

- input stride, $i \in S$
- output stride, $o \in S$
- transform size, $n \in N$
- compiler options set, $k \in O$
where \( S = \{ s_1, s_2, \ldots, s_N \} \) is the set of all input and output strides with \( 2^0 \leq s_N \leq 2^{16} \), \( N = \{ n_1, n_2, \ldots, n_N \} \) is the set of transform sizes and \( O = \{ o_1, o_2, \ldots, o_N \} \) is the collection of compiler option sets. Each element in \( P \) is the performance in MFLOPS over the combination of all parameters.

Next, we define a matrix of relative performance values

\[
Pr = (pr(i, o, n, k)),
\]

where

\[
pr(i, o, n, k) = p(i, o, n, k)/\max_p(i, o, n, k).
\]

This represents a normalized performance over the range. We use \( Pr \) to define the following compiler performance measures which help in determining which set of compiler options would be optimal to use. They are:

- **Option Performance Index**: This measure tells if a given set of compiler options is good over the parameter range defined by a combination of input and output strides and transform size. It is calculated as follows:

\[
IO_k = 1/(N_n \times (N_o)^2) \sum_{i, o, n} pr(i, o, n, k).
\]

The parameters used are weighted equally in the calculation of this measure. Good performance is typically anything over 0.9 meaning the option combination yeilds 90% of the ‘best’ performance over the entire parameter range.

- **Size/Option Performance Index**: This measure tells if a given set of compiler options is good over a specific range of parameter combinations. The range is determined by the codelet size. This measure is useful because there may not be any one option that shows good performance with the previous
measure (across the whole parameter range). In this case, performance can be made better by taking different options over subsets of the parameter range. It is calculated as follows:

\[ IOS_{k,n} = 1/(N_s)^2 \sum_{i:o} pr(i, o, n, k). \]

### 4.4.2 Compiler Performance Results

Compiler analysis was done on all platforms that the library was tested. On the Pentium III’s, the analysis was done using the gcc compiler version 2.2.14-5.0. Analysis was done for all possible combinations of the following optimization levels and options:

- Optimization levels: O1 - O5
- Option: -fomit-frame-pointer
- Option: -malign-double

The results are shown in the following charts.
Figure 4.1: Option Performance Index ($IO_k$) on the Intel Pentium III’s.

Figure 4.2: Performance for 2–Point Transform.
Figure 4.3: Performance for 4-Point Transform.

Figure 4.4: Performance for 8-Point Transform.
Relative performance for different compiler options and for 16-Point Transform

**Figure 4.5: Performance for 16-Point Transform.**

Relative performance for different compiler options and for 32-Point Transform

**Figure 4.6: Performance for 32-Point Transform.**
Figure 4.7: Performance for 64-Point Transform.

Figure 4.8: Performance for 128-Point Transform.
The option combination set of \textit{O1 -fomit-frame-pointer} resulted in the best performance for both of the defined measures, i.e., the Option Performance Index and the Size/Option Performance Index, for both the forward and inverse transforms.

The goc compiler used on the SGI Origin 2000 was the MIPSpro Compiler \textit{version 7.3.1m} and the options tested were:

- Optimization levels \texttt{O0 - O3}
- \texttt{-mips4}
- \texttt{-align64}

The best performance for the Option Performance Index was obtained by the option combination \texttt{-O2 -mips4} as shown in Figure 4.9.

![Figure 4.9: Option Performance Index (IO_k) on the SGI Origin 2000.](image)

The rest of the results are shown in Table 4.6.
<table>
<thead>
<tr>
<th>Transform Size</th>
<th>(Option Set)_{max}</th>
<th>#(IO_k)_{max}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>-O2 -mips4</td>
<td>97.293%</td>
</tr>
<tr>
<td>2</td>
<td>-O2 -mips4</td>
<td>99.841%</td>
</tr>
<tr>
<td>4</td>
<td>-O2 -mips4</td>
<td>98.668%</td>
</tr>
<tr>
<td>8</td>
<td>-O2 -mips4</td>
<td>96.556%</td>
</tr>
<tr>
<td>16</td>
<td>-O2 -mips4</td>
<td>95.595%</td>
</tr>
<tr>
<td>32</td>
<td>-O2 -mips4</td>
<td>97.195%</td>
</tr>
<tr>
<td>64</td>
<td>-O2 -mips4</td>
<td>97.596%</td>
</tr>
<tr>
<td>128</td>
<td>-O2 -mips4</td>
<td>98.633%</td>
</tr>
</tbody>
</table>

Table 4.6: Compiler Performance on the SGI Origin 2000.

The C for AIX Compiler, version 5 was installed on the IBM NightHawk nodes and the options tested were:

- Optimization levels O2 - O5
- -qarch=pwr3
- -qalign=power

The best performance for the Option Performance Index (IO_k) was obtained by the option combination -O2 -qarch=pwr3 as shown in Figure 4.10.

The rest of the results are shown in Table 4.7.
Figure 4.10: Option Performance Index \( (IO_k) \) on the IBM Power3 Processor.

<table>
<thead>
<tr>
<th>Transform Size</th>
<th>((OptionSet)_{\text{max}})</th>
<th># ((IO_k)_{\text{max}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>-O2 -qarch=pwr3</td>
<td>95.283%</td>
</tr>
<tr>
<td>2</td>
<td>-O2 -qarch=pwr3</td>
<td>99.096%</td>
</tr>
<tr>
<td>4</td>
<td>-O2 -qarch=pwr3</td>
<td>98.146%</td>
</tr>
<tr>
<td>8</td>
<td>-O2 -qarch=pwr3</td>
<td>97.271%</td>
</tr>
<tr>
<td>16</td>
<td>-O2 -qarch=pwr3</td>
<td>97.485%</td>
</tr>
<tr>
<td>32</td>
<td>-O2 -qarch=pwr3</td>
<td>97.325%</td>
</tr>
<tr>
<td>64</td>
<td>-O2 -qarch=pwr3</td>
<td>98.291%</td>
</tr>
<tr>
<td>128</td>
<td>-O2 -qarch=pwr3</td>
<td>98.645%</td>
</tr>
</tbody>
</table>

Table 4.7: Compiler Performance on the IBM NightHawk Nodes.
4.5 Performance of the FFT Codelets

We benchmarked the codelets for transforms of size 128 or less for a range of input and output strides on the data to evaluate the performance of the FFT codelets in the UHRFFT library. For our benchmarking purposes, we chose strides that are powers of 2. This choice was due to the fact that the sizes of cache lines, cache and memory are usually equal to some power of 2 and we would, therefore, be able to catch some of the worst performance behavior this way. Each of the reported data items is the average of multiple runs over a period of at least one second to ensure that errors that would arise due to clock resolution are eliminated from the results.

We see that for all platforms considered, the performance of the codelets decreases considerably for large data strides. This happens due to the effect of cache trashing [34]. This phenomenon occurs when two or more data elements required by a codelet are mapped to the same physical block in cache. Thus, the loading of one data element into the cache results in the expulsion of the other. Cache trashing occurs most frequently for strides of data that are powers of two and it also depends on the type of cache used by the architecture. For all the architectures, the performance decrease due to cache trashing occurs when

\[
datapoint\_size \times stride \times \frac{codelet\_size}{2} > \frac{cache\_size}{Associativity},
\]

where \( datapoint\_size \) is the size of one data element (8 Bytes for real data), \( codelet\_size \) is the number of data elements being transformed by the codelet, \( cache\_size \) is the total size of the cache in Bytes, \( stride \) is the data access stride, and \( Associativity \) is the type of cache being used by the architecture.
4.5.1 The SGI R10000

In this section, we present the performance of the UHRFFT codelets for power of two input and output strides on the SGI R10000 processor. The performance is seen to be symmetric with respect to the input and output strides implying that the reads and writes from cache affect the performance in a similar fashion. Both the level-one and level-two caches of the SGI R10000 processor are two-way set-associative implying that a data point in memory may be mapped onto one of two physical blocks in cache. The level-one cache is of size 32 KB and the level-two cache is of size 4 MB. A sharp drop in performance due to level-one cache trashing for the codelets occurs when the stride is

\[
\text{stride} > \frac{32KB}{8 \times 2 \times \frac{\text{codelet size}}{2}} = \frac{2^{12}}{\text{codelet size}}
\]

due to level-one cache trashing and again when the stride is

\[
\text{stride} > \frac{4MB}{8 \times 2 \times \frac{\text{codelet size}}{2}} = \frac{2^{19}}{\text{codelet size}}
\]

due to level-two cache trashing. In Figure 4.21, the average performance for each codelet is given along with its range of performance over the combination of all input and output strides. Figure 4.22 shows the average time for each codelet to execute. The peak performance for each codelet is given in Table 4.8. The peak performance is achieved for strides that are smaller than the cache line size (128 Bytes) of the processor.
Figure 4.11: Performance for Codelets of Size 2 and Size 3 on a 250 MHz SGI R10000.

Figure 4.12: Performance for Codelets of Size 4 and Size 5 on a 250 MHz SGI R10000.
Figure 4.13: Performance for Codelets of Size 6 and Size 7 on a 250 MHz SGI R10000.

Figure 4.14: Performance for Codelets of Size 8 and Size 9 on a 250 MHz SGI R10000.
Figure 4.15: Performance for Codelets of Size 10 and Size 11 on a 250 MHz SGI R10000.

Figure 4.16: Performance for Codelets of Size 12 and Size 13 on a 250 MHz SGI R10000.
Figure 4.17: Performance for Codelets of Size 14 and Size 16 on a 250 MHz SGI R10000.

Figure 4.18: Performance for Codelets of Size 17 and Size 19 on a 250 MHz SGI R10000.
Figure 4.19: Performance for Codelets of Size 32 and Size 64 on a 250 MHz SGI R10000.

Figure 4.20: Performance for Codelet of Size 128 on a 250 MHz SGI R10000.
Figure 4.21: UHRFFT Codelet Efficiency on a 250 MHz SGI R10000.

Figure 4.22: UHRFFT Codelet Timings on a 250 MHz SGI R10000.
<table>
<thead>
<tr>
<th>Codelet size</th>
<th>Performance (MFLOPS)</th>
<th>Time (sec)</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>rfft2</td>
<td>89.26</td>
<td>5.60e-08</td>
<td>0.179</td>
</tr>
<tr>
<td>rfft3</td>
<td>148.53</td>
<td>8.00e-08</td>
<td>0.297</td>
</tr>
<tr>
<td>rfft4</td>
<td>249.91</td>
<td>8.00e-08</td>
<td>0.499</td>
</tr>
<tr>
<td>rfft5</td>
<td>250.11</td>
<td>1.16e-07</td>
<td>0.500</td>
</tr>
<tr>
<td>rfft6</td>
<td>242.26</td>
<td>1.60e-07</td>
<td>0.485</td>
</tr>
<tr>
<td>rfft7</td>
<td>227.37</td>
<td>2.16e-07</td>
<td>0.455</td>
</tr>
<tr>
<td>rfft8</td>
<td>312.39</td>
<td>1.92e-07</td>
<td>0.625</td>
</tr>
<tr>
<td>rfft9</td>
<td>236.76</td>
<td>3.01e-07</td>
<td>0.474</td>
</tr>
<tr>
<td>rfft10</td>
<td>302.25</td>
<td>2.75e-07</td>
<td>0.605</td>
</tr>
<tr>
<td>rfft11</td>
<td>233.10</td>
<td>4.08e-07</td>
<td>0.466</td>
</tr>
<tr>
<td>rfft12</td>
<td>363.22</td>
<td>2.96e-07</td>
<td>0.726</td>
</tr>
<tr>
<td>rfft13</td>
<td>222.63</td>
<td>5.40e-07</td>
<td>0.445</td>
</tr>
<tr>
<td>rfft14</td>
<td>326.49</td>
<td>4.08e-07</td>
<td>0.653</td>
</tr>
<tr>
<td>rfft16</td>
<td>393.17</td>
<td>4.07e-07</td>
<td>0.786</td>
</tr>
<tr>
<td>rfft17</td>
<td>212.81</td>
<td>8.16e-07</td>
<td>0.426</td>
</tr>
<tr>
<td>rfft19</td>
<td>201.70</td>
<td>1.00e-06</td>
<td>0.403</td>
</tr>
<tr>
<td>rfft32</td>
<td>364.37</td>
<td>1.10e-06</td>
<td>0.729</td>
</tr>
<tr>
<td>rfft64</td>
<td>284.91</td>
<td>3.37e-06</td>
<td>0.570</td>
</tr>
<tr>
<td>rfft128</td>
<td>256.99</td>
<td>8.72e-06</td>
<td>0.514</td>
</tr>
</tbody>
</table>

Table 4.8: Peak Performance of UHRFFT Codelets on a 250 MHz SGI R10000.
4.5.2 The Intel Pentium

The Intel Pentium–III 500

In this section, we present the performance of the UHRFFT codelets on the Intel Pentium–III 500 processor. The size of the instruction cache and data cache on the Pentiums is 16 KB and is four-way set associative. Therefore, for the smaller codelets (Size <= 4), there is no cache trashing as the codelet and the data both fit in the primary instruction and data cache respectively. For the larger codelets, the performance is affected adversely by the cache trashing phenomenon and also for the fact there are a few registers (8) that affect codelets with a large number of data points. A sharp drop in performance due to level-one cache trashing occurs when

\[ \text{stride} > \frac{16KB}{8 \times 4 \times \text{codelet size}} = \frac{2^{10}}{\text{codelet size}}. \]

The performance drop attributed to level-two cache trashing occurs when

\[ \text{stride} > \frac{512KB}{8 \times 4 \times \text{codelet size}} = \frac{2^{15}}{\text{codelet size}}. \]

Furthermore, the Pentium–III 500 processor employs a mechanism whereby reads from memory are done in a speculative manner (pre-fetching). This reduces the cache miss rate for reads as compared to writes which are directly written to memory on a write miss to cache. Therefore, the performance drop due to large data output strides is greater than that due to large input strides. In Figure 4.27, the average performance for each codelet is given along with its range of performance over all possible combinations of input and output strides. Figure 4.28 shows the average time for each codelet to execute. The peak performance for each codelet is given in Table 4.9.
Radix-2 Performance avg. = 109.4273 (UHRFFT on the 500 MHz Pentium III)

Radix-4 Performance avg. = 251.0783 (UHRFFT on the 500 MHz Pentium III)

Radix-8 Performance avg. = 217.5891 (UHRFFT on the 500 MHz Pentium III)

Radix-16 Performance avg. = 217.4936 (UHRFFT on the 500 MHz Pentium III)

Figure 4.23: Performance for Codelets of Size 2 and Size 4 on a 500 MHz Pentium III.

Figure 4.24: Performance for Codelets of Size 8 and Size 16 on a 500 MHz Pentium III.
Figure 4.25: Performance for Codelets of Size 32 and Size 64 on a 500 MHz Pentium III.

Figure 4.26: Performance for Codelet of Size 128 on a 500 MHz Pentium III.
Figure 4.27: UHRFFT Codelet Efficiency on a 500 MHz Pentium III.

Figure 4.28: UHRFFT Codelet Timings on a 500 MHz Pentium III.
<table>
<thead>
<tr>
<th>Codelet size</th>
<th>Performance (MFlops)</th>
<th>Time (sec)</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>rfft2</td>
<td>110.11</td>
<td>4.54e-08</td>
<td>0.220</td>
</tr>
<tr>
<td>rfft3</td>
<td>172.67</td>
<td>6.88e-08</td>
<td>0.345</td>
</tr>
<tr>
<td>rfft4</td>
<td>278.13</td>
<td>7.19e-08</td>
<td>0.556</td>
</tr>
<tr>
<td>rfft5</td>
<td>256.85</td>
<td>1.13e-07</td>
<td>0.514</td>
</tr>
<tr>
<td>rfft6</td>
<td>273.06</td>
<td>1.42e-07</td>
<td>0.546</td>
</tr>
<tr>
<td>rfft7</td>
<td>233.95</td>
<td>2.10e-07</td>
<td>0.468</td>
</tr>
<tr>
<td>rfft8</td>
<td>368.10</td>
<td>1.63e-07</td>
<td>0.736</td>
</tr>
<tr>
<td>rfft9</td>
<td>226.42</td>
<td>3.15e-07</td>
<td>0.453</td>
</tr>
<tr>
<td>rfft10</td>
<td>311.04</td>
<td>2.67e-07</td>
<td>0.622</td>
</tr>
<tr>
<td>rfft11</td>
<td>184.01</td>
<td>5.17e-07</td>
<td>0.368</td>
</tr>
<tr>
<td>rfft12</td>
<td>368.32</td>
<td>2.92e-07</td>
<td>0.737</td>
</tr>
<tr>
<td>rfft13</td>
<td>155.58</td>
<td>7.73e-07</td>
<td>0.311</td>
</tr>
<tr>
<td>rfft14</td>
<td>266.51</td>
<td>5.00e-07</td>
<td>0.533</td>
</tr>
<tr>
<td>rfft16</td>
<td>390.24</td>
<td>4.10e-07</td>
<td>0.780</td>
</tr>
<tr>
<td>rfft17</td>
<td>126.80</td>
<td>1.37e-06</td>
<td>0.254</td>
</tr>
<tr>
<td>rfft19</td>
<td>93.85</td>
<td>2.15e-06</td>
<td>0.188</td>
</tr>
<tr>
<td>rfft32</td>
<td>372.57</td>
<td>1.07e-06</td>
<td>0.745</td>
</tr>
<tr>
<td>rfft64</td>
<td>345.70</td>
<td>2.78e-06</td>
<td>0.691</td>
</tr>
<tr>
<td>rfft128</td>
<td>106.16</td>
<td>2.11e-05</td>
<td>0.212</td>
</tr>
</tbody>
</table>

Table 4.9: Peak Performance of UHRFFT Codelets on a 500 MHz Intel Pentium III.
The Intel Pentium-III 550 Xeon

We also evaluated the performance of the UHRFFT codelets on an Intel Pentium-III 550 Xeon Processor. The Xeon processor also had a 32 KB non-blocking level-one cache (16KB instruction cache and 16 KB data cache) and a 512KB level-two cache. A sharp drop in performance due to level-one cache trashing occurs when

$$ \text{stride} > \frac{16KB}{8 \times 4 \times \frac{\text{codelet size}}{2}} = \frac{2^{10}}{\text{codelet size}}. $$

The performance drop attributed to level-two cache trashing occurs when

$$ \text{stride} > \frac{512KB}{8 \times 4 \times \frac{\text{codelet size}}{2}} = \frac{2^{15}}{\text{codelet size}}. $$

The Pentium-III 550 Xeon also executes reads from memory in a speculative manner, thus reducing the number of cache misses for reads. This is reflected in a larger performance drop due to large output strides as opposed to that due to large input strides. Figure 4.33 gives the average performance for each codelet along with its range of performance over all possible combinations of input and output strides while Figure 4.34 shows the average time for each codelet to execute. The peak performance for each codelet is given in Table 4.10.
Figure 4.29: Performance for Codelets of Size 2 and Size 4 on a 550 MHz Pentium III Xeon.

Figure 4.30: Performance for Codelets of Size 8 and Size 16 on a 550 MHz Pentium III Xeon.
Figure 4.31: Performance for Codelets of Size 32 and Size 64 on a 550 MHz Pentium III Xeon.

Figure 4.32: Performance for Codelet of Size 128 on a 550 MHz Pentium III Xeon.
Figure 4.33: UHRFFT Codelet Efficiency on a 550 MHz Pentium III Xeon.

Figure 4.34: UHRFFT Codelet Timings on a 550 MHz Pentium III Xeon.
<table>
<thead>
<tr>
<th>Codelet size</th>
<th>Performance (MFlops)</th>
<th>Time (sec)</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>rfft2</td>
<td>120.61</td>
<td>4.15e-08</td>
<td>0.219</td>
</tr>
<tr>
<td>rfft3</td>
<td>191.11</td>
<td>6.22e-08</td>
<td>0.347</td>
</tr>
<tr>
<td>rfft4</td>
<td>303.26</td>
<td>6.60e-08</td>
<td>0.551</td>
</tr>
<tr>
<td>rfft5</td>
<td>275.23</td>
<td>1.05e-07</td>
<td>0.500</td>
</tr>
<tr>
<td>rfft6</td>
<td>295.99</td>
<td>1.31e-07</td>
<td>0.538</td>
</tr>
<tr>
<td>rfft7</td>
<td>250.66</td>
<td>1.96e-07</td>
<td>0.456</td>
</tr>
<tr>
<td>rfft8</td>
<td>413.79</td>
<td>1.45e-07</td>
<td>0.752</td>
</tr>
<tr>
<td>rfft9</td>
<td>252.03</td>
<td>2.83e-07</td>
<td>0.458</td>
</tr>
<tr>
<td>rfft10</td>
<td>351.90</td>
<td>2.36e-07</td>
<td>0.640</td>
</tr>
<tr>
<td>rfft11</td>
<td>205.03</td>
<td>4.64e-07</td>
<td>0.373</td>
</tr>
<tr>
<td>rfft12</td>
<td>407.38</td>
<td>2.64e-07</td>
<td>0.741</td>
</tr>
<tr>
<td>rfft13</td>
<td>171.81</td>
<td>7.00e-07</td>
<td>0.312</td>
</tr>
<tr>
<td>rfft14</td>
<td>298.78</td>
<td>4.46e-07</td>
<td>0.543</td>
</tr>
<tr>
<td>rfft16</td>
<td>461.10</td>
<td>3.47e-07</td>
<td>0.838</td>
</tr>
<tr>
<td>rfft17</td>
<td>137.87</td>
<td>1.26e-06</td>
<td>0.251</td>
</tr>
<tr>
<td>rfft19</td>
<td>123.86</td>
<td>1.63e-06</td>
<td>0.225</td>
</tr>
<tr>
<td>rfft32</td>
<td>406.04</td>
<td>9.85e-07</td>
<td>0.738</td>
</tr>
<tr>
<td>rfft64</td>
<td>377.95</td>
<td>2.54e-06</td>
<td>0.687</td>
</tr>
<tr>
<td>rfft128</td>
<td>149.33</td>
<td>1.50e-05</td>
<td>0.272</td>
</tr>
</tbody>
</table>

Table 4.10: Peak Performance of UHRFFT Codelets on a 550 MHz Intel Pentium III Xeon.
The Intel Pentium–III 600

We also evaluated the performance of the UHRFFT codelets on an Intel Pentium–III 600 Coppermine Processor. The Coppermine processor also had a 32 KB non-blocking level-one cache (16KB instruction cache and 16 KB data cache) and a 512KB level-two cache. A sharp drop in performance due to level-one cache trashing occurs when

$$\text{stride} > \frac{16KB}{8 \times 4 \times \frac{\text{codelet size}}{2}} = \frac{2^{10}}{\text{codelet size}}.$$

The performance drop attributed to level-two cache trashing occurs when

$$\text{stride} > \frac{512KB}{8 \times 4 \times \frac{\text{codelet size}}{2}} = \frac{2^{15}}{\text{codelet size}}.$$

The Pentium–III 600 also executes reads from memory in a speculative manner, thus reducing the number of cache misses for reads. This is reflected in a larger performance drop due to large output strides as opposed to that due to large input strides. Figure 4.39 gives the average performance for each codelet along with its range of performance over all possible combinations of input and output strides while Figure 4.40 shows the average time for each codelet to execute. The peak performance for each codelet is given in Table 4.11.
Figure 4.35: Performance for Codelets of Size 2 and Size 4 on a 600 MHz Pentium III.

Figure 4.36: Performance for Codelets of Size 8 and Size 16 on a 600 MHz Pentium III.
Figure 4.37: Performance for Codelets of Size 32 and Size 64 on a 600 MHz Pentium III.

Figure 4.38: Performance for Codelet of Size 128 on a 600 MHz Pentium III.
Figure 4.39: UHRFFT Codelet Efficiency on a 600 MHz Pentium III.

Figure 4.40: UHRFFT Codelet Timings on a 600 MHz Pentium III.
<table>
<thead>
<tr>
<th>Codelet size</th>
<th>Performance (MFlops)</th>
<th>Time (sec)</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>rfft2</td>
<td>132.68</td>
<td>3.77e-08</td>
<td>0.221</td>
</tr>
<tr>
<td>rfft3</td>
<td>206.79</td>
<td>5.75e-08</td>
<td>0.345</td>
</tr>
<tr>
<td>rfft4</td>
<td>330.58</td>
<td>6.05e-08</td>
<td>0.551</td>
</tr>
<tr>
<td>rfft5</td>
<td>305.55</td>
<td>9.50e-08</td>
<td>0.509</td>
</tr>
<tr>
<td>rfft6</td>
<td>328.60</td>
<td>1.18e-07</td>
<td>0.548</td>
</tr>
<tr>
<td>rfft7</td>
<td>274.46</td>
<td>1.79e-07</td>
<td>0.457</td>
</tr>
<tr>
<td>rfft8</td>
<td>441.18</td>
<td>1.36e-07</td>
<td>0.735</td>
</tr>
<tr>
<td>rfft9</td>
<td>269.14</td>
<td>2.65e-07</td>
<td>0.449</td>
</tr>
<tr>
<td>rfft10</td>
<td>370.75</td>
<td>2.24e-07</td>
<td>0.618</td>
</tr>
<tr>
<td>rfft11</td>
<td>220.73</td>
<td>4.31e-07</td>
<td>0.368</td>
</tr>
<tr>
<td>rfft12</td>
<td>444.42</td>
<td>2.42e-07</td>
<td>0.741</td>
</tr>
<tr>
<td>rfft13</td>
<td>187.49</td>
<td>6.41e-07</td>
<td>0.312</td>
</tr>
<tr>
<td>rfft14</td>
<td>321.88</td>
<td>4.14e-07</td>
<td>0.536</td>
</tr>
<tr>
<td>rfft15</td>
<td>336.80</td>
<td>4.35e-07</td>
<td>0.561</td>
</tr>
<tr>
<td>rfft16</td>
<td>465.12</td>
<td>3.44e-07</td>
<td>0.775</td>
</tr>
<tr>
<td>rfft17</td>
<td>151.06</td>
<td>1.15e-06</td>
<td>0.252</td>
</tr>
<tr>
<td>rfft18</td>
<td>310.67</td>
<td>6.04e-07</td>
<td>0.512</td>
</tr>
<tr>
<td>rfft20</td>
<td>406.96</td>
<td>5.31e-07</td>
<td>0.678</td>
</tr>
<tr>
<td>rfft32</td>
<td>446.31</td>
<td>8.96e-07</td>
<td>0.744</td>
</tr>
<tr>
<td>rfft64</td>
<td>413.79</td>
<td>2.32e-06</td>
<td>0.690</td>
</tr>
<tr>
<td>rfft128</td>
<td>229.69</td>
<td>9.75e-06</td>
<td>0.383</td>
</tr>
</tbody>
</table>

Table 4.11: Peak Performance of UHRFFT Codelets on a 600 MHz Intel Pentium III.
4.5.3 The IBM Power3

The Power3 has a primary data cache of 64 KB which is 128–way set associative and a secondary data cache of 4 MB which is direct mapped. Peak performance on this processor is achieved at 888 MFLOPS.

Figure 4.45 gives the average performance for each codelet along with its range of performance over all possible combinations of input and output strides while Figure 4.46 shows the average time for each codelet to execute. The peak performance for each codelet is given in Table 4.12.

Figure 4.41: Performance for Codelets of Size 2 and Size 4 on a 222 MHz Power3.
Figure 4.42: Performance for Codelets of Size 8 and Size 16 on a 222 MHz Power3.

Figure 4.43: Performance for Codelets of Size 32 and Size 64 on a 222 MHz Power3.
Figure 4.44: Performance for Codelet of Size 128 on a 222 MHz Power3.

Figure 4.45: UHRFFT Codelet Efficiency on a 222 MHz Power3.
<table>
<thead>
<tr>
<th>Codelet size</th>
<th>Performance (MFlops)</th>
<th>Time (sec)</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>rfft2</td>
<td>62.50</td>
<td>8.00e-08</td>
<td>0.070</td>
</tr>
<tr>
<td>rfft3</td>
<td>113.70</td>
<td>1.05e-07</td>
<td>0.128</td>
</tr>
<tr>
<td>rfft4</td>
<td>202.58</td>
<td>9.87e-08</td>
<td>0.228</td>
</tr>
<tr>
<td>rfft5</td>
<td>176.98</td>
<td>1.64e-07</td>
<td>0.199</td>
</tr>
<tr>
<td>rfft6</td>
<td>255.09</td>
<td>1.52e-07</td>
<td>0.287</td>
</tr>
<tr>
<td>rfft7</td>
<td>169.41</td>
<td>2.90e-07</td>
<td>0.191</td>
</tr>
<tr>
<td>rfft8</td>
<td>315.79</td>
<td>1.90e-07</td>
<td>0.356</td>
</tr>
<tr>
<td>rfft9</td>
<td>185.26</td>
<td>3.85e-07</td>
<td>0.209</td>
</tr>
<tr>
<td>rfft10</td>
<td>269.48</td>
<td>3.08e-07</td>
<td>0.303</td>
</tr>
<tr>
<td>rfft11</td>
<td>203.74</td>
<td>4.67e-07</td>
<td>0.229</td>
</tr>
<tr>
<td>rfft12</td>
<td>384.10</td>
<td>2.80e-07</td>
<td>0.433</td>
</tr>
<tr>
<td>rfft13</td>
<td>200.44</td>
<td>6.00e-07</td>
<td>0.226</td>
</tr>
<tr>
<td>rfft14</td>
<td>233.78</td>
<td>5.70e-07</td>
<td>0.263</td>
</tr>
<tr>
<td>rfft16</td>
<td>368.51</td>
<td>4.34e-07</td>
<td>0.415</td>
</tr>
<tr>
<td>rfft17</td>
<td>179.66</td>
<td>9.67e-07</td>
<td>0.202</td>
</tr>
<tr>
<td>rfft19</td>
<td>166.76</td>
<td>1.21e-06</td>
<td>0.188</td>
</tr>
<tr>
<td>rfft32</td>
<td>389.38</td>
<td>1.03e-06</td>
<td>0.438</td>
</tr>
<tr>
<td>rfft64</td>
<td>314.28</td>
<td>3.05e-06</td>
<td>0.354</td>
</tr>
<tr>
<td>rfft128</td>
<td>246.64</td>
<td>9.08e-06</td>
<td>0.278</td>
</tr>
</tbody>
</table>

Table 4.12: Peak Performance of UHRFFT Codelets on a 222MHz Power3.
Figure 4.46: UHRFFT Codelet Timings on a 222 MHz Power3.

4.5.4 UHRFFT Codelet Efficiency comparison with UHFFT

An observation of the results shows that the theoretical peak performance is not achieved on all architectures. This theoretical peak is the maximum number of floating point operations (FLOPS) that the architecture is capable of performing. There are various factors that limit the ability of achieving this peak performance. They include:

- The floating point density of the codelets.
- Cache size of system.
- Cache addressability (associativity).
- Number of registers available.

For smaller sized codelets, the floating point density is low and as a result, performance is limited by this. As the codelet size increases, the floating point density
also increases (asymptotically) and since the code size is small enough to fit in the cache, the actual measured performance also increases. As the codelet size grows larger, the combined effects of cache size, cache associativity and number of registers becomes more pronounced. For the codelets which can fit in cache, the number of temporary variables that are needed in the computation may be larger than the number of registers available in the system. In such cases, the stack memory is used to store these temporary variables. This has the effect of lowering the floating point density of the codelets with respect to loads/stores and thus reduces the ability to achieve peak performance. For larger size codelets which are not able to fit in the instruction cache, off-chip memory is used and since the speed of this memory is slower than the processor speed, degradation in performance is inevitable.

These phenomena can be seen in the comparison of the performance of the UHRFFT (real) library with the performance of the UHFFT (complex) library. The floating point density of the codelets (with respect to loads/stores) in the two libraries is the same. i.e, \( O(5 \cdot N \log_2 N / 2 \cdot N) \) for the complex codelets and \( O(2.5 \cdot N \log_2 N / N) \) for the codelets for real input data. However, the size of a codelet of size \( N \) in the UHFFT library is twice the size of a codelet of size \( N \) in the UHRFFT library. For this reason, on systems which have a small size cache, comparable codelets (in terms of size) of the UHRFFT perform better than codelets of the UHFFT library since cache misses due to cache capacity and associativity affect them less. This can be observed in Figure 4.48. The spike in this chart is due to the fact that the transform of size 64 is the only transform from the UHRFFT library that is not able to fit in the cache and therefore, its performance is much worse than that of the smaller size transforms. In this one case, the UHFFT codelet performs much better than the UHRFFT codelet. On the other hand, on systems which have a relatively large
cache configuration, the codelets from the UHFFT perform better as compared to codelets from the UHRFFT library as observed in Figure 4.47. It can be observed from the figures that the prime size transforms exhibit some irregular behaviour. This is based on the fact that the scaling that is used for the performance is based on power 2 transforms, and therefore, for other size transforms, the ratio for the real to complex transforms does not favor these size transforms as can be seen in Table 2.1.

![Graph](image-url)

**Figure 4.47:** UHRFFT vs. UHFFT Codelet Efficiency comparison on a 222 MHz Power3.
Figure 4.48: UHRFFT vs. UHFFT Codelet Efficiency comparison on a 550 MHz Xeon Pentium III.

4.6 Analysis of Execution Strategy

In this section, we present the performance data for different execution plans for the UHRFFT. From the users point of view, the performance of the selected execution plan is what is most important for the performance of the application that uses the library. The execution times that are reported include the time for the computation of the transform using a particular execution plan but it excludes the time for computation of the twiddle factors and the time for generation of the execution plan.

It is impossible to list in a reasonable and meaningful way, in this thesis, the performance of all plans for all possible FFT sizes. Therefore, we illustrate the performance for some FFT sizes that are powers of 2 (as these are most commonly used in scientific applications). The performance of the different plans are presented in the order in which they are generated. Table 4.13 lists the performance of all plans for a real transform of size 256 for the SGI R100000 while tables 4.14, 4.15,
and 4.16 give the same information for the IBM Power3, Intel Pentium-III 500 and Intel Pentium-III 550 Xeon respectively. The other figures demonstrate how the performance varies for different plans on the different platforms and also show how important it is, in order to yield good performance, that the mechanism that selects the library execution plan choose the plan with the shortest execution time, since the next best plan may be significantly slower.

4.6.1 The SGI R10000

Herein, we present the SGI R10000 processor performance results of certain execution plans of the UHRFFT for some transform sizes. For the smaller transform sizes (i.e., 2, 4, 8, 16, 32 and 64), the second best plan yields a performance that is significantly lower than that of the best plan. For example, for an FFT of size 16, the second best plan yields a performance of 100 MFLOPS which represents 27% of the performance of the best plan (370 MFLOPS). For the larger size transforms, the performance drop due to the selection of an execution plan is not as significant as that of the smaller transforms. For instance, for a real transform of size 256, the performance drops from a peak of 191.05 MFLOPS for the plan with the smallest execution time to 84.68 MFLOPS for the worst plan. This represents a drop of 56% in performance between the best plan and the worst plan. The second best plan, in this case, yields 95% of the performance of the best plan. This phenomenon can be attributed to the fact that for the smaller size FFT’s, the best performance is achieved by the plan composed of the codelet of that transform size. This plan performs much better than other plans that involve overhead in combining different codelets to form the transform. On the other hand, for FFT sizes that are greater than a size of 128, the low performance can be attributed to the fact that there were no codelets larger than size 128 at the time.
of benchmarking the library and thus the overhead involved in combining different codelets to form the larger transforms was inevitable. In Figure 4.58 we present the performance of the best performance plans for the UHRFFT library for power of two sizes up to 1 MB. This performance is what is available to the applications that use the library for these given sizes. We also compare this performance to that yielded by the FFTW library [14]. From Figure 4.58, we can see that for the smaller size transforms (size ≤ 128), the UHRFFT library outperforms the FFTW library. For the larger transforms, the FFTW library performs slightly better than the UHRFFT. This is because the mechanism used to combine the smaller transforms to get the larger ones was not yet optimal at the time of benchmarking the library. We are currently working on this so as to produce more efficient transforms of larger sizes.

<table>
<thead>
<tr>
<th>Plan</th>
<th>Time (sec)</th>
<th>MFlops</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 64</td>
<td>2.99e-05</td>
<td>171.12</td>
<td>0.342</td>
</tr>
<tr>
<td>64 32</td>
<td>2.68e-05</td>
<td>191.05</td>
<td>0.382</td>
</tr>
<tr>
<td>32 16</td>
<td>2.79e-05</td>
<td>182.98</td>
<td>0.366</td>
</tr>
<tr>
<td>16 8</td>
<td>3.34e-05</td>
<td>152.97</td>
<td>0.306</td>
</tr>
<tr>
<td>8 4</td>
<td>4.36e-05</td>
<td>117.43</td>
<td>0.235</td>
</tr>
<tr>
<td>4 2</td>
<td>6.04e-05</td>
<td>84.68</td>
<td>0.169</td>
</tr>
</tbody>
</table>

Table 4.13: Execution Plan Performance for Size 256 FFTs on a 250 MHz SGI R10000.
Figure 4.49: Execution Plan Performance for Size 8 and Size 16 FFTs on a 250 MHz SGI R10000.

Figure 4.50: Execution Plan Performance for Size 32 and Size 64 FFTs on a 250 MHz SGI R10000.
Figure 4.51: Execution Plan Performance for Size 128 and Size 256 FFTs on a 250 MHz SGI R10000.

Figure 4.52: Execution Plan Performance for Size 512 and Size 1024 FFTs on a 250 MHz SGI R10000.
Figure 4.53: Execution Plan Performance for Size 2048 and Size 4096 FFTs on a 250 MHz SGI R10000.

Figure 4.54: Execution Plan Performance for Size 8192 and Size 16384 FFTs on a 250 MHz SGI R10000.
Figure 4.55: Execution Plan Performance for Size 32768 and Size 65536 FFTs on a 250 MHz SGI R10000.

Figure 4.56: Execution Plan Performance for Size 131072 and Size 262144 FFTs on a 250 MHz SGI R10000.
Figure 4.57: Execution Plan Performance for Size 524288 and Size 1048576 FFTs on a 250 MHz SGI R10000.

Figure 4.58: 250 MHz SGI R10000 Execution Plan Performance.
4.6.2 The IBM Power3

In Table 4.14, we list the performance of each execution plan for a real transform of size 256 on the 222MHz IBM Power3 processor. Peak Performance on the Power3 for this size is achieved at 241.36 MFLOPS and drops to 76.75 MFLOPS for the worst plan with respect to execution time. The drop in performance between the best and worst plan is 68% (i.e. the worst plan only yields 32% of the peak performance). The second best plan yields 88% of the performance of the best plan. For the smaller transform sizes, the behavior exhibited is similar to that of the previous architecture where the second best plan represents a drastic decrease (60-90%) in performance from that of the best plan due to the overhead incurred in combining the codelets. In Figure 4.68, we present the performance of the best execution plans of the UHRFFT library on the Power3 processor for sizes that are powers of two and compare it with the performance of the FFTW library for similar sizes. Again we see that for the smaller sizes, the UHRFFT library outperforms the FFTW library as in the previous architecture. The performance that is shown is what is available to the applications that use the library for these given sizes. The complete plan performance data is given in Figures 4.59, 4.61, 4.63, 4.65 and 4.67. Irregular behavior for the execution plans of size 64 and 512 for the FFTW library is observed in Figure 4.68. It is not certain why this behavior is observed, but repeated measurements yielded the same results and thus ruled out the fact that the irregularity was due to instrumentation errors.
<table>
<thead>
<tr>
<th>Plan</th>
<th>Time (sec)</th>
<th>MFlops</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 64</td>
<td>2.15e-05</td>
<td>219.39</td>
<td>0.247</td>
</tr>
<tr>
<td>64 32</td>
<td>2.10e-05</td>
<td>241.36</td>
<td>0.271</td>
</tr>
<tr>
<td>32 16</td>
<td>2.47e-05</td>
<td>209.42</td>
<td>0.236</td>
</tr>
<tr>
<td>16 8</td>
<td>3.40e-05</td>
<td>164.52</td>
<td>0.185</td>
</tr>
<tr>
<td>8 4</td>
<td>4.72e-05</td>
<td>115.20</td>
<td>0.130</td>
</tr>
<tr>
<td>4 2</td>
<td>8.05e-05</td>
<td>76.75</td>
<td>0.086</td>
</tr>
</tbody>
</table>

Table 4.14: Execution Plan Performance for Size 256 FFTs on a 222 MHz Power3.

Figure 4.59: Execution Plan Performance for Size 8 and Size 16 FFTs on a 222 MHz Power3.
Figure 4.60: Execution Plan Performance for Size 32 and Size 64 FFTs on a 222 MHz Power3.

Figure 4.61: Execution Plan Performance for Size 128 and Size 256 FFTs on a 222 MHz Power3.
Figure 4.62: Execution Plan Performance for Size 512 and Size 1024 FFTs on a 222 MHz Power3.

Figure 4.63: Execution Plan Performance for Size 2048 and Size 4096 FFTs on a 222 MHz Power3.
<table>
<thead>
<tr>
<th>Plan</th>
<th>Performance (MFLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(128, 64)</td>
<td>120</td>
</tr>
<tr>
<td>(64, 32)</td>
<td>130</td>
</tr>
<tr>
<td>(32, 16)</td>
<td>130</td>
</tr>
<tr>
<td>(16, 8)</td>
<td>120</td>
</tr>
<tr>
<td>(8, 4)</td>
<td>110</td>
</tr>
<tr>
<td>(4, 2)</td>
<td>100</td>
</tr>
</tbody>
</table>

Figure 4.64: Execution Plan Performance for Size 8096 and Size 16192 FFTs on a 222 MHz Power3.

<table>
<thead>
<tr>
<th>Plan</th>
<th>Performance (MFLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(128, 64)</td>
<td>120</td>
</tr>
<tr>
<td>(64, 32)</td>
<td>130</td>
</tr>
<tr>
<td>(32, 16)</td>
<td>130</td>
</tr>
<tr>
<td>(16, 8)</td>
<td>120</td>
</tr>
<tr>
<td>(8, 4)</td>
<td>110</td>
</tr>
<tr>
<td>(4, 2)</td>
<td>100</td>
</tr>
</tbody>
</table>

Figure 4.65: Execution Plan Performance for Size 32768 and Size 65536 FFTs on a 222 MHz Power3.
Figure 4.66: Execution Plan Performance for Size 131072 and Size 262144 FFTs on a 222 MHz Power3.

<table>
<thead>
<tr>
<th>Plan</th>
<th>Performance (MFLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(128 64)</td>
<td>74.3</td>
</tr>
<tr>
<td>(64 32)</td>
<td></td>
</tr>
<tr>
<td>(32 16)</td>
<td></td>
</tr>
<tr>
<td>(16 8)</td>
<td></td>
</tr>
<tr>
<td>(8 4)</td>
<td></td>
</tr>
<tr>
<td>(4 2)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.67: Execution Plan Performance for Size 524288 and Size 1048576 FFTs on a 222 MHz Power3.

<table>
<thead>
<tr>
<th>Plan</th>
<th>Performance (MFLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(128 64)</td>
<td>42.5</td>
</tr>
<tr>
<td>(64 32)</td>
<td></td>
</tr>
<tr>
<td>(32 16)</td>
<td></td>
</tr>
<tr>
<td>(16 8)</td>
<td></td>
</tr>
<tr>
<td>(8 4)</td>
<td></td>
</tr>
<tr>
<td>(4 2)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Plan</th>
<th>Performance (MFLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(128 64)</td>
<td>33.6</td>
</tr>
<tr>
<td>(64 32)</td>
<td></td>
</tr>
<tr>
<td>(32 16)</td>
<td></td>
</tr>
<tr>
<td>(16 8)</td>
<td></td>
</tr>
<tr>
<td>(8 4)</td>
<td></td>
</tr>
<tr>
<td>(4 2)</td>
<td></td>
</tr>
</tbody>
</table>
4.6.3 The Intel Pentium–III 500

Table 4.15 shows the performance of all the execution plans for a real transform of size 256. For this size, the peak performance of the UHRFFT execution plans on the Pentium–III 500 is 186.18 MFLOPS time. The second best plan yields 84% of the performance of the best plan while the worst plan only yields 45% of peak performance. This behavior is typical of all the larger sized transforms. For the smaller sized transforms, the disparity between the performance of the best plan and that of the second best plan is much greater. This demonstrates the importance of selecting the best plan during execution. In Figure 4.78, we present the performance of the best execution plans of the UHRFFT library on the Intel Pentium–III 500 processor for sizes that are powers of two and compare it with the performance of the FFTW library for similar sizes. The performance presented is what is available to the applications that use the library for these given sizes. The complete plan
performance data is given in Figures 4.69, 4.71, 4.73, 4.75 and 4.77.

<table>
<thead>
<tr>
<th>Plan</th>
<th>Time (sec)</th>
<th>MFlops</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 64</td>
<td>6.00e-05</td>
<td>85.33</td>
<td>0.171</td>
</tr>
<tr>
<td>64 32</td>
<td>3.39e-05</td>
<td>150.68</td>
<td>0.301</td>
</tr>
<tr>
<td>32 16</td>
<td>2.75e-05</td>
<td>186.18</td>
<td>0.372</td>
</tr>
<tr>
<td>16 8</td>
<td>3.25e-05</td>
<td>157.49</td>
<td>0.315</td>
</tr>
<tr>
<td>8 4</td>
<td>3.45e-05</td>
<td>148.41</td>
<td>0.297</td>
</tr>
<tr>
<td>4 2</td>
<td>4.50e-05</td>
<td>113.73</td>
<td>0.228</td>
</tr>
</tbody>
</table>

Table 4.15: Execution Plan Performance for Size 256 FFTs on a 500 MHz Pentium–III.

The drop in the performance of the UHRFFT transform of size 128 is due to the fact that the codelet of size 128 is too large to fit in the system cache and as a result, better performance is acheived by using a different plan [32, 16], as can be seen in Figure 4.71. This plan performance is better with respect to the performance of the codelet of size 128 but still suffers performance loss due to cache capacity. This phenomenon is also true for the Intel Pentium–III 550 Xeon, and can be oberved in Figure 4.81.
Figure 4.69: Execution Plan Performance for Size 8 and Size 16 FFTs on a 500 MHz Pentium-III.

Figure 4.70: Execution Plan Performance for Size 32 and Size 64 FFTs on a 500 MHz Pentium-III.
Figure 4.71: Execution Plan Performance for Size 128 and Size 256 FFTs on a 500 MHz Pentium–III.

Figure 4.72: Execution Plan Performance for Size 512 and Size 1024 FFTs on a 500 MHz Pentium–III.
Figure 4.73: Execution Plan Performance for Size 2048 and Size 4096 FFTs on a 500 MHz Pentium–III.

Figure 4.74: Execution Plan Performance for Size 8192 and Size 16384 FFTs on a 500 MHz Pentium–III.
Figure 4.75: Execution Plan Performance for Size 32768 and Size 65536 FFTs on a 500 MHz Pentium–III.

Figure 4.76: Execution Plan Performance for Size 131072 and Size 262144 FFTs on a 500 MHz Pentium–III.
Figure 4.77: Execution Plan Performance for Size 524288 and Size 1048576 FFTs on a 500 MHz Pentium-III.

Figure 4.78: 500 MHz Pentium-III Execution Plan Performance.
4.6.4 The Intel Pentium–III 550 Xeon

Table 4.16 shows the performance of all execution plans for a size 256 FFT on the Pentium–III 550 Xeon processor. Figure 4.88 shows the performance of the best execution plans for the UHRFFT library for sizes that are powers of two. A comparison with the performance of the FFTW library for these sizes is also shown. Complete plan performance data is given in Figures 4.79, 4.81, 4.83, 4.85 and 4.87.

<table>
<thead>
<tr>
<th>Plan</th>
<th>Time (sec)</th>
<th>MFlops</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 64</td>
<td>4.77e-05</td>
<td>107.27</td>
<td>0.195</td>
</tr>
<tr>
<td>64 32</td>
<td>2.86e-05</td>
<td>178.77</td>
<td>0.325</td>
</tr>
<tr>
<td>32 16</td>
<td>2.38e-05</td>
<td>214.49</td>
<td>0.390</td>
</tr>
<tr>
<td>16 8</td>
<td>2.88e-05</td>
<td>177.29</td>
<td>0.322</td>
</tr>
<tr>
<td>8 4</td>
<td>3.10e-05</td>
<td>165.16</td>
<td>0.300</td>
</tr>
<tr>
<td>4 2</td>
<td>3.93e-05</td>
<td>130.02</td>
<td>0.236</td>
</tr>
</tbody>
</table>

Table 4.16: Execution Plan Performance for Size 256 FFTs on a 550 MHz Xeon.

![Figure 4.79: Execution Plan Performance for Size 8 and Size 16 FFTs on a 550 MHz Xeon.](image)

126
Figure 4.80: Execution Plan Performance for Size 32 and Size 64 FFTs on a 550 MHz Xeon.
Figure 4.81: Execution Plan Performance for Size 128 and Size 256 FFTs on a 550 MHz Xeon.

Figure 4.82: Execution Plan Performance for Size 512 and Size 1024 FFTs on a 550 MHz Xeon.
Figure 4.83: Execution Plan Performance for Size 2048 and Size 4096 FFTs on a 550 MHz Xeon.

Figure 4.84: Execution Plan Performance for Size 8096 and Size 16384 FFTs on a 550 MHz Xeon.
<table>
<thead>
<tr>
<th>Plan</th>
<th>Performance (MFLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(128 64)</td>
<td>100</td>
</tr>
<tr>
<td>(64 32)</td>
<td>95.5</td>
</tr>
<tr>
<td>(32 16)</td>
<td>80</td>
</tr>
<tr>
<td>(16 8)</td>
<td>65</td>
</tr>
<tr>
<td>(8 4)</td>
<td>50</td>
</tr>
<tr>
<td>(4 2)</td>
<td>35</td>
</tr>
</tbody>
</table>

Figure 4.85: Execution Plan Performance for Size 32768 and Size 65536 FFTs on a 550 MHz Xeon.

<table>
<thead>
<tr>
<th>Plan</th>
<th>Performance (MFLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(128 64)</td>
<td>100</td>
</tr>
<tr>
<td>(64 32)</td>
<td>95.5</td>
</tr>
<tr>
<td>(32 16)</td>
<td>80</td>
</tr>
<tr>
<td>(16 8)</td>
<td>65</td>
</tr>
<tr>
<td>(8 4)</td>
<td>50</td>
</tr>
<tr>
<td>(4 2)</td>
<td>35</td>
</tr>
</tbody>
</table>

Figure 4.86: Execution Plan Performance for Size 131072 and Size 262144 FFTs on a 550 MHz Xeon.
Figure 4.87: Execution Plan Performance for Size 524288 and Size 1048576 FFTs on a 550 MHz Xeon.

Figure 4.88: 550 MHz Xeon Execution Plan Performance.
Chapter 5

Interfaces and Library Packaging

In this chapter we describe the UHRFFT library Applications Programmer Interface (API) and provide some examples that demonstrate how to use the library to compute FFTs on real input data. We also briefly discuss how to install the library. Before we describe the interfaces and the packaging of the library, we present a brief argument in favor of a common portable interface for the FFT library.

5.1 Motivation

We observed certain shortcomings in the APIs of different Fast Fourier Transform libraries that are available publicly and undertook to minimize these in the UHRFFT library. Some of the shortcomings were the following:

- There are no uniform requirements/restrictions on length, auxiliary storage and data distribution among the software packages.

- Current interfaces do not support calls from all major languages (eg. C, C++, F77, F90, HPF). A single library on any platform usually supports calls from
only particular programming languages. In order to maximize the reuse of
library code we strongly recommend a single library that can support the com-
monly used programming languages.

- Portability of application codes becomes restricted to machines of a single archi-
tecture if hardware vendor unique interfaces are used. Software vendor unique
interfaces may also restrict portability in that the library may not be available
on all platforms on which it may be desirable to execute the application code.

- Interfaces are not common for single and multi-processor codes. Though the
results reported in this thesis are focused on single node performance the objec-
tive of the effort in which these results were obtained is to develop a portable
and efficient FFT library suitable for parallel and distributed execution en-
vironments. No FFT library we have come across provides a single interface
suitable for single and multi-processor architectures, except the Connection
Machine Scientific Software Library (CMSSL) [12] that provided a single in-
terface supporting both single and multi-processor execution environments for
languages with an array syntax.

- Interfaces are not common for one-dimensional and multi-dimensional FFTs.
Some libraries use a unique interface for each dimensionality of transforms.
Both with respect to user effort in learning library interfaces and how to use
them, as well as with respect to development and maintenance of libraries we
strongly prefer library designs that use a single interface regardless of dimen-
sionality of the transform. The CMSSL provided a single interfaces supporting
transforms of any dimensionality.
The most portable library we have come across so far is the FFTW [14, 35] library from MIT, but this library uses different interfaces for real and complex transforms. We have simplified the interfaces compared to the ones in the FFTW library by trying to use a common interface for all types of transforms. We view the interfaces presented next as both portable across different platforms and simple to use.

### 5.2 UHRFFT’s API

The first step in using the UHRFFT is to create a plan called \textit{FftPlan}, which is a special structure used by the UHRFFT library in executing the desired transform. The plan contains all information that the UHRFFT library needs to compute the FFT. A separate plan is required for each size transform that the application uses. There are two interfaces that can be used by UHRFFT in order to generate the plan. They are:

- \texttt{UHRFftInit()}.
- \texttt{UHRFftInitDetail()}.  

The \texttt{UHRFftInit()} function is used when the transform is to be computed on all data points on the specified axes, while \texttt{UHRFftInitDetail()} allows the user to compute the transform on a subset of the data points on the specified axes by giving the number of data points and strides for data on each axes. The \texttt{UHRFftInit()} routine has the following arguments:

1) A pointer to the \textit{FftPlan}.

2) The total number of axes in the input array.
3) An array which contains the number of data elements along each axis.

4) The number of axes on which the transform is to be computed.

5) An array which specifies the axes on which the transform is to be computed.

The interface for `UHRFftInit()` is:

```c
UHRFftInit(plan, Array_dims, Dim_sizes, Transform_dims, Transform_axes)
```

where `plan` is a pointer to the `FftPlan` structure used by the UHRFFT, `Array_dims` is the total number of axes in the input array, `Dim_sizes` is an array which contains the sizes (lengths) of each axis in the input array, i.e., `Dim_sizes[0]` contains the length of axis zero, `Dim_sizes[1]` contains the length of axis one, and so on, `Transform_dims` is the total number of axes on which the transform is to be performed and `Transform_axes` specifies the specific axes on which the transform is to be performed. For the `UHRFftInit()` function, the transform will be performed on all the data points on the specified axes. The interface definition for `UHRFftInit()` in C and Fortran 90 are given below:

C:
```
void UHRFftInit(plan, Array_dims, Dim_sizes, 
                Transform_dims, Transform_axes)

FftPlan *plan;
int Array_dims;
int *Dim_sizes;
int Transform_dims;
int *Transform_axes;
```

Fortran 90:
```
UHRFftInit(plan, Array_dims, Dim_sizes, Transform_dims, Transform_Axes)

integer plan(UHRFftPlanSize),Array_dims,Dim_sizes(Array_dims),
            Transform_dims, Transform_axes(Transform_dims)
```

135
In Fortran, the \textit{FftPlan} is pointed to by using an integer array of size \textit{UHRFftPlan-Size}, which is a constant defined in the library.

The \texttt{UHRFftInitDetail()} routine takes the same arguments as the \texttt{UHRFftInit()} routine along with a few extra parameters. This routine takes as parameters an array which specifies the number of data points along each axis on which the transform is to be performed and an array which contains the strides for those data points. The interface for \texttt{UHRFftInitDetail()} is:

\begin{verbatim}
UHRFftInitDetail(plan, Array_dims, Dim_sizes, Transform_dims, Transform_axes, Lengths, Strides)
\end{verbatim}

where the meanings of \texttt{plan}, \texttt{Array_dims}, \texttt{Dim_sizes}, \texttt{Transform_dims} and \texttt{Transform_axes} are the same as for the \texttt{UHRFftInit()} interface. \texttt{Lengths} is an array of length \texttt{Transform_dims}, which contains the number of data-points on which the transform is to be performed on each specified axis, i.e., \texttt{Lengths[0]} is the number of data-points on the first specified axis, \texttt{Lengths[1]} is the number of data-points on the second specified axis and so on, and \texttt{Strides} is an array of length \texttt{Transform_dims}, which contains the strides for the data on each specified axes on which the transforms are to be performed. The interface definition for \texttt{UHRFftInitDetail()} in C and Fortran 90 are as follows:

**C:**

\begin{verbatim}
void UHRFftInitDetail(plan, Array_dims, Dim_sizes, Transform_dims, Transform_axes, Lengths, Strides)
\end{verbatim}

\begin{verbatim}
FftPlan *plan;
int Array_dims;
int *Dim_sizes;
int Transform_dims;
int *Transform_axes;
int *Lengths;
\end{verbatim}

136
int *Strides;

Fortran 90:
   UHRFftInitDetail(plan, Array_dims, Dim_sizes, Transform_dims,
                      Transform_axes, Lengths, Strides)

   integer plan(UHRFftPlanSize), Array_dims, Dim_sizes(Array_dims),
              Transform_dims, Transform_axes(Transform_dims),
              Lengths(Transform_dims), Strides(Transform_dims)

The next step is the actual transform execution. One can now compute as many
FFTs as required by using the function UHRFftExe(). This routine takes three
arguments: the input array, the output array and the FftPlan created by UHRFf-
tInit() for that particular size FFT. If an in-place FFT is desired, both the input
array and output array should point to the same array of data. For both cases, no
large temporary storage is required. The UHRFFT library uses temporary storage
only within codelets and this storage is not more than the size of the codelets them-
selves. Usually these temporary variables are stored in the cache or registers and are
never stored in or accessed from the main memory.

The interface for UHRFftExe() is:

   UHRFftExe(in, out, plan),

where in and out are pointers to the input and output arrays respectively, and plan
is a pointer to the FftPlan structure used by the UHRFFT library. The interface
definition for UHRFftExe() in C and Fortran 90 are as follows:

C:

   UHRFftExe(in, out, plan)
   REAL *in;
   REAL *out;
   FftPlan *plan;
Fortran 90:

```fortran
UHRFftExe(in,out,plan)
real in(*), out(*)
in plan(UHRFftPlanSize)
```

Here, *REAL* is defined to be either a *float* or a *double* in 'C.' *in* and *out* are pointers to the input and output arrays for the data for which the transform is to be computed. The output vector from the FFT computation on real data input has conjugate–even symmetry and therefore, the data storage format that is chosen in UHRFFFT for the output vector follows the conjugate–even storage scheme which is defined as follows:

If \( N = 2M \), the conjugate–even vector \( \textbf{out} \) has the form

\[
\textbf{out} = \begin{pmatrix}
    a \\
    b + ic \\
    d \\
    Eb - iEc
\end{pmatrix},
\]

where \( a \) and \( d \) are real scalars, \( b \) and \( c \) are real vectors.

The vector \( \textbf{out} \) can be represented as a single real \( N \)-vector

\[
\textbf{out}^{(\alpha)} = \begin{pmatrix}
    a \\
    b \\
    d \\
    c
\end{pmatrix},
\]

when \( N \) is even and as

\[
\textbf{out}^{(\alpha)} = \begin{pmatrix}
    a \\
    b \\
    c
\end{pmatrix},
\]
when $N$ is odd. This representation amounts to the non-redundant stacking of the real and imaginary parts of $y$. This concept is explained in more detail in section 2.5.

Once all FFT computations using a particular plan are completed, the plan is deallocated using the function UHRFftClean(). The function takes as argument the pointer to the plan which one wishes to deallocate. This frees up memory used by the library to store twiddle factors and other information used by the library to compute the FFT. The interface for the UHRFftClean() function is as follows:

```
UHRFftClean(plan)
```

The interface definition for UHRFftClean() in C and Fortran 90 are as follows:

C:
```
UHRFftClean(plan)
FftPlan *plan;
```

Fortran 90:
```
UHRFftClean(plan)
int plan(UHRFftPlanSize)
```

This general interface can be used for any size FFT with any number of dimensions. We can also compute transforms on parts of an array by supplying the proper axes, lengths and strides for each dimension.

### 5.3 Usage

In this section we describe the usage of the UHRFFT library, i.e., how to compute FFTs using the library. We assume that the UHRFFT library is already installed on the system. The library installation details are described in section 5.4.
We first create the execution plan \texttt{FftPlan} using either the \texttt{UHRFFtInit()} routine or the \texttt{UHRFFtInitDetail()} routine. One may then compute as many FFTs of the specified size as required using the \texttt{UHRFFtExe()} routine. Finally, when all computations are done, we free memory using the \texttt{UHRFFtClean()} routine. A typical call in 'C' to the UHRFFT for a one-dimensional transform looks like:

```c
#include <uhrfft.h>
...
{
  FftPlan *plan, Plan;
  REAL in[N], out[N];
  int sz[1], axs[1];

  plan = &Plan;
  sz[0] = N;
  axs[0] = 1;
  istr[0] = ostr[0] = 1;
  UHRFFtInit(plan, 1, sz,1, axs);
  ...
  UHRFFtExe(in,out,plan);
  ...
  UHRFFtClean(plan);
}
```

and a typical call to the UHRFFT in 'Fortran' looks like:

```fortran
include 'uhrfft.f'
...
{
  double real in, out
  dimension in(N), out(N)
  integer plan
  dimension plan(UHRFFtPlanSize)
  integer sz, axs
  dimension axs(1),sz(1)

  istr(1) = ostr(1) = 1
```
sz(1) = N
    call UHRFftInit(plan,1,sz,1,axs)
    ...
    call UHRFftExe(in,out,plan)
    ...
    call UHRFftClean(plan)
}

The uhrfft.h file needs to be included since the FftPlan structure definition and other function definitions are included in this file. For Fortran, we include a corresponding uhrfft.f.h file. Since Fortran does not support structures, we use wrapper functions in 'C' to access the FftPlan structure. We declare the pointer as an integer array of size UHRFftPlanSize in Fortran, which is a constant declared in the uhrfft.f.h header file. This is required in Fortran to allocate the memory that the FftPlan structure uses in 'C'. The pointer to the plan is returned to the code via the first argument in the UHRFftInit() routine and this plan is used in subsequent calls to the UHRFftExe() routines. All arrays have their indices starting from 1 instead of zero as in 'C'.

5.4 Installation

UHRFFT makes very few assumptions about the execution environment. All that is required is an ANSI C compiler (gcc will work fine though vendor–provided ANSI C compilers are usually better).

Installation of the library is very simple on Unix systems or a GNU system, such as Linux. The UHRFFT comes with a configure program in the GNU format. This makes the installation as simple as:

    ./configure
make
make install
benchuhrrfft

The installation process proceeds as follows. First the ./configure command creates a Makefile for the system with all compiler flags and system optimizations possible enabled. Next, the make command creates the UHRFFT library and test programs for the library. Then, the make install command installs the library in the standard library directory or in a directory specified by INSTALLDIR in the configure script. The default value for INSTALLDIR is the current directory from which the user attempts to install the library. Then, a benchmark test routine benchuhrrfft should be run to create the codelet and transforms databases on the particular system. The databases are created in the INSTALLDIR directory. This is done only once. The applications then use these databases in creating execution plans according to their needs.

The customization is an automatic process in the UHRFFT library installation. The configure script contains information about good optimization flags (C compiler Flags) for many systems. The library is now initialized and ready to use.

For the systems discussed in section 4.2, the initialization procedure uses the compiler flags that were found to yield the best performance for the codelets as observed in section 4.4 and summarised in Table 5.1.

<table>
<thead>
<tr>
<th>System</th>
<th>OS version</th>
<th>Compiler</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGI R10000</td>
<td>IRIX 6.5.1</td>
<td>MIPSpro cc 7.3.1.m</td>
<td>-02 -mips4</td>
</tr>
<tr>
<td>IBM Power3</td>
<td>AIX 5.0.0</td>
<td>xlc</td>
<td>-02 -qarch=pwr3</td>
</tr>
<tr>
<td>Intel Pentium-III</td>
<td>Linux 2.2.14-5.0</td>
<td>gcc 2.7.2.3</td>
<td>-01 -fomit-frame-pointer</td>
</tr>
</tbody>
</table>

Table 5.1: Compiler Flags for the UHRFFT on Some Architectures.
For other systems, the configure script uses some general optimization flags such as "O2" or "O–max" to compile the library. One can easily add new flags to the script for new systems or change existing flags if a better set of flags are discovered for a particular system. The **CFLAGS** variable in the Makefile needs to be changed to include the new optimization flags. Users can experiment with different optimization flags to see how the performance of the library varies for each of them.
Chapter 6

Conclusion and Future Work

6.1 Summary

We have evaluated the UHRFFT library on multiple platforms and seen that we achieve good performance on all architectures. The adaptive approach that we have chosen for the library is shown to be an elegant and efficient way of achieving both portability and good performance. The codelet approach is also shown to be an effective method in tackling the performance tuning problem for multiple platforms. Unlike earlier approaches where typically a single algorithm was hand tuned to perform well on a particular architecture, the adaptive approach automatically finds computationally efficient combination of codelets on any given architecture.

Straight line code for large DFTs proves to be more efficient than constructing them with smaller kernels. For example, a straight line DFT codelet of size 16 is much more efficient than computing the size 16 DFT using smaller codelets. But writing large straight line code is time consuming. Thus, the code generator approach to building the codelets is an efficient way to address the problem. The effectiveness
of straight line code is seen best when the entire codelet fits in the instruction cache and can make use of the floating-point registers well. Hence, codelets of size 8 and 16 perform well on almost all architectures, whereas some larger codelets do not perform as well as expected due to this restriction.

The overall design of the library is also seen to be flexible and extensible. The adaptivity of the library combines portability with these features. The ease with which the whole UHRFFT library can be regenerated allows us to easily incorporate new features and optimizations to the library. We can easily incorporate new optimization rules and techniques to the library and regenerate the whole UHRFFT library in a matter of seconds. We can also extend the library to add new codelets to accommodate more transform sizes if required. For example, if one needs to compute transforms that require a codelet of size 31, we can very easily add this codelet to the library using the code generator and use it for the application.

6.2 Future Work

The UHRFFT library is far from complete. We have so far only demonstrated the effectiveness of the adaptive algorithm and our approach to the design of the library. There are still many more optimizations possible. Since there are many different algorithms and ways of doing the FFT, there are an endless number of options and possibilities. We would like to try to find new optimizations of existing algorithms if possible and further enhance the performance of the library. One such way to enhance the performance of the library would be to include a scheduler for the instructions generated by the codelets. The scheduler would reorder these instructions (taking into consideration data dependancies of the instructions) in order
to achieve a more efficient performance of the codelets. This idea is currently being
looked into. The UHFFT library also needs to be extended further to include sine
and cosine transforms. Other applications such as convolution also can be included
in the library.

Currently, UHFFT does not support FFT’s of data sets that do not fit in the
memory of the system that it is running on. UHFFT needs to have an out-of-core
FFT algorithm to compute the FFT of data sets that are stored in secondary or
tertiary storage because many scientific applications deal with large amounts of data
that may not fit in the real memory of the system.

We have so far implemented a uniprocessor library. Many applications that use
FFT run on parallel systems with data distributed over many processors. Hence, a
parallel implementation of the library is essential. The same approach of multi-level
optimization can be applied to the parallel implementation too. The communication
pattern of the different algorithms should also be taken into consideration while
choosing the algorithm for a particular FFT size. It is necessary for the library to
have a portable interface for the multiprocessor library so that it can be used easily
by different applications. This interface needs to support most parallel programming
paradigms such as PVM, MPI etc., and data parallel languages, such as HPF.
References


