COMPUTING THE FAST FOURIER TRANSFORM ON SIMD MICROPROCESSORS

A thesis

submitted in fulfilment

of the requirements for the Degree

of

Doctor of Philosophy

at the

University of Waikato

by

ANTHONY BLAKE

University of Waikato

2012

COMPUTING THE FAST FOURIER TRANSFORM ON SIMD MICROPROCESSORS

ANTHONY BLAKE

February 2012

ABSTRACT

This thesis describes how to compute the fast Fourier transform (FFT) of a power-of-two length signal on single-instruction, multiple-data (SIMD) microprocessors faster than or very close to the speed of state of the art libraries such as FFTW ("Fastest Fourier Transform in the West"), SPIRAL and Intel Integrated Performance Primitives (IPP).

The conjugate-pair algorithm has advantages in terms of memory bandwidth, and three implementations of this algorithm, which incorporate latency and spatial locality optimizations, are automatically vectorized at the algorithm level of abstraction. Performance results on 2-way, 4-way and 8-way SIMD machines show that the performance scales much better than FFTW or SPIRAL.

The implementations presented in this thesis are compiled into a high-performance FFT library called SFFT ("Streaming Fast Fourier Transform"), and benchmarked against FFTW, SPIRAL, Intel IPP and Apple Accelerate on sixteen x86 machines and two ARM NEON machines, and shown to be, in many cases, faster than these state of the art libraries, but without having to perform extensive machine specific calibration, thus demonstrating that there are good heuristics for predicting the performance of the FFT on SIMD microprocessors (i.e., the need for empirical optimization may be overstated).

ACKNOWLEDGMENTS

I would like to thank my chief supervisor, Ian Witten, for all his patience, support and guidance, much of which was needed before I finished this thesis. I benefited a great deal from his skill and experience, and I am very fortunate to have been his student. I would also like to thank my other supervisors, Michael Cree and John Perrone, for all their support and guidance, and I appreciate all the time, insightful comments and advice they have given me, and I am grateful for having a panel of supervisors from diverse backgrounds.

Everyone in the Digital Libraries group inspired me with their excellent academic writing and presentation skills, and helped me with feedback on my presentations, which I appreciate because most of my research is unrelated to their interests.

I would like to thank Clint Dilks, Glen Ogilvie, my supervisors and everyone else who helped with running the benchmarks in this thesis.

I'm grateful to have been part of the Department of Computer Science, and for the many interesting conversations I've had with its members over the years, on a diverse range of topics, but in particular I'd like to thank Gian, Perry and Anu for all their time.

This work was funded with scholarships from the University of Waikato and the Department of Computer Science, and with support from John Perrone's Marsden Fund grant.

Finally, to all my other friends and family, thank-you.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ACM	association for computing machinery
ARM	advanced RISC machine
ASIC	application specific integrated circuit
AVX	advanced vector extensions
CPU	central processing unit
CTG	Cooley-Tukey gigaflop
DAG	directed acyclic graph
DDR	double data rate
DFT	discrete Fourier transform
DIF	decimation in frequency
DIT	decimation in time
DRAM	dynamic random-access memory
DSP	digital signal processor
FFT	fast Fourier transform
FHT	fast Hartley transform
FLOP	floating point operation
FLOPS	floating point operations per second

FPGA	field-programmable gate array
GPU	graphics processing unit
ICC	Intel C compiler
IDC	International Data Corporation
IDFT	inverse discrete Fourier transform
IFFT	inverse fast Fourier transform
IPP	integrated performance primitives
LUT	lookup table
MMX	multimedia extensions
OS	operating system
PC	personal computer
RAM	random-access memory
RISC	reduced instruction set computer
RMS	root-mean-square
SAM	sequential-access memory
SDRAM	synchronous dynamic random-access memory
SIMD	single instruction, multiple data
SOC	system on chip
SPIRAL	signal processing algorithms implementation research for adaptive libraries
SSE	streaming SIMD extensions

VL	vector length

Part I

STATE OF THE ART

1 INTRODUCTION

"...the manner in which the author arrives at these equations is not exempt of difficulties and...his analysis to integrate them still leaves something to be desired on the score of generality and even rigour."

— Peer review committee on Fourier's 1807 paper [12]

In 1990, it was estimated that Cray Research's installed base of approximately 200 machines spent 40% of all central processing unit (CPU) cycles computing the fast Fourier transform (FFT) [45]. With each machine worth about USD$25 million, the performance of the FFT was of prime importance.

Today, use of the FFT is even more pervasive, and it is counted among the 10 algorithms that have had the greatest influence on the development and practice of science and engineering in the 20th century [20]. Huge numbers of mobile smartphones, tablets and PCs [58, 26], most of which are equipped with single instruction, multiple data (SIMD) [24, 27] microprocessors, compute the FFT on a large scale for a plethora of sound, video and image processing applications. In the space of a few years, mobile applications have become a part of many people's everyday lives [36].

This thesis shows that the key to optimizing the performance of the split-radix FFT algorithms on SIMD microprocessors is latency and spatial locality optimizations, and in some cases, a variant of the split-radix

FFT called the conjugate-pair algorithm [37, 48, 53, 68]. It is also shown that extensive machine specific calibration may be superfluous.

1.1 HYPOTHESES

FFTW [34, 35, 46], SPIRAL [32, 66, 67] and UHFFT [4, 6, 3, 61, 60] are state of the art FFT libraries that employ automatic empirical optimization. SPIRAL automatically performs machine-specific optimizations at compile time, and FFTW and UHFFT automatically adapt to a machine at run-time. Aside from the use of automatic optimization, a common denominator among these libraries is the use of large straight line blocks of code and optimized memory locality.

The hypotheses outlined below test whether good heuristics and model-based optimization can be used in the place of automatic empirical optimization.

Hypothesis 1: Accessing memory in sequential "streams" is critical for best performance

Large FFTs exhibit poor temporal locality, and when computing these transforms on microprocessor based systems that feature a cache, best performance is typically achieved when "streaming" sequential data through the CPU. Hypothesis 1 is tested in Chapter 3 with replicated coefficient lookup tables that trade-off increased memory size for better spatial locality, and in Chapter 5 by topologically sorting a directed acyclic graph (DAG) of sub-transforms to again improve spatial locality.

Hypothesis 2: The conjugate-pair algorithm is faster than the ordinary split-radix algorithm

Hypothesis 2 is based on the idea that memory bandwidth is a bottleneck, and on the fact that the conjugate-pair algorithm requires only half the number of twiddle factor loads. This hypothesis is tested in Section 7.6, where a highly optimized implementation of the conjugate-pair algorithm is benchmarked against an equally highly optimized implementation of the ordinary split-radix algorithm.

Hypothesis 3: The performance of an FFT can be predicted based on characteristics of the underlying machine and the compiler

Exploratory experiments suggest that good results can be obtained without empirical techniques, and that certain parameters can be predicted based on the characteristics of the underlying machine and the compiler used. Hypothesis 3 is tested in Chapter 7 by building a model that predicts performance, and by benchmarking FFTW against an implementation that does not require extensive calibration, on 18 different machines.

1.2 SCOPE

In investigating the hypotheses, the scope of this work has been limited in several ways:

1. It is limited to single-threaded complex 1D FFTs, because multi-dimensional, multi-threaded or multi-processor FFTs (or any combination thereof) are ultimately decomposed into 1D components

running on a single core, and all other things being equal, it is the performance of these 1D components running on a single microprocessor core that determines the overall performance of a given multi-threaded implementation;

2. It is limited to transforms that operate on vectors of length 2^m where $m \in \mathbb{N}^0$, because these are the easiest to compute on machines, and consequently the most often used by applications. This excludes the prime-factor algorithm [64, 74], and the Radar [71] and Bluestein [11, 64, 70] algorithms for prime sizes;

3. It is limited to the split-radix [22, 23, 56, 75, 78] and conjugate-pair [37, 48, 53, 68] algorithms. The Winograd algorithm [21, 23, 40, 76] is excluded because of its low performance on systems where multiplication costs about the same as addition;

4. It is limited to out-of-place transforms, because they are generally faster than in-place transforms, except at the boundaries of the cache [5];

5. The benchmark experiments are limited to the Intel x86 and ARM machines, because it is estimated that 92% of the microprocessors in the rapidly expanding mobile market are ARM devices [26], while Intel's share of the worldwide PC and mobile PC microprocessors markets is estimated to be 79.3% and 84.4%, respectively [58].

1.3 CONTRIBUTIONS

The contributions of this work are summarized as follows:

1. Three methods of computing the conjugate-pair algorithm on SIMD microprocessors are described in Chapter 5;

2. The source code for the high-performance SIMD FFT library developed in this thesis is publicly available under a permissive open source licence at `https://github.com/anthonix/sfft`.

1.4 ORGANIZATION

This work is divided into two parts. The first part, Chapters 1 – 4, encompasses the relevant background, while the second part, Chapters 5 – 8, is concerned with contributions that challenge the state of the art.

A brief overview of the contents of each chapter:

2. *Algorithms* provides an overview of FFT algorithms from the mathematical perspective;

3. *Implementation details* complements the mathematical perspective of the previous chapter with a more focused view of the low level details that are relevant to efficient implementation on SIMD microprocessors;

4. *Existing libraries* reviews existing state of the art libraries, with reference to algorithms and implementation details of the previous chapters;

5. *A high-performance FFT library* describes SFFT, a library for SIMD microprocessors that is, in many cases, faster than the state of the art FFT libraries reviewed in Chapter 4;

6. *Benchmark methods* describes the benchmarking methods used to evaluate performance and accuracy of various FFT implementations throughout this work;
7. *Results and discussion* presents the results of benchmarks on 18 different machines, as well as the results of model-based optimization experiments, with reference to earlier chapters and other related work;
8. *Conclusions and future work* concludes the work with a review of the hypotheses, a summary of the contributions, and some idea for directions that future work might take.

2 ALGORITHMS

"...we want good algorithms in some loosely defined aesthetic sense. One criterion ... is the length of time taken to perform the algorithm ... Other criteria are adaptability of the algorithm to computers, its simplicity and elegance, etc"

— Donald E. Knuth [50]

Efficient computation of the FFT requires an understanding of the computation at every level of abstraction, from the high-level algorithmic view down to the low-level details of the target machine (or failing that, a lot of time to code all known FFT algorithms and exhaustively search the configuration space). This chapter provides an overview of FFTs from the mathematical perspective.

Fast Fourier transform algorithms are derived from the discrete Fourier transform (DFT), which is formally defined as [13]:

$$X_k = \sum_{n=0}^{N-1} \omega_N^{nk} x_n \tag{1}$$

where $k = 0, \ldots, N-1$ and ω_N is the primitive root-of-unity, defined as $e^{-2\pi\sqrt{-1}/N}$ (often referred to as a "twiddle factor" in the context of fast Fourier transforms). X_k and x_n are sequences of complex numbers, X_k being the outputs in the frequency domain, and x_n being the inputs in the time or space domain.

A source of mild confusion in the FFT literature is the sign of the twiddle factor [57]; the definition in Equation 1 is considered to be the engineers view of the discrete Fourier transform, where the goal is to compute the coefficients of a discrete Fourier series. Mathematicians, on the other hand, typically view the DFT as a method of evaluating a polynomial at the powers of a primitive root of unity, and thus consider Equation 1 to be an inverse DFT [57]. Cooley and Tukey [16], Fiduccia [25] and Bernstein [9] are notable examples of those who adopt the mathematicians convention. This work adopts the engineer's view of a DFT, and thus the inverse discrete Fourier transform (IDFT) is defined by the following equation:

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} \omega_N^{-nk} X_k \tag{2}$$

where $n = 0, \ldots, N-1$. It should be noted that in some implementations, such as FFTW and the implementation presented in this thesis, the IDFT is actually non-normalised for reasons of efficiency; i.e., $\text{IFFT}(\text{FFT}(x)) = Nx$, thus avoiding division of each of the samples in time by N [34].

2.1 COOLEY-TUKEY

In 1965 James Cooley and John Tukey published a description of an economical algorithm for computing the DFT that became known as the Cooley-Tukey FFT, or simply *the* FFT due to its overwhelming popularity [16]. A later investigation by Heideman, Johnson and Burrus [41] revealed that the algorithm had actually been discovered several times in various forms prior to Cooley and Tukey, most notably by Gauss sometime around 1805 [14].

The algorithm recursively divides a transform of size $N = N_1 N_2$ into smaller DFTs of size N_1 and N_2 (where N is highly composite), reducing the time complexity from $O(n^2)$ to $O(n \log n)$ by exploiting common factors.

As the algorithm recursively divides a DFT, either N_1 or N_2 is typically a small factor, and is known as the radix. Small N_1 characterizes the algorithm as being decimation in time (DIT), otherwise the algorithm is decimation in frequency (DIF). If the radix changes between stages, then the algorithm is referred to as 'mixed-radix'.

For example, a radix-2 decimation-in-time algorithm decomposes Equation 1 into a sum over the even indices ($n = 2n_2$) and a sum over the odd indices ($n = 2n_2 + 1$):

$$X_k = \sum_{n_2=0}^{N/2-1} \omega_N^{(2n_2)k}\, x_{2n_2} + \sum_{n_2=0}^{N/2-1} \omega_N^{(2n_2+1)k}\, x_{2n_2+1} \tag{3}$$

The trigonometric coefficient in the second sum can be expanded to $\omega_N^{2n_2k}\omega_N^k$, and the term now common to both sums is simplified using the identity $\omega_N^{mnk} = \omega_{N/m}^{nk}$. Because one of the trigonometric coefficients in the second sum is constant with respect to the index variable, it may be factored out to obtain:

$$X_k = \sum_{n_2=0}^{N/2-1} \omega_{N/2}^{n_2k}\, x_{2n_2} + \omega_N^k \sum_{n_2=0}^{N/2-1} \omega_{N/2}^{n_2k}\, x_{2n_2+1} \tag{4}$$

where the two sums are now DFTs of the even indexed terms (x_{2n_2}) and the odd indexed terms (x_{2n_2+1}), which are combined with twiddle factor ω_N^k.

In order to compute the transform more efficiently, the Cooley-Tukey algorithm divides X_k into two halves, and exploits the periodicity of sub-transforms and symmetries in the trigonometric coefficients. Firstly, Equation 4 is rewritten as two halves with E_k substituted for the even sub-transform, and O_k substituted for the odd sub-transform:

$$\begin{aligned} X_k &= E_k + \omega_N^k O_k \\ X_{k+N/2} &= E_{k+N/2} + \omega_N^{k+N/2} O_{k+N/2} \end{aligned} \tag{5}$$

where $k = 0, \ldots, N/2 - 1$. Because of the periodicity property of the outputs of a DFT, $E_k = E_{k+N/2}$ and $O_k = O_{k+N/2}$, Equation 5 simplifies thus:

$$\begin{aligned} X_k &= E_k + \omega_N^k O_k \\ X_{k+N/2} &= E_k + \omega_N^{k+N/2} O_k \end{aligned} \tag{6}$$

And finally, by exploiting symmetries in the complex exponential function, namely that $\omega_N^{k+N/2} = -\omega_N^k$, the radix-2 DIT FFT can be expressed as:

$$\begin{aligned} X_k &= E_k + \omega_N^k O_k \\ X_{k+N/2} &= E_k - \omega_N^k O_k \end{aligned} \tag{7}$$

which makes it clear that each pair of outputs share common computation, approximately halving the number of arithmetic operations when compared to the DFT. But since the even and odd terms in Equation 7 are themselves DFTs that can be computed with the FFT, the savings compound with each stage of recursion. The total number of real arithmetic operations required to compute the radix-2 FFT can be expressed with the following recurrence relation:

$$T(n) = \begin{cases} 2T(n/2) + 5n - 6 & \text{for } n \geqslant 2 \\ 0 & \text{for } n = 1 \end{cases} \tag{8}$$

which is in $\Theta(n \log n)$.

2.2 SPLIT-RADIX

In 1968 a derivitive of the Cooley-Tukey algorithm broke the record for the lowest number of arithmetic operations for computing the DFT [23, 56, 75]. The algorithm was initially discovered by Yavne [78], but was not widely cited until 1984 when it was rediscovered by Duhamel and Hollman [22] and became known as the split-radix algorithm.

The split-radix algorithm improves the arithmetic complexity of the Cooley-Tukey algorithm by further decomposing the odd parts into odd-odd and odd-even parts, while the even parts are left alone because they have no multiplicative factor. More formally, Equation 1 can be re-written as three sums:

$$\begin{aligned} X_k &= \sum_{n_2=0}^{N/2-1} \omega_N^{2n_2k}\, x_{2n_2} + \sum_{n_4=0}^{N/4-1} \omega_N^{(4n_4+1)k}\, x_{4n_4+1} \\ &\quad + \sum_{n_4=0}^{N/4-1} \omega_N^{(4n_4+3)k}\, x_{4n_4+3} \end{aligned} \tag{9}$$

where $n = 4n_4 = 2n_2$. As with the Cooley-Tukey radix-2 example in Section 2.1, the trigonometric coefficients are expanded and simplified, and the terms constant with respect to the index variables factored out:

$$\begin{aligned} X_k &= \sum_{n_2=0}^{N/2-1} \omega_{N/2}^{n_2 k}\, x_{2n_2} + \omega_N^k \sum_{n_4=0}^{N/4-1} \omega_{N/4}^{n_4 k}\, x_{4n_4+1} \\ &\quad + \omega_N^{3k} \sum_{n_4=0}^{N/4-1} \omega_{N/4}^{n_4 k}\, x_{4n_4+3} \end{aligned} \tag{10}$$

By substituting the even sum with U_k (where $k = 0, \ldots, N/2-1$) and the odd sums with Z_k and Z'_k (where $k = 0, \ldots, N/4-1$), Equation 10 is simplified:

$$X_k = U_k + \omega_N^k Z_k + \omega_N^{3k} Z'_k \tag{11}$$

Computation can be factored out of Equation 11 by again exploiting periodicity in the sub-transforms and symmetries in the twiddle factors. Equation 11 is first expressed as an equation of four parts:

$$\begin{aligned} X_k &= U_k + \omega_N^k Z_k + \omega_N^{3k} Z'_k \\ X_{k+N/2} &= U_{k+N/2} + \omega_N^{k+N/2} Z_{k+N/2} + \omega_N^{3(k+N/2)} Z'_{k+N/2} \\ X_{k+N/4} &= U_{k+N/4} + \omega_N^{k+N/4} Z_{k+N/4} + \omega_N^{3(k+N/4)} Z'_{k+N/4} \\ X_{k+3N/4} &= U_{k+3N/4} + \omega_N^{k+3N/4} Z_{k+3N/4} + \omega_N^{3(k+3N/4)} Z'_{k+3N/4} \end{aligned} \tag{12}$$

where $k = 0, \ldots, N/4 - 1$. The periodicity properties of the sub-transforms can be expressed with the relationships $U_k = U_{k+N/2}$, $Z_k = Z_{k+N/4}$ and $Z'_k = Z'_{k+N/4}$. These are used to simplify Equation 12 thus:

$$\begin{aligned}
X_k &= U_k + \omega_N^k Z_k + \omega_N^{3k} Z'_k \\
X_{k+N/2} &= U_k + \omega_N^{k+N/2} Z_k + \omega_N^{3(k+N/2)} Z'_k \\
X_{k+N/4} &= U_{k+N/4} + \omega_N^{k+N/4} Z_k + \omega_N^{3(k+N/4)} Z'_k \\
X_{k+3N/4} &= U_{k+N/4} + \omega_N^{k+3N/4} Z_k + \omega_N^{3(k+3N/4)} Z'_k
\end{aligned} \tag{13}$$

Symmetries in the complex exponential function are again used to expose common computation among each part of the equation; hence

$$\begin{aligned}
X_k &= U_k + (\omega_N^k Z_k + \omega_N^{3k} Z'_k) \\
X_{k+N/2} &= U_k - (\omega_N^k Z_k + \omega_N^{3k} Z'_k) \\
X_{k+N/4} &= U_{k+N/4} - i(\omega_N^k Z_k - \omega_N^{3k} Z'_k) \\
X_{k+3N/4} &= U_{k+N/4} + i(\omega_N^k Z_k - \omega_N^{3k} Z'_k)
\end{aligned} \tag{14}$$

which, when recursively applied to the sub-transforms, results in the following recurrence relation for real arithmetic operations:

$$T(n) = \begin{cases} T(n/2) + 2T(n/4) + 6n - 4 & \text{for } n \geqslant 2 \\ 0 & \text{for } n = 1 \end{cases} \tag{15}$$

The exact solution $T(n) = 4n \log_2 n - 6n + 8$ for $n \geqslant 2$ was the best arithmetic complexity of all known FFT algorithms for over 30 years, until Van Buskirk was able to break the record in 2004 [55], as described in Section 2.3.

Van Buskirk's arithmetic complexity breakthrough was based on a variant of the split-radix algorithm known as the "conjugate-pair" algorithm [48] or the "-1 exponent" split-radix algorithm [9, 57]. In 1989 the conjugate-pair algorithm was published with the claim that it had broken the record set by Yavne in 1968 for the lowest number of arithmetic operations for computing the DFT [48]. Unfortunately the reduction in the number of arithmetic operations was due to an error in the author's analysis, and the algorithm was subsequently proven to have an arithmetic count equal to the original split-radix algorithm [37, 68, 53]. Despite initial claims about the arithmetic savings being discredited, the conjugate-pair algorithm has been used to reduce twiddle factor loads in software implementations of the FFT and fast Hartley transform (FHT) [48], and the algorithm was also recently used as the basis for an algorithm that *does* reduce the arithmetic operation count, as described in Section 2.3.

The difference between the conjugate-pair algorithm and the split-radix algorithm is in the decomposition of odd elements. In the standard split-radix algorithm, the odd elements are decomposed into two parts: x_{4n_4+1} and x_{4n_4+3} (see Equation 10), while in the conjugate-pair algorithm, the last sub-sequence is cyclically shifted by -4, where negative indices wrap around (i.e., $x_{-1} = x_{N-1}$). The result of this cyclic shift is that twiddle factors are now conjugate pairs. Formally, the conjugate-pair algorithm is defined as:

$$\begin{aligned} X_k &= \sum_{n_2=0}^{N/2-1} \omega_{N/2}^{n_2 k}\, x_{2n_2} + \omega_N^k \sum_{n_4=0}^{N/4-1} \omega_{N/4}^{n_4 k}\, x_{4n_4+1} \\ &\quad + \omega_N^{-k} \sum_{n_4=0}^{N/4-1} \omega_{N/4}^{n_4 k}\, x_{4n_4-1} \end{aligned} \tag{16}$$

As with the ordinary split-radix algorithm, a DIT decomposition of the conjugate-pair algorithm can be expressed as a system of equations:

$$\begin{aligned}
X_k &= U_k + (\omega_N^k Z_k + \omega_N^{-k} Z'_k) \\
X_{k+N/2} &= U_k - (\omega_N^k Z_k + \omega_N^{-k} Z'_k) \\
X_{k+N/4} &= U_{k+N/4} - i(\omega_N^k Z_k - \omega_N^{-k} Z'_k) \\
X_{k+3N/4} &= U_{k+N/4} + i(\omega_N^k Z_k - \omega_N^{-k} Z'_k)
\end{aligned} \tag{17}$$

where $k = 0, \ldots, N/4 - 1$. As can be seen, the trigonometric coefficients are conjugates – a feature that can be exploited to reduce twiddle factor loads.

2.3 TANGENT

In 2004, some thirty years after Yavne set the record for the lowest arithmetic operation count, Van Buskirk posted software to Usenet that had asymptotically reduced the arithmetic operation count by about 6%. Three papers were subsequently published [55, 9, 47] with differing explanations on how to achieve the lowest arithmetic operation count initially demonstrated by Van Buskirk.

Although all three papers describe algorithms that achieve the lowest arithmetic operation count in the same way, and thus can be considered to be different views of the same algorithm, all three papers refer to the algorithms by different names. Lundy and Van Buskirk [55] refer to their algorithm as "scaled odd tail FFT", Bernstein [9] describes an algorithm named "tangent FFT", while Johnson and Frigo [47] refer to the algorithm by various names. Many works have cited Johnson and Frigo for the algorithm [15]. Of these names, "tangent FFT" is used in this work because it is the most descriptive; scaling the twiddle factors into tangent form was the linchpin of Van Buskirk's breakthrough in arithmetic complexity.

Bernstein expresses a DIF decomposition of the tangent FFT in a very concise but somewhat obscure polynomial form that was first practised by Fiduccia [25]. In order to be consistent with earlier sections, a DIT decomposition of the tangent FFT using linear functions will be described in this section.[1] While the polynomial form is more elegant and concise, expressing the FFT in terms of linear functions has the advantage of mapping to software or hardware more directly.

The key to the tangent FFT is Van Buskirk's observation that if the trigonometric constant $\omega_N^k = \cos\theta + i\sin\theta$ is factored as $(1 + i\tan\theta)\cos\theta$ or $(\cot\theta + i)\sin\theta$, the multiplication by $\cos\theta$ or $\sin\theta$ can sometimes be absorbed elsewhere in the computation, assuming the constants are precomputed, and the remaining multiplication by constants of the form $\pm(1 + i\tan\theta)$ or $\pm(\cot\theta + i)$ now only costs four floating point operations instead of six, assuming the usual scheme of complex multiplication using four multiply and two add operations.

Firstly, consider the conjugate-pair FFT being recursively scaled by a wavelet $s_{N,k}$:

$$\frac{X_k}{s_{N,k}} = U_k \left(\frac{s_{N/2,k}}{s_{N,k}}\right) + \omega_N^k \left(\frac{s_{N/4,k}}{s_{N,k}}\right) Z_k + \omega_N^{-k} \left(\frac{s_{N/4,k}}{s_{N,k}}\right) Z'_k \qquad (18)$$

for $k = 0, \ldots, N/4 - 1$, and where U_k is evaluated with $X_k/s_{N/2,k}$, and Z_k and Z'_k are evaluated with $X_k/s_{N/4,k}$.

1 Although derived differently, the underlying structure presented here is identical to the network transpose of Bernstein's tangent FFT. In contrast to Johnson and Frigo's algorithm of four sub-transforms, the view presented here uses only one sub-transform and a scaled split-radix transform.

The wavelet is crafted such that it is periodic in k (i.e., $s_{N,k} = s_{N,k+N/4}$) and $\omega_N^k(s_{N/4,k}/s_{N,k})$ is of the form $\pm(1 + i\tan\theta)$ or $\pm(\cot\theta + i)$. Bernstein defines the wavelet as [9]:

$$s_{N,k} = \prod_{\ell \geqslant 0} \max\left\{\left|\cos\left(\frac{4^\ell 2\pi k}{N}\right)\right|, \left|\sin\left(\frac{4^\ell 2\pi k}{N}\right)\right|\right\} \tag{19}$$

Multiplying Z_k and Z'_k by the scaled constants saves a total of four floating point operations, while scaling U_k costs four operations, resulting in no gain or loss. But the cost of scaling the result back to X_k is about 2N real operations. In order to realize a *reduction* in the number of floating point operations, the split-radix FFT is decomposed further, so that the extra operations can be absorbed into constants in the sub-transform. Starting with the unscaled split-radix FFT (see Equation 9), the sum over the x_{2n_2} terms is itself decomposed with a split-radix decomposition into x_{4n_4}, x_{8n_8+2} and x_{8n_8+6}, resulting in a DFT of five sums:

$$\begin{aligned} X_k &= \sum_{n_4=0}^{N/4-1} \omega_N^{4n_4k}\, x_{4n_4} \\ &+ \sum_{n_8=0}^{N/8-1} \omega_N^{(8n_8+2)k}\, x_{8n_8+2} + \sum_{n_8=0}^{N/8-1} \omega_N^{(8n_8+6)k}\, x_{8n_8+6} \\ &+ \sum_{n_4=0}^{N/4-1} \omega_N^{(4n_4+1)k}\, x_{4n_4+1} + \sum_{n_4=0}^{N/4-1} \omega_N^{(4n_4+3)k}\, x_{4n_4+3} \end{aligned} \tag{20}$$

where $n = 4n_4 = 8n_8$. As with earlier decompositions, invariants are factored out to obtain:

$$\begin{aligned} X_k &= \sum_{n_4=0}^{N/4-1} \omega_N^{4n_4k}\, x_{4n_4} \\ &+ \omega_N^{2k} \sum_{n_8=0}^{N/8-1} \omega_{N/8}^{n_8k}\, x_{8n_8+2} + \omega_N^{6k} \sum_{n_8=0}^{N/8-1} \omega_{N/8}^{n_8k}\, x_{8n_8+6} \\ &+ \omega_N^{k} \sum_{n_4=0}^{N/4-1} \omega_{N/4}^{n_4k}\, x_{4n_4+1} + \omega_N^{3k} \sum_{n_4=0}^{N/4-1} \omega_{N/4}^{n_4k}\, x_{4n_4+3} \end{aligned} \tag{21}$$

Following from the conjugate-pair split-radix algorithm, x_{8n_8+6} is shifted cyclically by -8 and x_{4n_4+3} is shifted cyclically by -4 to obtain:

$$\begin{aligned} X_k &= \sum_{n_4=0}^{N/4-1} \omega_N^{4n_4k}\, x_{4n_4} \\ &+ \omega_N^{2k} \sum_{n_8=0}^{N/8-1} \omega_{N/8}^{n_8k}\, x_{8n_8+2} + \omega_N^{-2k} \sum_{n_8=0}^{N/8-1} \omega_{N/8}^{n_8k}\, x_{8n_8-2} \\ &+ \omega_N^{k} \sum_{n_4=0}^{N/4-1} \omega_{N/4}^{n_4k}\, x_{4n_4+1} + \omega_N^{-k} \sum_{n_4=0}^{N/4-1} \omega_{N/4}^{n_4k}\, x_{4n_4-1} \end{aligned} \tag{22}$$

where $x_{-n} = x_{N-n}$. Note that the sums over x_{8n_8+2} and x_{8n_8-2} are multiplied by constants that are now conjugate pairs, as are the sums over x_{4n_4+1} and x_{4n_4-1}.

The sum over x_{4n_4} is now substituted with U_k (where $k = 0, \ldots, N/4 - 1$), while the sums over x_{8n_8+2} and x_{8n_n-2} are respectively substituted with Y_k and Y'_k (where $k = 0, \ldots, N/8 - 1$) and the sums over x_{4n_4+1} and

x_{4n_4-1} respectively substituted with Z_k and Z'_k (where $k = 0, \dots, N/4 - 1$), simplifying Equation 22 thus:

$$X_k = U_k + \omega_N^{2k} Y_k + \omega_N^{-2k} Y'_k + \omega_N^{k} Z_k + \omega_N^{-k} Z'_k \tag{23}$$

As with earlier examples, computation is factored out of Equation 23 by exploiting periodicity in the sub-transforms and symmetries in the twiddle factors. Equation 23 is first expressed as a parametric equation of eight parts:

$$\begin{aligned} X_{k+pN} &= U_{k+pN} \\ &+ \omega_N^{2(k+pN)} Y_{k+pN} + \omega_N^{-2(k+pN)} Y'_{k+pN} \\ &+ \omega_N^{k+pN} Z_{k+pN} + \omega_N^{-(k+pN)} Z'_{k+pN} \end{aligned} \tag{24}$$

where $k = 0, \dots, N/8 - 1$ and $\forall p \in \left\{0, \frac{1}{2}, \frac{1}{4}, \frac{3}{4}, \frac{1}{8}, \frac{3}{8}, \frac{5}{8}, \frac{7}{8}\right\}$. By exploiting periodicity in the sub-transforms:

$$\begin{aligned} U_k &= U_{k+N/4} \\ Y_k &= Y_{k+N/8} \\ Y'_k &= Y'_{k+N/8} \\ Z_k &= Z_{k+N/4} \\ Z'_k &= Z'_{k+N/4} \end{aligned} \tag{25}$$

and the following symmetries in the twiddle factors:

$$\begin{aligned}
\omega_N^{2k} &= \omega_N^{2(k+N/2)} = -\omega_N^{2(k+N/4)} = -\omega_N^{2(k+3N/4)} \\
&= -i\omega_N^{2(k+N/8)} = i\omega_N^{2(k+3N/8)} = -i\omega_N^{2(k+5N/8)} \\
&= i\omega_N^{2(k+7N/8)} \\
\omega_N^{-2k} &= \omega_N^{-2(k+N/2)} = -\omega_N^{-2(k+N/4)} = -\omega_N^{-2(k+3N/4)} \\
&= i\omega_N^{-2(k+N/8)} = -i\omega_N^{-2(k+3N/8)} = i\omega_N^{-2(k+5N/8)} \\
&= -i\omega_N^{-2(k+7N/8)} \\
\omega_N^{k} &= -\omega_N^{k+N/2} = -i\omega_N^{k+N/4} = i\omega_N^{k+3N/4} \\
\omega_N^{-k} &= -\omega_N^{-(k+N/2)} = i\omega_N^{-(k+N/4)} = -i\omega_N^{-(k+3N/4)} \\
\omega_N^{k+N/8} &= -i\omega_N^{k+3N/8} = -\omega_N^{k+5N/8} = i\omega_N^{k+7N/8} \\
\omega_N^{-(k+N/8)} &= i\omega_N^{-(k+3N/8)} = -\omega_N^{-(k+5N/8)} = -i\omega_N^{-(k+7N/8)}
\end{aligned} \tag{26}$$

Equation 24 is rewritten thus:

$$\begin{aligned}
X_k &= U_k + (\omega_N^{2k}Y_k + \omega_N^{-2k}Y'_k) + (\omega_N^{k}Z_k + \omega_N^{-k}Z'_k) \\
X_{k+N/2} &= U_k + (\omega_N^{2k}Y_k + \omega_N^{-2k}Y'_k) - (\omega_N^{k}Z_k + \omega_N^{-k}Z'_k) \\
X_{k+N/4} &= U_k - (\omega_N^{2k}Y_k + \omega_N^{-2k}Y'_k) - i(\omega_N^{k}Z_k - \omega_N^{-k}Z'_k) \\
X_{k+3N/4} &= U_k - (\omega_N^{2k}Y_k + \omega_N^{-2k}Y'_k) + i(\omega_N^{k}Z_k - \omega_N^{-k}Z'_k) \\
X_{k+N/8} &= U_{k+N/8} - i(\omega_N^{2k}Y_k - \omega_N^{-2k}Y'_k) \\
&\quad +(\omega_N^{k+N/8}Z_{k+N/8} + \omega_N^{-(k+N/8)}Z'_{k+N/8}) \\
X_{k+3N/8} &= U_{k+N/8} + i(\omega_N^{2k}Y_k - \omega_N^{-2k}Y'_k) \\
&\quad -i(\omega_N^{k+N/8}Z_{k+N/8} - \omega_N^{-(k+N/8)}Z'_{k+N/8}) \\
X_{k+5N/8} &= U_{k+N/8} - i(\omega_N^{2k}Y_k - \omega_N^{-2k}Y'_k) \\
&\quad -(\omega_N^{k+N/8}Z_{k+N/8} + \omega_N^{-(k+N/8)}Z'_{k+N/8}) \\
X_{k+7N/8} &= U_{k+N/8} + i(\omega_N^{2k}Y_k - \omega_N^{-2k}Y'_k) \\
&\quad +i(\omega_N^{k+N/8}Z_{k+N/8} - \omega_N^{-(k+N/8)}Z'_{k+N/8})
\end{aligned} \tag{27}$$

By applying terms with the appropriate scaling, viz. $\alpha_{N,k} = s_{N/4,k}/s_{N,k}$, $\beta_{N,k} = s_{N/8,k}/s_{N/2,k}$, $\gamma_{N,k} = s_{N/4,k+N/8}/s_{N,k+N/8}$, $\delta_{N,k} = s_{N/2,k}/s_{N,k}$ and $\epsilon_{N,k} = s_{N/2,k+N/8}/s_{N,k+N/8}$, Equation 27 now becomes:

$$\begin{aligned}
X_k/s_{N,k} &= U_k\alpha_{N,k} + (\beta_{N,k}\omega_N^{2k}Y_k + \beta_{N,k}\omega_N^{-2k}Y'_k)\delta_{N,k} \\
&\quad +(\alpha_{N,k}\omega_N^{k}Z_k + \alpha_{N,k}\omega_N^{-k}Z'_k) \\
X_{k+N/2}/s_{N,k} &= U_k\alpha_{N,k} + (\beta_{N,k}\omega_N^{2k}Y_k + \beta_{N,k}\omega_N^{-2k}Y'_k)\delta_{N,k} \\
&\quad -(\alpha_{N,k}\omega_N^{k}Z_k + \alpha_{N,k}\omega_N^{-k}Z'_k) \\
X_{k+N/4}/s_{N,k} &= U_k\alpha_{N,k} - (\beta_{N,k}\omega_N^{2k}Y_k + \beta_{N,k}\omega_N^{-2k}Y'_k)\delta_{N,k} \\
&\quad -i(\alpha_{N,k}\omega_N^{k}Z_k - \alpha_{N,k}\omega_N^{-k}Z'_k) \\
X_{k+3N/4}/s_{N,k} &= U_k\alpha_{N,k} - (\beta_{N,k}\omega_N^{2k}Y_k + \beta_{N,k}\omega_N^{-2k}Y'_k)\delta_{N,k} \\
&\quad +i(\alpha_{N,k}\omega_N^{k}Z_k - \alpha_{N,k}\omega_N^{-k}Z'_k) \\
X_{k+N/8}/s_{N,k} &= U_{k+N/8}\gamma_{N,k} - i(\beta_{N,k}\omega_N^{2k}Y_k - \beta_{N,k}\omega_N^{-2k}Y'_k)\epsilon_{N,k} \\
&\quad +(\gamma_{N,k}\omega_N^{k+N/8}Z_{k+N/8} + \gamma_{N,k}\omega_N^{-(k+N/8)}Z'_{k+N/8}) \\
X_{k+3N/8}/s_{N,k} &= U_{k+N/8}\gamma_{N,k} + i(\beta_{N,k}\omega_N^{2k}Y_k - \beta_{N,k}\omega_N^{-2k}Y'_k)\epsilon_{N,k} \\
&\quad -i(\gamma_{N,k}\omega_N^{k+N/8}Z_{k+N/8} - \gamma_{N,k}\omega_N^{-(k+N/8)}Z'_{k+N/8}) \\
X_{k+5N/8}/s_{N,k} &= U_{k+N/8}\gamma_{N,k} - i(\beta_{N,k}\omega_N^{2k}Y_k - \beta_{N,k}\omega_N^{-2k}Y'_k)\epsilon_{N,k} \\
&\quad -(\gamma_{N,k}\omega_N^{k+N/8}Z_{k+N/8} + \gamma_{N,k}\omega_N^{-(k+N/8)}Z'_{k+N/8}) \\
X_{k+7N/8}/s_{N,k} &= U_{k+N/8}\gamma_{N,k} + i(\beta_{N,k}\omega_N^{2k}Y_k - \beta_{N,k}\omega_N^{-2k}Y'_k)\epsilon_{N,k} \\
&\quad +i(\gamma_{N,k}\omega_N^{k+N/8}Z_{k+N/8} - \gamma_{N,k}\omega_N^{-(k+N/8)}Z'_{k+N/8})
\end{aligned} \tag{28}$$

Assuming that the scaling factors are absorbed into precomputed twiddle factors where possible (e.g., $\alpha_{N,k}\omega_N^k$ is a single precomputed constant), computing Equation 28 requires about $(68/8)N$ real operations, in contrast to $(72/8)N$ operations for Equation 27. Further assuming that operations are skipped in the cases where precomputed constants are of the form ± 1 or $\pm i$, a further 28 real operations are saved in Equation 28. Thus the arithmetic cost of Equation 28 can be expressed with the following recurrence relation:

$$T(n) = \begin{cases} 3T(n/4) + 2T(n/8) + \max\{n-12, 0\} + 7.5n - 16 & \text{for } n \geqslant 8 \\ 16 & \text{for } n = 4 \\ 4 & \text{for } n = 2 \\ 0 & \text{for } n = 1 \end{cases} \tag{29}$$

Bernstein gives the exact solution of Equation 29 as [9]:

$$\begin{aligned} T(n) &= (34/9)n\log_2 n - (142/27)n \\ &\quad -(2/9)(-1)^{\log_2 n}\log_2 n + (7/27)(-1)^{\log_2 n} + 7 \end{aligned} \tag{30}$$

for $n \geqslant 2$.

Equation 28 is scaled by $s_{N,k}$, but if the application is convolution in frequency, the scaling could be absorbed into the filter, and the cost of scaling the results back to X_k avoided. Otherwise, a split-radix FFT can be used to change basis, absorbing the scaling into the twiddle factors of the x_{4n_4+1} and x_{4n_4-1} terms:

$$\begin{aligned} X_k &= U_k + (s_{N,k}\omega_N^k Z_k + s_{N,k}\omega_N^{-k} Z'_k) \\ X_{k+N/2} &= U_k - (s_{N,k}\omega_N^k Z_k + s_{N,k}\omega_N^{-k} Z'_k) \\ X_{k+N/4} &= U_{k+N/4} - i(s_{N,k}\omega_N^k Z_k - s_{N,k}\omega_N^{-k} Z'_k) \\ X_{k+3N/4} &= U_{k+N/4} + i(s_{N,k}\omega_N^k Z_k - s_{N,k}\omega_N^{-k} Z'_k) \end{aligned} \tag{31}$$

where Z_k and Z'_k are now recursively computed with the tangent FFT of Equation 28, and the U_k terms are themselves computed with Equation 31. The arithmetic cost of computing the tangent FFT in the traditional basis is thus expressed:

$$T'(n) = \begin{cases} T'(n/2) + 2T(n/4) + 3n + \max\{3n - 16, 0\} & \text{for } n \geqslant 4 \\ 4 & \text{for } n = 2 \\ 0 & \text{for } n = 1 \end{cases} \tag{32}$$

giving rise to Van Buskirk's exact operation count of [55]:

$$\begin{aligned} T'(n) &= (34/9)n\log_2 n - (124/27)n - 2\log_2 n \\ &\quad -(2/9)(-1)^{\log_2 n}\log_2 n + (16/27)(-1)^{\log_2 n} + 8 \end{aligned} \tag{33}$$

for $n \geqslant 2$.

3 IMPLEMENTATION DETAILS

"Anyone can build a fast CPU. The trick is to build a fast system."

— Seymour Cray

This Chapter complements the mathematical perspective of Chapter 2 with a more focused view of the low level details that are relevant to efficient implementation on SIMD microprocessors. These techniques are widely practised by today's state of the art implementations, and form the basis for more advanced techniques presented in later chapters.

3.1 SIMPLE PROGRAMS

The FFT equations of Chapter 2 can be succinctly expressed as microprocessor programs that are depth first recursive. For example, Equation 7 divides a size N transform into two size $N/2$ transforms, which in turn are divided into size $N/4$ transforms. This recursion continues until the base case of two size 1 transforms is reached, where the two smaller sub-transforms are then combined into a size 2 sub-transform, and then two completed size 2 transforms are combined into a size 4 transform, and so on, until the size N transform is complete.

Computing the FFT with such a depth first traversal has an important advantage in terms of memory locality: at any point during the traversal, the two completed sub-transforms that compose a larger sub-

Algorithm 1 DITFFT2$_N(x_n)$

```
Require: An array of complex numbers $x_{n=0,...,N-1}$ where $N = 2^m$ and $m \in \mathbb{N}^0$.
Ensure: $X_{k=0,...,N-1}$ = the DFT of $x_{n=0,...,N-1}$.
1:  if $N = 1$ then
2:      return $x_0$
3:  else
4:      $E_{k_2=0,...,N/2-1} \leftarrow$ DITFFT2$_{N/2}(x_{2n_2})$
5:      $O_{k_2=0,...,N/2-1} \leftarrow$ DITFFT2$_{N/2}(x_{2n_2+1})$
6:      for $k = 0$ to $N/2 - 1$ do
7:          $X_k \leftarrow E_k + \omega_N^k O_k$
8:          $X_{k+N/2} \leftarrow E_k - \omega_N^k O_k$
9:      end for
10:     return $X_k$
11: end if
```

transform will still be in the closest level of the memory hierarchy in which they fit (see, i.a., [73] and [46]). In contrast, a breadth first traversal of a sufficiently large transform could force data out of cache during every pass (ibid.).

Many implementations of the FFT require a bit-reversal permutation of either the input or the output data, but a depth first recursive algorithm implicitly performs the permutation during recursion. The bit-reversal permutation is an expensive computation, and despite being the subject of hundreds of research papers over the years, it can easily account for a large fraction of the FFTs runtime – more so for conjugate-pair algorithms with their "twisted" bit-reversal permutation. Such permutations will be encountered in later sections, but for the mean time it should be noted that the algorithms in this chapter do not require bit-reversal permutations – the input and output are in natural order.

IMPLEMENTATION	MACHINE	RUNTIME
Danielson-Lanczos, 1942 [18]	Human	140 minutes
Cooley-Tukey, 1965 [16]	IBM 7094	$\sim$ 10.5 ms
Listing 20, 2011	Macbook Air 4,2	$\sim$ 440 μs

Table 1: Performance of simple radix-2 FFT (see Listing 20) from a historical perspective, for size 64 real FFT

3.1.1 *Radix-2*

A recursive depth first implementation of the Cooley-Tukey radix-2 decimation in time FFT is described with pseudocode in Algorithm 1, and an implementation coded in C with only the most basic optimization – avoiding multiply operations where ω_N^0 is unity in the first iteration of the loop – is included in Appendix A (Listing 20). Even when compiled with a state-of-the-art auto-vectorizing compiler,[1] the code achieves poor performance on modern microprocessors, and is useful only as a baseline reference.[2]

However it is worth noting that when considered from a historical perspective, the performance *does* seem impressive – as shown in Table 1. The runtimes in Table 1 are approximate; the Cooley-Tukey figure is roughly extrapolated from the floating point operations per second (FLOPS) count of a size 2048 complex transform given in their 1965 paper [16]; and the speed of the reference implementation is derived from the runtime of a size 64 complex FFT (again, based on the FLOPS count). Furthermore, the precision differs between the results; Danielson and Lanczos computed the DFT to 3–5 significant figures (possibly with the aid of slide rules or adding machines), while the other results were

1 Intel(R) C Intel(R) 64 Compiler XE for applications running on Intel(R) 64, Version 12.1.0.038 Build 20110811.
2 Chapter 6 contains a full account of the benchmark methods.

computed with the host machines' implementation of single precision floating point arithmetic.

The runtime performance of the FFT has improved by about seven orders of magnitude in 70 years, and this can largely be attributed to the computing machines of the day being generally faster. The following sections and chapters will show that the performance can be further improved by over two orders of magnitude if the algorithm is enhanced with optimizations that are amenable to the underlying machine.

3.1.2 *Split-radix*

Algorithm 2 SPLITFFT$_N(x_n)$

```
Require: An array of complex numbers $x_{n=0,...,N-1}$ where $N = 2^m$ and
    $m \in \mathbb{N}^0$.
Ensure: $X_{k=0,...,N-1}$ = the DFT of $x_{n=0,...,N-1}$.
 1: if $N = 1$ then
 2:     return $x_0$
 3: else if $N = 2$ then
 4:     $X_0 \leftarrow x_0 + x_1$
 5:     $X_1 \leftarrow x_0 - x_1$
 6:     return $X_k$
 7: else
 8:     $U_{k_2=0,...,N/2-1} \leftarrow$ SPLITFFT$_{N/2}(x_{2n_2})$
 9:     $Z_{k_4=0,...,N/4-1} \leftarrow$ SPLITFFT$_{N/4}(x_{4n_4+1})$
10:     $Z'_{k_4=0,...,N/4-1} \leftarrow$ SPLITFFT$_{N/4}(x_{4n_4+3})$
11:     for $k = 0$ to $N/4 - 1$ do
12:         $X_k \leftarrow U_k + (\omega_N^k Z_k + \omega_N^{3k} Z'_k)$
13:         $X_{k+N/2} \leftarrow U_k - (\omega_N^k Z_k + \omega_N^{3k} Z'_k)$
14:         $X_{k+N/4} \leftarrow U_{k+N/4} - i(\omega_N^k Z_k - \omega_N^{3k} Z'_k)$
15:         $X_{k+3N/4} \leftarrow U_{k+N/4} + i(\omega_N^k Z_k - \omega_N^{3k} Z'_k)$
16:     end for
17:     return $X_k$
18: end if
```

As was the case with the radix-2 FFT in the previous section, the split-radix FFT neatly maps from the system of linear functions given in Equation 14 to the pseudocode of Algorithm 2, and then to the C implementation included in Appendix A (Listing 21).

Algorithm 2 explicitly handles the base case for $N = 2$, to accommodate not only size 2 transforms, but also size 4 and size 8 transforms (and all larger transforms that are ultimately composed of these smaller transforms). A size 4 transform is divided into two size 1 sub-transforms and one size 2 transform, which cannot be further divided by the split-radix algorithm, and so it must be handled as a base case. Likewise with the size 8 transform that divides into one size 4 sub-transform and two size 2 sub-transforms: the size 2 sub-transforms cannot be further decomposed with the split-radix algorithm.

Also note that two twiddle factors, viz. ω_N^k and $\omega_N^3 k$, are required for the split-radix decomposition; this is an advantage compared to the radix-2 decomposition which would require four twiddle factors for the same size 4 transform.

3.1.3 *Conjugate-pair*

From a pseudocode perspective, there is little difference between the ordinary split-radix algorithm and the conjugate-pair algorithm (see Algorithm 3). In line 10, the x_{4n_4+3} terms have been shifted cyclically by -4 to x_{4n_4-1}, and in lines 12-15, the coefficient of Z'_k has been shifted cyclically from ω_N^{3k} to ω_N^{-k}.

The source code (see Listing 22) has a few subtle differences that are not revealed in the pseudocode. The pseudocode in Algorithm 3 requires an array of complex numbers x_n for input, but the code in Listing 22 requires a reference to an array of complex numbers with a stride[3] – this

3 A stride of n would indicate that only every n^{th} term is being referred to.

Algorithm 3 $\textsc{conjfft}_N(x_n)$

Require: An array of complex numbers $x_{n=0,\ldots,N-1}$ where $N = 2^m$ and $m \in \mathbb{N}^0$.
Ensure: $X_{k=0,\ldots,N-1}$ = the DFT of $x_{n=0,\ldots,N-1}$.

```
 1: if $N = 1$ then
 2:     return $x_0$
 3: else if $N = 2$ then
 4:     $X_0 \leftarrow x_0 + x_1$
 5:     $X_1 \leftarrow x_0 - x_1$
 6:     return $X_k$
 7: else
 8:     $U_{k_2=0,\ldots,N/2-1} \leftarrow \textsc{conjfft}_{N/2}(x_{2n_2})$
 9:     $Z_{k_4=0,\ldots,N/4-1} \leftarrow \textsc{conjfft}_{N/4}(x_{4n_4+1})$
10:     $Z'_{k_4=0,\ldots,N/4-1} \leftarrow \textsc{conjfft}_{N/4}(x_{4n_4-1})$
11:     for $k = 0$ to $N/4 - 1$ do
12:         $X_k \leftarrow U_k + (\omega_N^k Z_k + \omega_N^{-k} Z'_k)$
13:         $X_{k+N/2} \leftarrow U_k - (\omega_N^k Z_k + \omega_N^{-k} Z'_k)$
14:         $X_{k+N/4} \leftarrow U_{k+N/4} - i(\omega_N^k Z_k - \omega_N^{-k} Z'_k)$
15:         $X_{k+3N/4} \leftarrow U_{k+N/4} + i(\omega_N^k Z_k - \omega_N^{-k} Z'_k)$
16:     end for
17:     return $X_k$
18: end if
```

avoids copying x_n into three separate arrays, viz. x_{2n_2}, x_{4n_4+1} and x_{4n_4-1}, with every invocation of Algorithm 3. The subtle complication arises due to the cyclic shifting of the x_{4n_4-1} term; the negative shifting results in pointers that reference data before the start of the array. Rather than immediately wrapping the references around to end of the array such that they always point to valid data, the recursion proceeds until the base cases are reached before any adjustment is performed. Once at the leaves of the recursion, any pointers that reference data lying before the start of the input array are incremented by N elements,[4] so as to point to the correct data.

4 In this case, N refers to the size of the outer most transform rather than the size of the sub-transform.

Algorithm 4 TANGENTFFT4$_N(x_n)$

```
Require: An array of complex numbers $x_{n=0,...,N-1}$ where $N = 2^m$ and $m \in \mathbb{N}^0$.
Ensure: $X_{k=0,...,N-1}$ = the DFT of $x_{n=0,...,N-1}$.
1: if $N = 1$ then
2:     return $x_0$
3: else if $N = 2$ then
4:     $X_0 \leftarrow x_0 + x_1$
5:     $X_1 \leftarrow x_0 - x_1$
6:     return $X_k$
7: else
8:     $U_{k_2=0,...,N/2-1} \leftarrow$ TANGENTFFT4$_{N/2}(x_{2n_2})$
9:     $Z_{k_4=0,...,N/4-1} \leftarrow$ TANGENTFFT8$_{N/4}(x_{4n_4+1})$
10:    $Z'_{k_4=0,...,N/4-1} \leftarrow$ TANGENTFFT8$_{N/4}(x_{4n_4-1})$
11:    for $k = 0$ to $N/4 - 1$ do
12:        $X_k \leftarrow U_k + (\omega_N^k s_{N/4,k} Z_k + \omega_N^{-k} s_{N/4,k} Z'_k)$
13:        $X_{k+N/2} \leftarrow U_k - (\omega_N^k s_{N/4,k} Z_k + \omega_N^{-k} s_{N/4,k} Z'_k)$
14:        $X_{k+N/4} \leftarrow U_{k+N/4} - i(\omega_N^k s_{N/4,k} Z_k - \omega_N^{-k} s_{N/4,k} Z'_k)$
15:        $X_{k+3N/4} \leftarrow U_{k+N/4} + i(\omega_N^k s_{N/4,k} Z_k - \omega_N^{-k} s_{N/4,k} Z'_k)$
16:    end for
17:    return $X_k$
18: end if
```

3.1.4 *Tangent*

The tangent FFT is divided into two functions, described with pseudocode in Algorithm 4 and Algorithm 5. If the tangent FFT is computed prior to convolution in the frequency domain, the convolution kernel can absorb the final scaling and only Algorithm 5 is required. Otherwise Algorithm 4 is used as a wrapper around Algorithm 5 to perform the rescaling, and the result X_k is in the correct basis.

Algorithm 4 is similar to Algorithm 3, except that Z_k and Z'_k are computed with Algorithm 5, and thus scaled by $1/s_{N/4,k}$. Because Z_k and Z'_k are respectively multiplied by the coefficients ω_N^k and ω_N^{-k}, the results are scaled into the correct basis by absorbing $s_{N/4,k}$ into the coefficients.

Algorithm 5 TANGENTFFT8$_N(x_n)$

Require: An array of complex numbers $x_{n=0,\dots,N-1}$ where $N = 2^m$ and $m \in \mathbb{N}^0$.
Ensure: $X_{k=0,\dots,N-1} = \mathrm{DFT}(x_{n=0,\dots,N-1})/s_{N,k}$.

```
 1: if $N = 1$ then
 2:     return $x_0$
 3: else if $N = 2$ then
 4:     $X_0 \leftarrow x_0 + x_1$
 5:     $X_1 \leftarrow x_0 - x_1$
 6:     return $X_k$
 7: else if $N = 4$ then
 8:     $T_{k_2=0,1} \leftarrow$ TANGENTFFT8$_2(x_{2n_2})$
 9:     $T'_{k_2=0,1} \leftarrow$ TANGENTFFT8$_2(x_{2n_2+1})$
10:     $X_0 \leftarrow T_0 + T'_0$
11:     $X_2 \leftarrow T_0 - T'_0$
12:     $X_1 \leftarrow T_1 + T'_1$
13:     $X_3 \leftarrow T_1 - T'_1$
14:     return $X_k$
15: else
16:     $U_{k_4=0,\dots,N/4-1} \leftarrow$ TANGENTFFT8$_{N/4}(x_{4n_4})$
17:     $Y_{k_8=0,\dots,N/8-1} \leftarrow$ TANGENTFFT8$_{N/8}(x_{8n_8+2})$
18:     $Y'_{k_8=0,\dots,N/8-1} \leftarrow$ TANGENTFFT8$_{N/8}(x_{8n_8-2})$
19:     $Z_{k_4=0,\dots,N/4-1} \leftarrow$ TANGENTFFT8$_{N/4}(x_{4n_4+1})$
20:     $Z'_{k_4=0,\dots,N/4-1} \leftarrow$ TANGENTFFT8$_{N/4}(x_{4n_4-1})$
21:     for $k = 0$ to $N/8 - 1$ do
22:         $\alpha_{N,k} \leftarrow s_{N/4,k}/s_{N,k}$
23:         $\beta_{N,k} \leftarrow s_{N/8,k}/s_{N/2,k}$
24:         $\gamma_{N,k} \leftarrow s_{N/4,k+N/8}/s_{N,k+N/8}$
25:         $\delta_{N,k} \leftarrow s_{N/2,k}/s_{N,k}$
26:         $\epsilon_{N,k} \leftarrow s_{N/2,k+N/8}/s_{N,k+N/8}$
27:         $\Omega_0 \leftarrow \omega_N^k * \alpha_{N,k}$
28:         $\Omega_1 \leftarrow \omega_N^{k+N/8} * \gamma_{N,k}$
29:         $\Omega_2 \leftarrow \omega_N^{2k} * \beta_{N,k}$
30:         $T_0 \leftarrow (\Omega_2 Y_k + \overline{\Omega}_2 Y_k) * \delta_{N,k}$
31:         $T_1 \leftarrow i(\Omega_2 Y_k - \overline{\Omega}_2 Y_k) * \epsilon_{N,k}$
32:         $X_k \leftarrow U_k * \alpha_{N,k} + T_0 + (\Omega_0 Z_k + \overline{\Omega}_0 Z'_k)$
33:         $X_{k+N/2} \leftarrow U_k * \alpha_{N,k} + T_0 - (\Omega_0 Z_k + \overline{\Omega}_0 Z'_k)$
34:         $X_{k+N/4} \leftarrow U_k * \alpha_{N,k} - T_0 - i(\Omega_0 Z_k - \overline{\Omega}_0 Z'_k)$
35:         $X_{k+3N/4} \leftarrow U_k * \alpha_{N,k} - T_0 + i(\Omega_0 Z_k - \overline{\Omega}_0 Z'_k)$
36:         $X_{k+N/8} \leftarrow U_{k+N/8} * \gamma_{N,k} - T_1 + (\Omega_1 Z_{k+N/8} + \overline{\Omega}_0 Z'_{k+N/8})$
37:         $X_{k+3N/8} \leftarrow U_{k+N/8} * \gamma_{N,k} + T_1 - i(\Omega_1 Z_{k+N/8} - \overline{\Omega}_0 Z'_{k+N/8})$
38:         $X_{k+5N/8} \leftarrow U_{k+N/8} * \gamma_{N,k} - T_1 - (\Omega_1 Z_{k+N/8} + \overline{\Omega}_0 Z'_{k+N/8})$
39:         $X_{k+7N/8} \leftarrow U_{k+N/8} * \gamma_{N,k} + T_1 + i(\Omega_1 Z_{k+N/8} - \overline{\Omega}_0 Z'_{k+N/8})$
40:     end for
41:     return $X_k$
42: end if
```

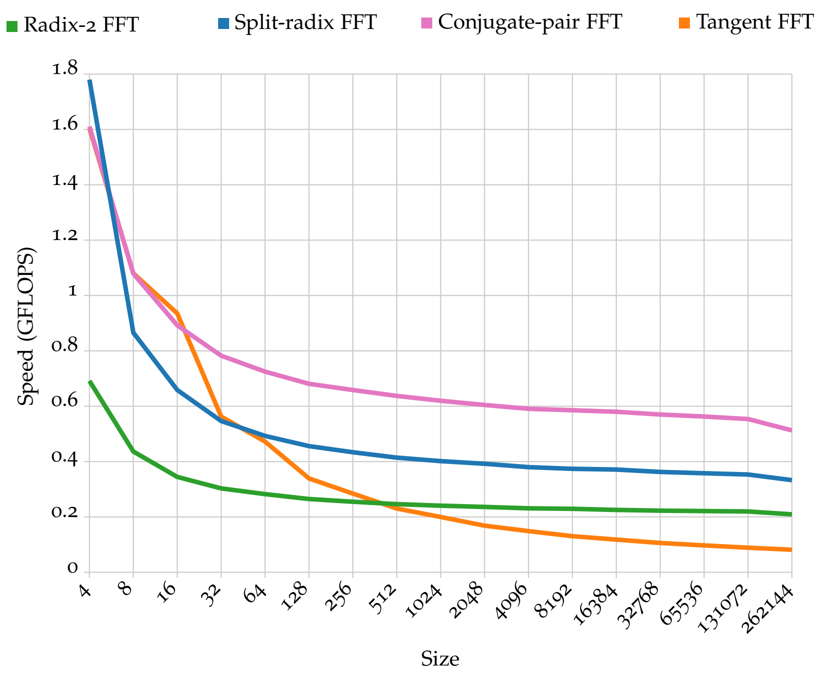


Figure 1: Speed of simple FFT implementations

Algorithm 5 is almost a 1:1 mapping of Equation 28, except that the base cases of $N = 1, 2, 4$ are handled explicitly. In Algorithm 5, the case of $N = 4$ is handled with two size 2 base cases, which are combined into a size 4 FFT.

3.1.5 *Putting it all together*

The simple implementations covered in this section were benchmarked for sizes of transforms 2^2 through to 2^{18} running on a Macbook Air 4,2 and the results are plotted in Figure 1. The speed of each transform is measured in Cooley-Tukey gigaflops (CTGs), where a higher measurement indicates a faster transform.[5]

It can be seen from Figure 1 that although the conjugate-pair and split-radix algorithms have exactly the same floating point operation (FLOP)

[5] CTGs are an inverse time measurement. See Chapter 6 for a full explanation of the benchmarking methods.

count, the conjugate-pair algorithm is substantially faster. The difference in speed can be attributed to the fact that the conjugate-pair algorithm requires only one twiddle factor per size 4 sub-transform, whereas the ordinary split-radix algorithm requires two.

Though the tangent FFT requires the same number of twiddle factors but uses fewer FLOPs compared to the conjugate-pair algorithm, its performance is worse than the radix-2 FFT for most sizes of transform, and this can be attributed to the cost of computing the scaling factors.

A simple analysis with a profiling tool reveals that each implementations' runtime is dominated by the time taken to compute the coefficients. Even in the case of the conjugate-pair algorithm, over 55% of the runtime is spent calculating the complex exponential function. Eliminating this performance bottleneck is the topic of the next section.

3.2 PRECOMPUTED COEFFICIENTS

The speed of Algorithms 1 – 5 may be dramatically improved if the coefficients are precomputed and stored in a lookup table (LUT).

When computing an FFT of size N, Algorithm 1 requires $N/2$ different twiddle factors that correspond to $N/2$ samples of a half rotation around the complex plane. Rather than storing $N/2$ complex numbers, the symmetries of the sine and cosine waves that compose ω_N^k may be exploited to reduce the storage to $N/4$ real numbers – a 75% reduction in memory – by storing only one quadrant of a sine or cosine wave from which the real and imaginary parts of any twiddle factor can be constructed. Such a scheme has advantages in hardware implementations where LUT memory is a costly resource [65], but for modern microprocessor implementations of the FFT, it is more advantageous to have a less complex indexing scheme and better memory locality, rather than a smaller LUT.

As already mentioned, each transform of size N that is computed with Algorithm 1 requires N/2 twiddle factors from ω_N^0 through to $\omega_N^{N/2}$, but the two sub-transforms of Algorithm 1 require twiddle factors ranging from $\omega_{N/2}^0$ through to $\omega_{N/2}^{N/4}$. The twiddle factors of the sub-transforms can be obtained by downsampling the parent transform's twiddle factors by a factor of 2, and because the downsampling factors are all powers of 2, simple shift operations can be used to index any twiddle factor anywhere in the transform from one LUT.

Appendix B contains listings of source code that augment each of the simple implementations from the previous section with LUTs of precomputed coefficients. The modifications are fairly minor: each implementation now has an initialization function that populates the LUT/LUTs based on the size of the transform to be computed, and each transform function now has a parameter of `log2(stride)`, so as to economically index the twiddle factors with little computation.

As Figure 2 shows, the speedup resulting from the precomputed twiddle LUT is dramatic – sometimes more than a factor of 6 (cf. Figure 1). Interestingly, the ordinary split-radix algorithm is now faster than the conjugate-pair algorithm, and inspection of the compiler output shows that this is due to the more complicated addressing scheme at the leaves of the computation, and because the compiler lacks good heuristics for complex multiplication by a conjugate. The performance of the tangent FFT is hampered by the same problem, yet the tangent FFT has better performance, which can be attributed to the tangent FFT having larger straight line blocks of code at the leaves of the computation (the tangent

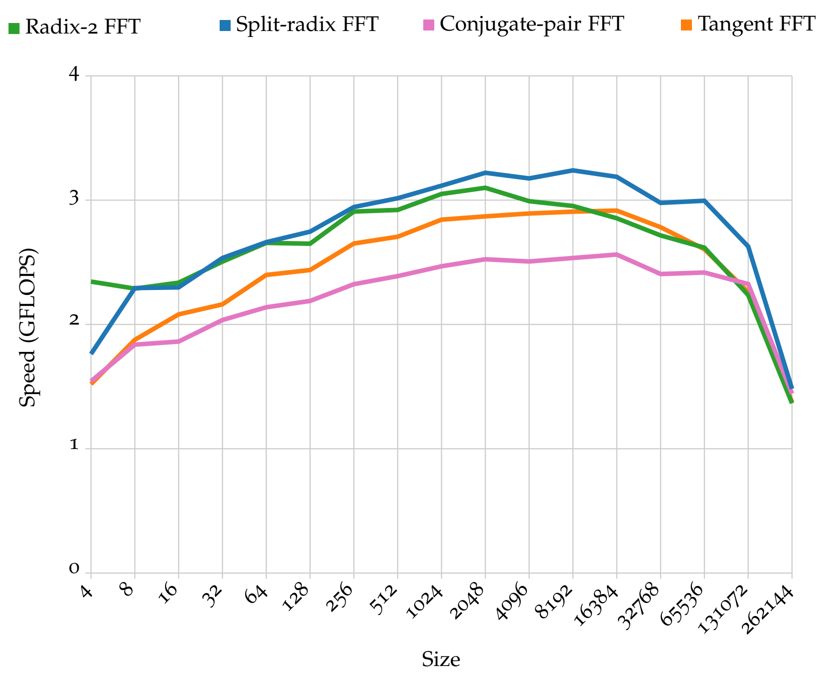


Figure 2: Speed of FFTs with precomputed coefficients

FFT has leaves of size 4, while the split-radix and conjugate-pair FFTs have leaves of size 2).

3.3 SINGLE INSTRUCTION, MULTIPLE DATA

The performance of the programs in the previous section may be further improved by explicitly describing the computation with SIMD intrinsics. Auto-vectorizing compilers, such as the Intel C compiler used to compile the previous examples, can extract some data-level parallelism and generate SIMD code from a scalar description of a computation, but better results can be obtained when using vector intrinsics to explicitly specify the parallel computation.

Intrinsics are an alternative to inline assembly code when the compiler fails to meet performance constraints. In most cases an intrinsic function directly maps to a single instruction on the underlying machine, and so intrinsics provide many of the advantages of inline assembler code. But in contrast to inline assembler code, the compiler uses its detailed knowledge of the intrinsic semantics to provide better optimizations and handle tasks such as register allocation.

Almost all desktop and handheld machines now have processors that implement some sort of SIMD extension to the instruction set. All major Intel processors since the Pentium III have implemented streaming SIMD extensions (SSE), an extension to the x86 architecture that introduced 4-way single precision floating point computation with a new register file consisting of eight 128-bit SIMD registers – known as XMM registers. The AMD64 architecture doubled the number of XMM registers to 16, and Intel followed by implementing 16 XMM registers in the Intel 64 architecture. SSE has since been expanded with support for other data types and new instructions with the introduction of SSE2, SSE3, SSSE3 and SSE4. Most notably, SSE2 introduced support for double precision floating point arithmetic and thus Intel's first attempt at SIMD extensions, multimedia extensions (MMX), was effectively deprecated. Intel's recent introduction of the sandybridge micro-architecture heralded the first implementation of advanced vector extensions (AVX) – a major upgrade to SSE that doubled the size of XMM registers to 256 bits (and renamed them YMM registers), enabling 8-way single precision and 4-way double precision floating point arithmetic.

Another notable example of SIMD extensions implemented in commodity microprocessors is the NEON extension to the ARMv7 architecture. The Cortex family of processors that implement ARMv7 are widely used in mobile, handheld and tablet computing devices such as the iPad, iPhone and Canon PowerShot A470, and the NEON extensions provide these embedded devices with the performance required for processing audio and video codecs as well as graphics and gaming workloads.

Compared to SSE and AVX, NEON has some subtle differences that can greatly improve performance if used properly. First, it has dual length SIMD vectors that are aliased over the same registers; a pair of 64-bit registers refers to the lower and upper half of one 128-bit register – in contrast, the AVX extension increases the size of SSE registers to 256-bit, but the SSE registers are only aliased over the lower half of the AVX registers. Second, NEON can interleave and de-interleave data during vector load or store operations, for up to four vectors of four elements interleaved together. In the context of FFTs, the interleaving/de-interleaving instructions can be used to reduce or eliminate vector permutations or shuffles.

3.3.1 *Split format vs. interleaved format*

In the previous examples, the data was stored in interleaved format (i.e., the real and imaginary parts composing each element of complex data are stored adjacently in memory), but operating on the data in split format (i.e., the real parts of each element are stored in one contiguous array, while the imaginary parts of each element are stored contiguously in another array) can simplify the computation when using SIMD. The case of complex multiplication illustrates this point.

3.3.1.1 *Interleaved format complex multiplication*

The function in Listing 1 takes complex data in two 4-way single precision SSE registers (*a* and *b*) and performs complex multiplication, returning the result in a single precision SSE register. The SSE intrinsic

Listing 1: SSE multiplication with interleaved complex data

```
1  static inline __m128 MUL_INTERLEAVED(__m128 a, __m128 b) {
2    __m128 re, im;
3    re = _mm_shuffle_ps(a,a,_MM_SHUFFLE(2,2,0,0));
4    re = _mm_mul_ps(re, b);
5    im = _mm_shuffle_ps(a,a,_MM_SHUFFLE(3,3,1,1));
6    b = _mm_shuffle_ps(b,b,_MM_SHUFFLE(2,3,0,1));
7    im = _mm_mul_ps(im, b);
8    im = _mm_xor_ps(im, _mm_set_ps(0.0f, -0.0f, 0.0f, -0.0f));
9    return _mm_add_ps(re, im);
10 }
```

functions are prefixed with '`_mm_`', and the SSE data type corresponding to a single 128-bit single precision register is '`__m128`'.

When operating with interleaved data, each SSE register contains two complex numbers. Two shuffle operations at lines 3 and 5 are used to replicate the real and imaginary parts (respectively) of the two complex numbers in input *a*. At line 4, the real and imaginary parts of the two complex numbers in *b* are each multiplied with the real parts of the complex numbers in *a*. A third shuffle is used to swap the real and imaginary parts of the complex numbers in *b*, before being multiplied with the imaginary parts of the complex numbers in *a* – and the exclusive or operation at line 8 is used to selectively negate the sign of the real parts in this result. Finally, the two intermediate results stored in the *re* and *im* registers are added. In total, seven SSE instructions are used to multiply two pairs of single precision complex numbers.

3.3.1.2 *Split format complex multiplication*

The function in Listing 2 takes complex data in two *structs* of SSE registers, performs the complex multiplication of each element of the vectors, and returns the result in a struct of SSE registers. Each struct is composed of a register containing the real parts of four complex numbers, Listing 2: SSE multiplication with split complex data

```c
typedef struct _reg_t {
  __m128 re, im;
} reg_t;

static inline reg_t MUL_SPLIT(reg_t a, reg_t b) {
  reg_t r;
  r.re = _mm_sub_ps(_mm_mul_ps(a.re,b.re),_mm_mul_ps(a.im,b.im));
  r.im = _mm_add_ps(_mm_mul_ps(a.re,b.im),_mm_mul_ps(a.im,b.re));
  return r;
}
```

and another register containing the imaginary parts – so the function in Listing 2 is effectively operating on vectors twice as long as the function in Listing 1. The benefit of operating in split format is obvious: the shuffle operations that were required in Listing 1 are avoided because the real and imaginary parts can be implicitly swapped at the instruction level, rather than by awkwardly manipulating SIMD registers at the data level of abstraction. Thus, Listing 2 computes complex multiplication for vectors twice as long while using one less SSE instruction – not to mention other advantages such as reducing chains of dependent instructions. The only disadvantage to the split format approach is that twice as many registers are needed to compute a given operation – this might preclude the use of a larger radix or force register paging for some kernels of computation.

3.3.1.3 *Fast interleaved format complex multiplication*

Listing 3 is fast method of interleaved complex multiplication that may be used in situations where one of the operands can be unpacked prior to multiplication – in such cases the instruction count is reduced from 7 instructions to 4 instructions (cf. Listing 1). This method of complex multiplication lends itself especially well to the conjugate-pair algorithm

Listing 3: SSE multiplication with partially unpacked interleaved data

```
static inline __m128
MUL_UNPACKED_INTERLEAVED(__m128 re, __m128 im, __m128 b) {
  re = _mm_mul_ps(re, b);
  im = _mm_mul_ps(im, b);
  im = _mm_shuffle_ps(im,im,_MM_SHUFFLE(2,3,0,1));
  return _mm_add_ps(re, im);
}
```

where the same twiddle factor is used twice – by doubling the size of the twiddle factor LUT, the multiplication instruction count is reduced from 14 instructions to 8 instructions. Furthermore, large chains of dependent instructions are reduced, and in practice the actual performance gain can be quite impressive.

Operand *a* in Listing 1 has been replaced with two operands in Listing 3: *re* and *im* – these operands have been unpacked, as was done in lines 3 and 5 of Listing 1. Furthermore, line 8 of Listing 1 is also avoided by performing the selective negation during unpacking.

3.3.2 *Vectorized loops*

The performance of the FFTs in the previous sections can be increased by explicitly vectorizing the loops. The Macbook Air 4,2 used to compile and run the previous examples has a central processing unit that implements SSE and AVX, but for the purposes of simplicity, SSE intrinsics are used in the following examples. The loop of the radix-2 implementation is used as an example in Listing 4.

Each iteration of the loop in Listing 4 accesses two elements of complex data in the array *out*, and one complex element from the twiddle factor LUT. Over multiple iterations of the loop, *out* is accessed contiguously in two places, but the LUT is accessed with a non-unit stride in

Listing 4: Inner loop of radix-2 FFT

```
    for(k=0;k<N/2;k++) {
      data_t Ek = out[k];
      data_t Ok = out[(k+N/2)];
      data_t w = LUT[k<<log2stride];
      out[k]        = Ek + w * Ok;
      out[(k+N/2) ] = Ek - w * Ok;
    }
```

Listing 5: Vectorized inner loop of radix-2 FFT

```
  for(k=0;k<N/2;k+=4) {
    __m128 Ok_re = _mm_load_ps((float *)&out[k+N/2]);
    __m128 Ok_im = _mm_load_ps((float *)&out[k+N/2+2]);
    __m128 w_re = _mm_load_ps((float *)&LUT[log2stride][k]);
    __m128 w_im = _mm_load_ps((float *)&LUT[log2stride][k+2]);
    __m128 Ek_re = _mm_load_ps((float *)&out[k]);
    __m128 Ek_im = _mm_load_ps((float *)&out[k+2]);
    __m128 wOk_re =
      _mm_sub_ps(_mm_mul_ps(Ok_re,w_re),_mm_mul_ps(Ok_im,w_im));
    __m128 wOk_im =
      _mm_add_ps(_mm_mul_ps(Ok_re,w_im),_mm_mul_ps(Ok_im,w_re));
    _mm_store_ps((float *)(out+k), _mm_add_ps(Ek_re, wOk_re));
    _mm_store_ps((float *)(out+k+2), _mm_add_ps(Ek_im, wOk_im));
    _mm_store_ps((float *)(out+k+N/2), _mm_sub_ps(Ek_re, wOk_re));
    _mm_store_ps((float *)(out+k+N/2+2), _mm_sub_ps(Ek_im, wOk_im));
  }
```

all sub-transforms except the outer transform. Some vector machines can perform what are known as vector scatter or gather memory operations – where a vector gather could be used in this case to gather elements from the LUT that are separated by a stride. But SSE only supports contiguous or *streaming* access to memory. Thus, to efficiently compute multiple iterations of the loop in parallel, the twiddle factor LUT is replaced with an array of LUTs – each corresponding to a sub-transform of a particular size. In this way, all memory accesses for the parallelized loop are contiguous and no memory bandwidth is wasted.

Listing 5 computes the loop of Listing 4 using split format data and a vector length of four (i.e., it computes four iterations at once). Note that the vector load and store operations used in Listing 5 require that the memory accesses are 16-byte aligned – this is a fairly standard proviso for vector memory operations, and use of the correct memory alignment attributes and/or memory allocation routines ensures that memory is always correctly aligned.

Some FFT libraries require the input to be in split format (i.e., the real parts of each element are stored in one contiguous array, while the imaginary parts are stored contiguously in another array) for the purposes of simplifying the computation, but this conflicts with many other libraries and use cases of the FFT – for example, Apple's vDSP library operates in split format, but many examples require the use of un-zip/zip functions on the input/output data (see Usage Case 2: Fast Fourier Transforms in [42]). A compromise is to convert interleaved format data to split format on the first pass of the FFT, computing most of the FFT with split format sub-transforms, and converting the data back to interleaved format as it is processed on the last pass.

Appendix C contains listings of FFTs with vectorized loops. The input and output of the FFTs is in interleaved format, but the computation of the inner loops is performed on split format data. At the leaves of the transform there are no loops, so the computation falls back to scalar arithmetic.

Figure 3 summarizes the performance of the listings in Appendix C. Interestingly, the radix-2 FFT is faster than both the conjugate-pair and ordinary split-radix algorithms until size 4096 transforms, and this is due to the conjugate-pair and split-radix algorithms being more complicated at the leaves of the computation. The radix-2 algorithm only has to deal with one size of sub-transform at the leaves, but the split-radix algorithms have to handle special cases for two sizes, and furthermore,

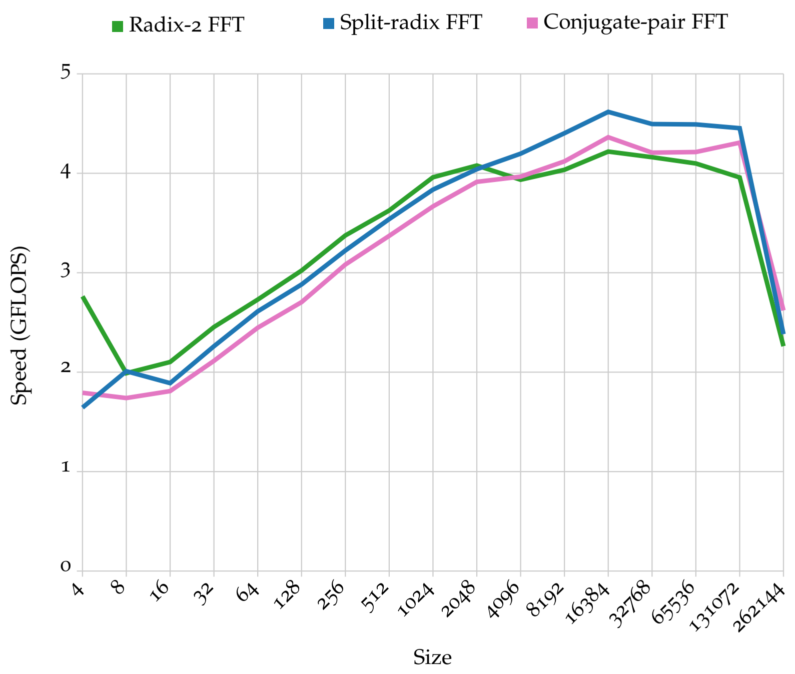


Figure 3: Speed of FFTs with vectorized loops

a larger proportion of the computation takes place at the leaves with the split-radix algorithms. The conjugate-pair algorithm is again slower than the ordinary split-radix algorithm, which can (again) be attributed to the compiler's relatively degenerate code output when computing complex multiplication with a conjugate.

Overall, performance improves with the use of explicit vector parallelism, but still falls short of the state of the art. The next section characterizes the remaining performance bottlenecks.

3.4 THE PERFORMANCE BOTTLENECK

The memory access patterns of an FFT are the biggest obstacle to performance on modern microprocessors. To illustrate this point, Figure 4 visualizes the memory accesses of each straight line block of code in a

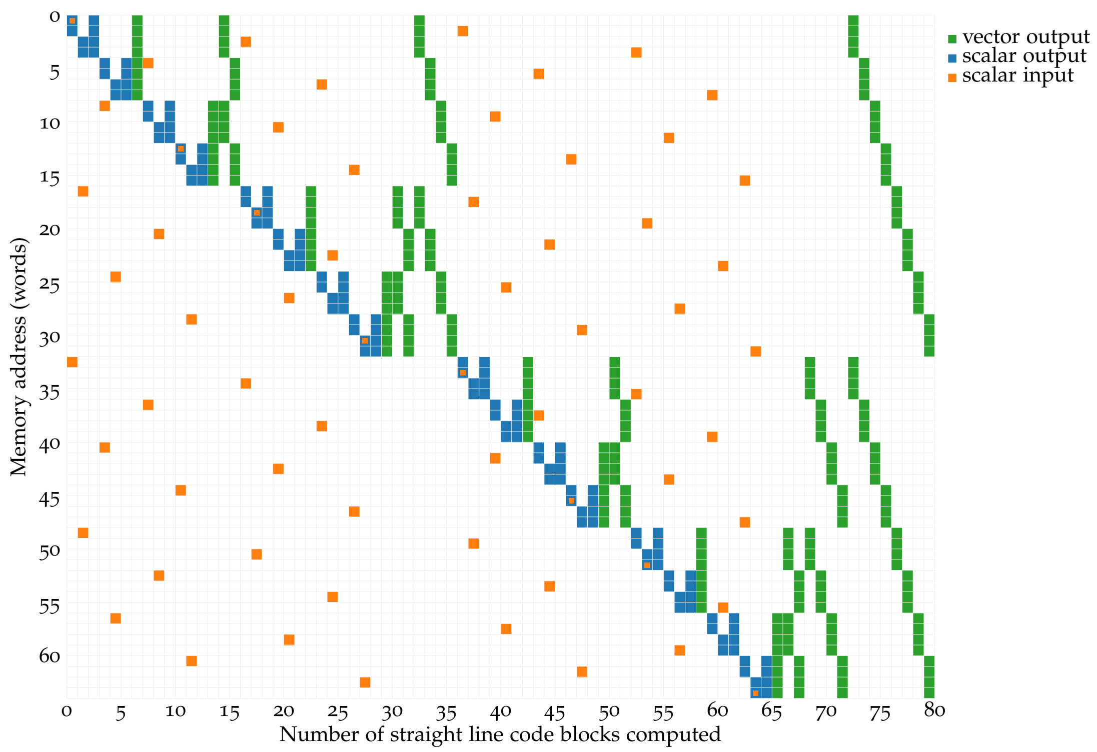


Figure 4: Memory access pattern of straight line blocks of code in a size 64 radix-2 FFT

size 64 radix-2 DIT FFT (the source code of which is provided in Appendix C, Listing 28).

The vertical axis of Figure 4 is memory. Because the diagram depicts a size 64 transform there are 64 rows, each corresponding to a complex word in memory. Because the transform is out-of-place, there are input and output arrays for the data. The input array contains the data "in time", while the output array contains the result "in frequency". Rather than show 128 rows – 64 for the input and 64 for the output – the input array's address space has been aliased over the output array's address space, where the orange code indicates an access to the input array and the green and blue codes for accesses to the output array.

Each column along the horizontal axis represents the memory accesses sampled at each kernel (i.e., butterfly) of the computation, which are all straight line blocks of code. The first column shows two orange and one blue memory operations, and these correspond to a radix-2 computation at the leaves reading two elements from the input data, and writing two elements into the output array. The second column shows a similar radix-2 computation at the leaves: two elements of data are read from the input at addresses 18 and 48, the size 2 DFT computed, and the results written to the output array at addresses 2 and 3.

There are columns that do not indicate accesses to the input array, and these are the blocks that are not at the leaves of the computation. They load data from some locations in the output, performing the computation, and store the data back to the same locations in the output array.

There are two problems that Figure 4 illustrates. The first is that the accesses to the input array – the samples “in time" – are indeed very decimated, as might be expected with a decimation in time algorithm. Second, it can be observed that the leaves of the computation are rather inefficient, because there are large numbers of straight line blocks of code performing scalar memory accesses, and no loops of more than a few iterations (i.e., the leaves of the computation are not taking advantage of the machine's SIMD capability).

Figure 3 in the previous section showed that the vectorized radix-2 FFT was faster than the split-radix algorithms up to size 4096 transforms; a comparison between Figure 4 and Figure 5 helps explain this phenomenon. The split-radix algorithm spends more time computing the leaves of the computation (blue), so despite the split-radix algorithms being more efficient in the inner loops of SIMD computation, the performance has been held back by higher proportion of very small straight line blocks of code (corresponding to sub-transforms smaller than size 4) performing scalar memory accesses at the leaves of the computation.

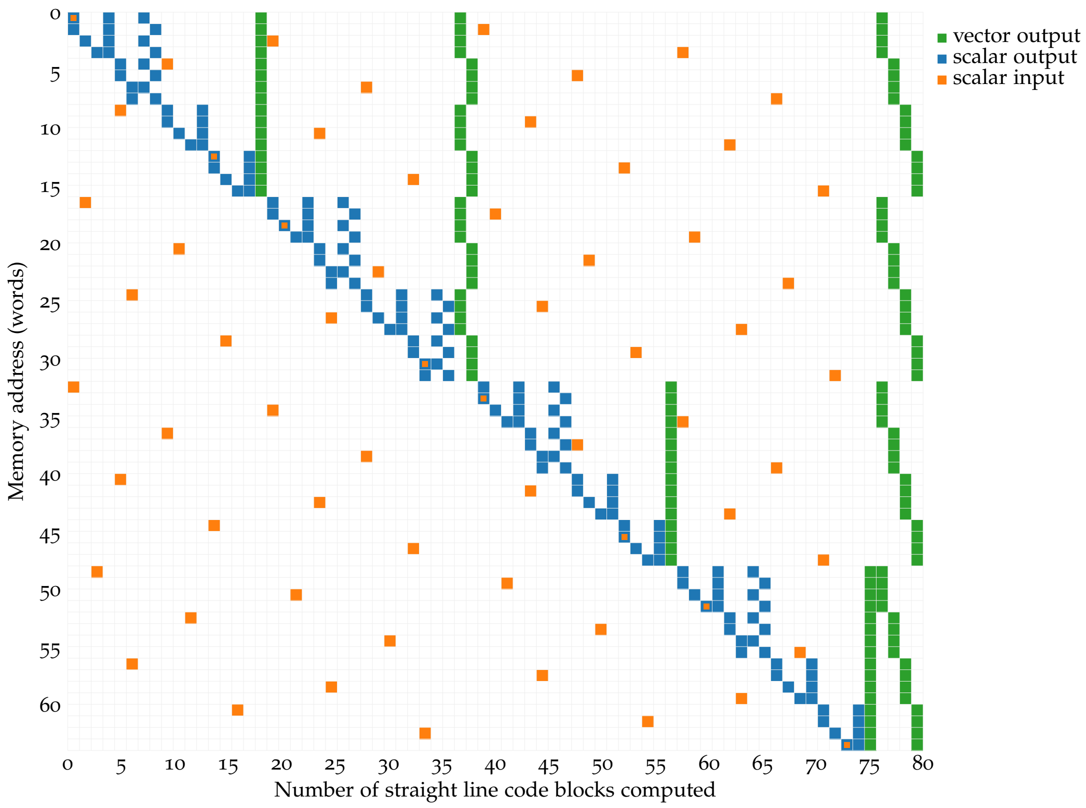


Figure 5: Memory access pattern of straight line blocks of code in a size 64 split-radix FFT

Because the addresses of memory operations at the leaves are a function of variables passed on the stack, it is very difficult for a hardware prefetch unit to keep these leaves supplied with data, and thus memory latency becomes an issue. In later chapters, it is shown that increasing the size of the base cases at the leaves improves performance.

4 EXISTING LIBRARIES

"... My unscheduled code, 86 lines long, did a size-256 single-precision transform in about 35000 Pentium cycles, faster than FFTW. A few days later, after some casual instruction scheduling, I had the time down to about 24000 Pentium cycles."

— Daniel Bernstein on the first release of djbfft [10]

Owing to the importance of efficiently computing FFTs in signal processing and other areas, there have been many implementations for microprocessors; FFTW's benchmark software, for example, includes a collection of 25 different FFT implementations. However, of the many implementations, only a few have competed with the state of the art over the last fifteen years. Since its first release in 1997, FFTW has risen to become one of the most well known fast Fourier transform libraries. Other libraries reviewed in this chapter are SPIRAL, UHFFT, djbfft, Apple vDSP, MatrixFFT, and Intel IPP.

4.1 THE "FASTEST FOURIER TRANSFORM IN THE WEST" (FFTW)

FFTW [34, 35, 47] is an implementation of the DFT that attempts to automatically adapt to the hardware in order to maximise performance, and its development in 1997 was predicated on the idea that it had become too complicated to optimize the performance of the fast Fourier transform for modern microprocessors.

The latest release of FFTW, version 3.3, generates a library of over 150 “codelets” at compile time. The codelets are fragments of machine-independent straight-line code derived from DFT algorithms, including the Cooley-Tukey [16] algorithm and its derivatives the split-radix [22, 78], conjugate-pair [47, 48] and mixed-radix algorithms. Radar [71] and Bluestein [11, 64, 70] algorithms are used for sizes that are prime, and the prime-factor algorithm [64, 74] for sizes that are factored by co-primes. At runtime, a plan for a specific problem, e.g., 1024 point 1D forward double precision out-of-place DFT, is generated by searching the huge space of possible codelet configurations for the best solution.

The codelet generator operates in four phases: creation, simplification, scheduling, and unparsing (code generation). During creation, the codelet generator produces a representation of the computation in the form of a DAG. The DAG is expressed in terms of complex numbers [46], and can be viewed as a linear network [17]. In the simplification stage, algebraic transformations and common subexpression elimination rewriting rules [72] are applied to each node of the DAG, which is then topologically sorted to produce a schedule. In a 2008 paper [46], Johnson and Frigo contend that “the compiler needs help with such long blocks of code", and an earlier paper from 1999 [34] is cited to support the hypothesis that compilers are not capable of efficiently allocating registers and scheduling code for hard-coded blocks of about size 64, which compares an earlier version of FFTW compiled with an older compiler[1] to an FFT from Sun’s Performance Library. There is no mention of re-testing the aforementioned hypothesis with more advanced compilers.

1 Sun WorkShop Compilers 4.2 30 Oct 1996 C 4.2

FFTW has several modes available for searching the configuration space of codelets. In "patient" mode, FFTW uses dynamic programming to evaluate the runtime of almost all combinations of possible plans. As the runtime of many sub-problems is repeatedly evaluated while searching the configuration space, the results of locally optimized sub-problems are cached, reducing runtime of the planner while producing results very close to that of an exhaustive search.

In "estimate" mode, FFTW minimizes a heuristic cost that is a function of a particular configuration's count of floating point operations and extraneous memory operations (for buffering and transposes). Compared to patient mode, the runtime of the planner is reduced by several orders of magnitude, at the expense of runtime performance while executing the plan. For executing plans of 1D complex transforms on a PowerPC G5, the median and peak difference in runtime performance between patient and estimate modes was 20% and 72%, respectively. This result is used by Frigo and Johnson to support the hypothesis that there is no longer any correlation between operation counts and runtime performance on modern machines [35].

Frigo and Johnson discuss a small number of planner solutions in their 2005 paper on the design of FFTW3 [35], and conclude that "we do not really understand the planner's choices because we cannot predict what plans will be produced. Indeed, this is the whole point of implementing a planner." They do not mention the use of more rigorous methods, such as machine learning, for the purposes of predicting performance.

FFTW supports computation of complex DFTs with SIMD extensions by means of two-way parallel computation of real DFTs [46]. The Vienna MAP vectorizer [31, 51, 52] has also been coupled with FFTW to produce a high-performance FFT library for the IBM Blue Gene/L supercomputer [62] that is up to 80% faster than the best-performing scalar FFT codes generated by FFTW [54].

4.2 DANIEL BERNSTEIN'S FFT (DJBFFT)

In 1997, Daniel Bernstein noticed that it was not difficult to write code that out-performed FFTW [10]. He had written 86 lines of unscheduled code that computed a size 256 single precision transform in about 35000 Pentium cycles – faster than FFTW. After spending a few more days doing "some casual instruction scheduling," he could compute the same transform with about 24000 Pentium cycles (ibid.).

These performance results directly contradicted the assumption that predicated FFTW: that it was too hard to predict the performance of FFT code on modern microprocessors. Development of djbfft continued until 1999, and it had succeeded in becoming the fastest library for computing FFTs on most Pentium and SPARC machines.

Bernstein's FFT is notable for having been the first publicly available library to exploit the advantages of the conjugate-pair or "-1 exponent" algorithm. After Bernstein demonstrated the advantages of the algorithm in djbfft, Frigo and Johnson followed with an implementation in FFTW [47].

4.3 SPIRAL

SPIRAL [32, 66, 67] attempts to automatically optimize code for signal processing functions such as the discrete Fourier transform. SPIRAL's goal is to automatically optimize signal processing functions at the push of a button, with results that are as good as hand-optimized codes.

In contrast to FFTW, SPIRAL's optimization is performed at compile time, and thus the generated code is less portable. Another point of difference is in the search methods: while FFTW uses dynamic programming, SPIRAL uses a wide range of techniques that include machine learning [67, 66].

Franchetti and Puschel argue that vectorization is best performed at the algorithm level of abstraction by manipulating Kronecker product expressions through mathematical identities [29], and this is the basis for a rewriting system [33] that vectorizes for short vector machines [28, 30, 51].

In [33], SPIRAL is slower than FFTW 3.1 for 2-way double-precision power of two transforms, but SPIRAL is fastest for 4-way single-precision power of two transforms where $16 \leqslant n \leqslant 128$. SPIRAL generates code that is characterized by large basic blocks and single-threaded performance does not scale beyond sizes of about 4096 points. Indeed, source code is only publicly available for sizes 2 through to 8192 points [2].

4.4 UHFFT

UHFFT [4, 6, 3, 61, 60], like FFTW, generates a library of codelets which are assembled into transforms by a planner. The planner uses dynamic programming to search an exponential space of possible algorithms, factors and schedules, relying on codelet timings to predict transform execution times [3].

UHFFT uses the mixed-radix and split-radix [22, 78] algorithms for power of two sizes, the prime-factor algorithm [64, 74] for sizes that are factored by co-primes, and the Radar [71] algorithm for sizes that are prime.

UHFFT generates a schedule from a DAG which has been topologically sorted, mainly to optimize memory reuse distance [3]. The schedule is then unparsed to C code.

Scalar results on Itanium2 and Opteron show that UHFFT's dynamic programming approach can choose a plan having performance within 10% of the actual optimal plan. For power of two sizes, UHFFT's performance was typically worse than FFTW or Intel MKL, while UHFFT was faster than FFTW for prime-factor and prime sizes (ibid.).

4.5 INTEL INTEGRATED PERFORMANCE PRIMITIVES (IPP)

Of the closed source FFT implementations, the integrated performance primitives (IPP) library [44] provides the best results for most sizes of DFT on machines with Intel processors. IPP includes a number of different FFT implementations that appear to be hand optimized for different machine configurations, and in contrast to FFTW, IPP deterministically chooses the best code to run based on the capabilities of the machine and the operating system (OS) – achieving results that are typically superior to FFTW.

Because IPP is closed source, there is no publicly available information regarding the algorithms and techniques used.

4.6 APPLE VDSP

The Apple Accelerate libraries contain a wide range of computationally intensive functions that have been optimized for vector computation on PowerPC, x86 and ARM architectures. Within the Accelerate library, vDSP is a collection of digital signal processor (DSP) functions that includes the FFT.

The vDSP implementation of the FFT is distinctive among the other libraries reviewed in this chapter in that it only operates on data that is stored in split format (where the real and imaginary parts of complex numbers are stored in separate arrays). However, many applications have data that is already in interleaved format (where the real and imaginary part of each complex number are stored adjacent in memory), or require data in interleaved format, and so vDSP provides un-zip/zip functions for converting data to/from split format.

The Apple vDSP library is notable for having very good FFT performance on ARM NEON devices, while its x86 performance is average (comparable with FFTW "estimate" mode performance).

As with IPP, vDSP is only distributed in binary form and thus little can be said about the algorithms and techniques employed.

4.7 MATRIXFFT

MatrixFFT is a library for efficiently computing large transforms of more than 2^{18} points on Apple hardware, with sustained processing rates reportedly being as high as 40 CTGs for very large single precision transforms. Large scale FFTs have been used in areas such as image processing (with images of over 10^9 pixels) and experimental mathematics (for extreme-precision computation of π).

MatrixFFT uses the four-step algorithm to decompose a transform into smaller sub-transforms that fit in the cache [8], and computes the smaller sub-transforms with Apple vDSP. Interestingly, MatrixFFT has better performance – in many cases – while using interleaved format to store the data, even though the interleaved format must be converted to split format before using vDSP [69].

MatrixFFT includes a calibration utility that evaluates the various implementation parameters for each size of transform on a given machine, which can then be used to re-compile the library so that it achieves best performance on that particular machine.

MatrixFFT is freely available and distributed in source code form by Apple [43].

Part II

FASTER FOURIER TRANSFORMS

5

SFFT: A HIGH-PERFORMANCE FFT LIBRARY

"It's time to nut up, or shut up."

— Woody Harrelson in *Zombieland* (2009 film)

This chapter describes SFFT: a high-performance FFT library for SIMD microprocessors that is, in many cases, faster than the state of the art FFT libraries reviewed in Chapter 4.

Chapter 3 described some simple implementations of the FFT and concluded with an analysis of the performance bottlenecks. The implementations presented in this chapter are designed to improve spatial locality, and utilize larger straight line blocks of code at the leaves, corresponding to sub-transforms of sizes 8 through to 64, in order to reduce latency and stack overheads.

In distinct contrast to the simple FFT programs of Chapter 3, this chapter employs meta-programming. Rather than describe FFT programs, we describe programs that *statically elaborate* the FFT into a DAG of nodes representing the computation, apply some optimizing transformations to the graph, and then generate code. Many other auto-vectorization techniques, such as those employed by SPIRAL, operate at the instruction level [54], but the techniques presented in this chapter vectorize blocks of computation at the algorithm level of abstraction, thus enabling some of the algorithms structure to be utilized.

Three types of implementation are described in this chapter, and the performance of each depends on the parameters of the transform to be computed and the characteristics of the underlying machine. For a given machine and FFT to be computed (which has parameters such as length and precision), the fastest configuration is selected from among a small set of up to eight possible FFT configurations – a much smaller space compared to FFTW's exhaustive search of all possible FFTs. The fastest configuration is easily selected by timing each of the possible options, but it is shown in Chapter 7 that it is also possible to use machine learning to build a classifier that will predict the fastest based on attributes such as the size of the cache.

SFFT comprises three types of conjugate-pair implementation, which are:

1. Fully hard-coded FFTs;
2. Four-step FFTs with hard-coded sub-transforms;
3. FFTs with hard-coded leaves.

5.1 FULLY HARD-CODED

Statically elaborating a DAG that represents a depth-first recursive FFT is much like computing a depth-first recursive FFT: instead of performing computation at the leaves of the recursion and where smaller DFTs are combined into one, a node representing the computation is appended to the end of a list, and the list of nodes, i.e., a topological ordering of the DAG, is later translated into a program that can be compiled and executed.

Emitting code with a vector length of 1 (i.e., scalar code or vector code where only one complex element fits in a vector register) is relatively simple and is described in Section 5.1.1. For vector lengths above 1, vectorizing the topological ordering of nodes poses some subtle challenges, and these details are described in Section 5.1.2. The fully hard-coded FFTs described in this section are generally only practical for smaller sizes of transforms, typically where $N \leqslant 128$, however these techniques are expanded in later sections to scale the performance to larger sizes.

5.1.1 *Vector length 1*

A vector length (VL) of 1 implies that the computation is essentially scalar, and only one complex element can fit in a vector register. An example of such a scenario is when using interleaved double-precision floating-point arithmetic on an SSE2 machine: one 128-bit XMM register is used to store two 64-bit floats that represent the real and imaginary parts of a complex number.

When $VL = 1$, the process of generating a program for a hard-coded FFT is as follows:

1. Elaborate a topological ordering of nodes, where each node represents either a computation at the leaves of the transform, or a computation in the body of the transform (i.e., where smaller subtransforms are combined into a larger transform);

2. Write the program header to output, including a list of variables that correspond to registers used by the nodes;

3. Traverse the list of nodes in order, and for each node, emit a statement that performs the computation represented by the given node. If a node is the last node to use a variable, a statement storing the variable to its corresponding location in memory is also emitted;

4. Write the program footer to output.

Listing 6: Elaborate function for hard-coded conjugate-pair FFT

```
 1 void
 2 CSplitRadix::elaborate(int N, int ioffset, int offset, int stride) {
 3   if(N > 4) {
 4     elaborate(N/2, ioffset, offset, stride+1);
 5     if(N/4 >= 4) {
 6       elaborate(N/4, ioffset+(1<<stride), offset+(N/2), stride+2);
 7       elaborate(N/4, ioffset-(1<<stride), offset+(3*N/4), stride+2);
 8     }else{
 9       CNodeLoad *n = new CNodeLoad(this, 4, ioffset, stride, 0);
10       ns.push_back(assign_leaf_registers(n));
11     }
12     for(int k=0;k<N/4;k++) {
13       CNodeBfly *n = new CNodeBfly(this, 4, k, stride);
14       ns.push_back(assign_body_registers(n,k,N);
15     }
16   }else if(N==4) {
17     CNodeLoad *n = new CNodeLoad(this, 4, ioffset, stride, 1);
18     ns.push_back(assign_leaf_registers(n));
19   }else if(N==2) {
20     CNodeLoad *n = new CNodeLoad(this, 2, ioffset, stride, 1);
21     ns.push_back(assign_leaf_registers(n));
22   }
23 }
```

5.1.1.1 *Elaborate*

Listing 6 is a function, written in C++, that performs the first task in the process. As mentioned earlier, elaborating a topological ordering of nodes with a depth-first recursive structure is much like actually computing an FFT with a depth-first recursive program (cf. Listing 26 in Appendix B). Table 2 lists the nodes contained in the list ‘`ns`’ after elaborating a size-8 transform by invoking `elaborate(8, 0, 0, 0)`.

A transform is divided into sub-transforms with recursive calls at lines 4, 6 and 7, until the base cases of size 2 or size 4 are reached at the leaves of the elaboration. As well as the size-2 and size-4 base cases, which are handled at lines 20-21 and 17-18 (respectively), there is a special case where two size-2 base cases are handled in parallel at lines 9-10. This special case of handling two size-2 base cases as a larger size-4 node

TYPE	SIZE	ADDRESSES	REGISTERS	TWIDDLE
`CNodeLoad`	4	{0,4,2,6}	{0,1,2,3}	
`CNodeLoad`	2(x2)	{1,5,7,3}	{4,5,6,7}	
`CNodeBfly`	4		{0,2,4,6}	ω_8^0
`CNodeBfly`	4		{1,3,5,7}	ω_8^1

Table 2: VL-1 size-8 conjugate-pair transform nodes

ensures that larger transforms are composed of nodes that are homogeneous in size – this is of little utility when emitting $\text{VL} = 1$ code, but it is exploited in Section 5.1.2 where the topological ordering of nodes is vectorized. The second row of Table 2 is just such a special case, since two size-2 leaf nodes are being computed, and thus the size is listed as 2(x2).

The `elaborate` function modifies the class member variable '`ns`' at lines 10, 14, 18 and 21, where it appends a new node to the back of the list. After the function returns, the `ns` list represents a topological ordering of the computation with `CNodeLoad` and `CNodeBfly` nodes. The nodes of type `CNodeLoad` represent leaf computations: these computations load elements from the input array and perform a small amount of leaf computation, leaving the result in a set of registers. The `CNodeBfly` nodes represent computations in the body of the transform: these use a twiddle factor to perform a butterfly computation on a vector of registers, leaving the result in the same registers.

The constructor for a `CNodeLoad` object computes input array addresses for the load operations using the input array offset (`ioffset`), the input array `stride`, the size of the node (the nodes instantiated at lines 9 and 17 are size-4, and the node instantiated at line 20 is size-2) and a final parameter that is non-zero if the node is a single node (the nodes instantiated at lines 17 and 20 are single nodes, while the node instantiated at line 9 is composed of two size-2 nodes).

As the newly instantiated `CNodeLoad` objects are appended to the back of `ns` at lines 10, 14 and 21, the `assign_leaf_registers` function assigns registers to the outputs of each instance. Registers are identified with integers beginning at zero, and when each register is created it is assigned an identifier from an auto-incrementing counter (R_{counter}). This function also maintains a map of registers to node pointers, referred to as `rmap`, where the node for a given register is the last node to reference that register.

The constructor for a `CNodeBfly` object uses `k` and `stride` to compute a twiddle factor for the new instance of a butterfly computation node. When the new instance of `CNodeBfly` is appended to the end of `ns` at line 14, the `assign_body_registers` function assigns registers R_i to a node of size N_{node} with the following logic:

$$R_i = R_{\mathrm{counter}} - N + k + i \times \frac{N}{4} \tag{34}$$

where $i = 0, \ldots, N_{\mathrm{node}} - 1$ and R_{counter} is the auto-incrementing register counter. The `assign_body_registers` functions also updates the map of registers to node pointers by setting `rmap`$[R_i]$ to point to the new instance of `CNodeBfly`.

5.1.1.2 *Emitting code*

Given a list of nodes, it is a simple process to emit C code that can be compiled to actually compute the transform.

The example in Listing 7 would be emitted from the list of four nodes in Table 2. Lines 1–4 are emitted from a function that generates a header, Listing 7: Hard-coded VL-1 size-8 FFT

```
 1 void sfft_dcf8_hc(sfft_plan_t *p, const void *vin, void *vout) {
 2   const SFFT_D *in = vin;
 3   SFFT_D *out = vout;
 4   SFFT_R r0,r1,r2,r3,r4,r5,r6,r7;
 5 
 6   L_4(in+0,in+8,in+4,in+12,&r0,&r1,&r2,&r3);
 7   L_2(in+2,in+10,in+14,in+6,&r4,&r5,&r6,&r7);
 8   K_0(&r0,&r2,&r4,&r6);
 9   S_4(r0,r2,r4,r6,out+0,out+4,out+8,out+12);
10   K_N(VLIT2(0.7071,0.7071),VLIT2(0.7071,-0.7071),&r1,&r3,&r5,&r7);
11   S_4(r1,r3,r5,r7,out+2,out+6,out+10,out+14);
12 
13 }
```

and line 13 is emitted from a function that generates a footer. Lines 6–11 are generated based on the list of nodes.

Listing 7 contains references to several types, functions and macros that use upper-case identifiers – these are *primitive* functions or types that have been predefined as inline functions or macros. A benefit of using primitives in this way is that the details specific to numerical representation and the underlying machine have been abstracted away; thus, the same function can be compiled for a variety of types and machines by simply including a different header file with different primitives. Listing 7, for example, could be compiled for double-precision arithmetic on an SSE2 machine by including `sse_double.h`, or it could be compiled with much slower scalar arithmetic by including `scalar.h`. The same code can even be used, without modification, to compute forward and backwards transforms, by using C preprocessor directives to conditionally alter the macros.

In order to accommodate mixed numerical representations, the signature of the outermost function references data with void pointers. In the case of the double-precision example in Listing 7, `SFFT_D` would be defined to be `double` in the appropriate header file, and the void pointers are then cast to `SFFT_D` pointers.

The size-8 transform in Table 2 uses 8 registers, and thus a declaration of 8 registers of type `SFFT_R` has been emitted at line 4 in Listing 7. In the case of double-precision arithmetic on a SSE2 machine, `SFFT_R` is defined as `__m128d` in `sse_double.h`.

The first two rows of Table 2 correspond to lines 6 and 7 of Listing 7, respectively. The `L_4` primitive is used to compute the size-4 leaf node in the first row of the table. The second row is a load/leaf node of size 2(x2), indicating two size-2 nodes in parallel, which is computed with the `L_2` primitive. The input addresses in the table are the addresses of complex words, while the addresses in the generated code refer to the real and imaginary parts of a complex word, and thus the addresses from Table 2 are multiplied by a factor of 2 to obtain the addresses in Listing 7.

The final two `CNodeBfly` nodes of Table 2 correspond to the `K_0` and `K_N` sub-transform (a.k.a. butterfly) primitives at lines 8 and 10, respectively. Because the node in the third row of Table 2 has a twiddle factor of ω_8^0 (i.e., unity), the computation requires no multiplication, and the `K_0` primitive is used for this special case. The `K_N` primitive at line 10 does require a twiddle factor, which is passed to `K_N` as two vector literals that represent the twiddle factor in unpacked form.[1] Section 3.3.1.3 describes how interleaved complex multiplication is faster if one operand is pre-unpacked.

After each node is processed, the registers that have been used by it are checked in a map (`rmap`) that maps each register to the last node to have used that register. If the current node is the last node to have used

1 For the purposes of brevity, the precision has been truncated to only a few decimal places.

a register, the register is stored to memory. In the case of the transform in Listing 7, four registers are stored with an instance of the `S_4` primitive at lines 9 and 11. In contrast to the load operations at the leaves, which are decimated-in-time and thus effectively pseudo-random memory accesses, the store operations are to linear regions of memory, the addresses of which can be determined from each register's integer identifier. The store address offset for data in register R_i is simply $i \times 2 \times \text{VL}$.

5.1.2 *Other vector lengths*

If VL > 1, the list of nodes that results from the `elaborate` function in Listing 6 is vectorized. Broadly speaking, `CNodeLoad` objects that operate on adjacent memory locations are collected together and computed in parallel. After each such computation, each position in a vector register contains an element that belongs to a different node. Transposes are then used to transform sets of vector registers such that each register contains elements from one node. Finally, the `CNodeBfly` objects can be easily computed in parallel, as they were with VL-1 because the elements in each vector register correspond to one node.

5.1.2.1 *Overview*

Table 3 lists the nodes that represent a VL-1 size-16 transform. A VL of 2 implies that each vector register contains 2 complex words, and load operations on each of the 4 addresses in the first row of Table 3 will also load the complex words in the adjacent memory locations. Note that the complex words that would be incidentally loaded in the upper half of the VL-2 registers are the complex words that the third `CNodeLoad` object

TYPE	SIZE	ADDRESSES	REGISTERS	TWIDDLE
CNodeLoad	4	{0,8,4,12}	{0,1,2,3}	
CNodeLoad	2(x2)	{2,10,14,6}	{4,5,6,7}	
CNodeBfly	4		{0,2,4,6}	ω_{16}^{0}
CNodeBfly	4		{1,3,5,7}	ω_{16}^{2}
CNodeLoad	4	{1,9,5,13}	{8,9,10,11}	
CNodeLoad	4	{15,7,3,11}	{12,13,14,15}	
CNodeBfly	4		{0,4,8,12}	ω_{16}^{0}
CNodeBfly	4		{1,5,9,13}	ω_{16}^{1}
CNodeBfly	4		{2,6,10,14}	ω_{16}^{2}
CNodeBfly	4		{3,7,11,15}	ω_{16}^{3}

Table 3: VL-1 size-16 conjugate-pair transform nodes

TYPE	SIZES	ADDRESSES	REGISTERS	TWIDDLES
Load	{4,4}	{{0,1},{8,9},{4,5},{12,13}}	{{0,1},{2,3},{8,9},{10,11}}	
Load	{2(x2),4}	{{2,3},{10,11},{14,15},{6,7}}	{{4,5},{6,7},{14,15},{12,13}}	
Bfly	{4,4}		{{0,1},{2,3},{4,5},{6,7}}	$\{\omega_{16}^{0},\omega_{16}^{2}\}$
Bfly	{4,4}		{{0,1},{4,5},{8,9},{12,13}}	$\{\omega_{16}^{0},\omega_{16}^{1}\}$
Bfly	{4,4}		{{2,3},{6,7},{10,11},{14,15}}	$\{\omega_{16}^{2},\omega_{16}^{3}\}$

Table 4: VL-2 size-16 conjugate-pair transform nodes

at row 5 would have loaded. This is exploited to load and compute the first and third CNodeLoad objects in parallel.

The second CNodeLoad object computes two size-2 leaf transforms in parallel, while the last CNodeLoad object computes a size-4 leaf transform. Because the size-4 transform is composed of two size-2 transforms, and memory addresses of the fourth CNodeLoad are adjacent (although permuted), some of the computation can be computed in parallel.

If the CNodeLoad objects at rows 1 and 5 are computed in parallel, the output will be four VL-2 registers: {{0,8}, {1,9}, {2,10}, {3,11}} – i.e., the first register contains what would have been register 0 in the lower half, and what would have been register 8 in the top half etc. Similarly, computing rows 2 and 6 in parallel would yield four VL-2 registers: {{4,14}, {5,15}, {6,12}, {7,13}} – note the permutation of the upper halves in this case. These registers are transposed to {{0,1}, {2,3}, {8,9}, {10,11}} and {{4,5}, {6,7}, {14,15}, {12,13}}, as in row 1 and 2 of Table 4.

With the transposed VL-2 registers, it is now possible to compute `CNodeBfly` nodes in parallel. For example, rows 2 and 3 of Table 3 can be computed in parallel on four VL-2 registers represented by {{0,1}, {2,3}, {4,5}, {6,7}}, as in row 3 of Table 4.

5.1.2.2 *Implementation*

Listing 9 is a C++ implementation of the `vectorize_loads` function. This function modifies a topological ordering of nodes (the class member variable `ns`) and uses two other functions: `find_parallel_loads`, which searches forward from the current node to find another `CNodeLoad` that shares adjacent memory addresses; and `merge_loads(a,b)`, which adds the addresses, registers and type of `b` to `a`. Type introspection is used at lines 7 and 36 (and in other Listings), to differentiate between the two types of object.

Listing 8 is a C++ implementation of the `vectorize_ks` function. For each `CNodeBfly` node, the function searches forward for another `CNodeBfly` that does not have a register dependence. Once found, the registers of the latter node are added to the former node, and the latter node erased. Finally, at line 19, the registers of the vectorized `CNodeBfly` node are merged using a perfect shuffle, which is then recursively applied on each half of the list. The effect is a merge that works for any power of 2 vector length.

If `vectorize_loads` and `vectorize_ks` are invoked with $\text{VL} = 2$ on the topological ordering of nodes in Table 3, the result is the vectorized node list shown in Table 4. As in Section 5.1.1.2, emitting code is a fairly

Listing 8: Body node vectorization

```
void CSplitRadix::vectorize_ks() {
  vector<CNodeHardCoded *>::iterator i;
  for(i=ns.begin(); i != ns.end();++i) {
    if(!(*i)->type().compare(‘‘blockbfly’’)) {
      vector<CNodeHardCoded *>::iterator j = i+1, pj = i;
      int count = 1;
      while(j != ns.end() && count < VL) {
        if(!(*j)->type().compare(‘‘blockbfly’’)
            && !register_dependence(*i, *j)) {
          (*i)->rs.insert(
            (*i)->rs.end(), (*j)->rs.begin(), (*j)->rs.end());
          ns.erase(j);
          count++;
          j = pj+1;
        }else {
          pj = j; ++j;
        }
      }
      (*i)->merge_rs();
    }
  }
}
```

simple process, and Listing 10 is the code emitted from the node list in Table 4. There are only a few differences to note about the emitted code when VL > 1.

1. The register identifiers in line 4 of Listing 10 consist of a list of two integers delimited with an underscore. The integers listed in each register's name are the VL-1 registers that were subsumed to create the larger register (cf. VL-1 code in Listing 7);

2. The leaf primitives (lines 6 and 7 in Listing 10) have a list of underscore delimited integers in the name, where each integer corresponds to the type of sub-transform to be computed on that position in the vector registers. For example, the `L_4_4` primitive is named to indicate a size-4 leaf operation on the lower and upper halves of the vector registers, while the `L_2_4` primitive performs

Listing 9: Leaf node vectorization

```
CNodeLoad *
CSplitRadix::find_parallel_load(vector<CNodeHardCoded*>::iterator i){
  CNodeLoad *b = (CNodeLoad *)(*i);
  for(int k=0;k<((N>2)?4:2);k++) {
    vector<CNodeHardCoded *>::iterator j = i+1;
    while(j != ns.end()) {
      if(!(*j)->type().compare(‘‘blockload’’)) {
        CNodeLoad *b2 = (CNodeLoad *)(*j);
        if(b2->iaddrs[k] > b->iaddrs[0] &&
            b2->iaddrs[k] < b->iaddrs[0]+VL) {
          ns.erase(j);
          return b2;
        }
        ++j;
      }
    }
  }
  return NULL;
}
void CSplitRadix::merge_loads(CNodeLoad *b1, CNodeLoad *b2) {
  for(int i=0;i<b1->size;i++) {
    for(int j=0;j<b2->iaddrs.size();j++) {
      if(b2->iaddrs[j] > b1->iaddrs[i] &&
          b2->iaddrs[j] < b1->iaddrs[i]+VL) {
        b1->iaddrs.push_back(b2->iaddrs[j]);
        b1->rs.push_back(b2->rs[j]);
        if(rmap[b2->rs[j]] == b2) rmap[b2->rs[j]] = b1;
      }
    }
  }
  b1->types.push_back(b2->types[0]);
}
void CSplitRadix::vectorize_loads() {
  vector<CNodeHardCoded *>::iterator i;
  for(i=ns.begin(); i != ns.end();++i) {
    if(!(*i)->type().compare(‘‘blockload’’)) {
      while(CNodeLoad *b2 = find_parallel_load(i))
        merge_loads((CNodeLoad *)(*i), b2);
    }
  }
}
```

Listing 10: Hard-coded VL-2 size-16 FFT

```
void sfft_fcf16_hc(sfft_plan_t *p, const void *vin, void *vout) {
  const SFFT_D *in = vin;
  SFFT_D *out = vout;
  SFFT_R r0_1,r2_3,r4_5,r6_7,r8_9,r10_11,r12_13,r14_15;

  L_4_4(in+0,in+16,in+8,in+24,&r0_1,&r2_3,&r8_9,&r10_11);
  L_2_4(in+4,in+20,in+28,in+12,&r4_5,&r6_7,&r14_15,&r12_13);
  K_N(VLIT4(0.7071,0.7071,1,1),
      VLIT4(0.7071,-0.7071,0,-0),
      &r0_1,&r2_3,&r4_5,&r6_7);
  K_N(VLIT4(0.9239,0.9239,1,1),
      VLIT4(0.3827,-0.3827,0,-0),
      &r0_1,&r4_5,&r8_9,&r12_13);
  S_4(r0_1,r4_5,r8_9,r12_13,out+0,out+8,out+16,out+24);
  K_N(VLIT4(0.3827,0.3827,0.7071,0.7071),
      VLIT4(0.9239,-0.9239,0.7071,-0.7071),
      &r2_3,&r6_7,&r10_11,&r14_15);
  S_4(r2_3,r6_7,r10_11,r14_15,out+4,out+12,out+20,out+28);
}
```

two size-2 leaf operations on the lower half of the registers and a size-4 leaf operation on the upper halves;

3. The body node primitives (K_N) and store primitives (S_4) are unchanged because they perform the same operation on each element of the vector registers. This is as a result of the register transposes that were previously performed on the outputs of the leaf primitives.

5.1.2.3 *Scalability*

So far, hard-coded transforms of vector length 1 and 2 have been presented. On Intel machines, VL-1 can be used to compute double-precision transforms with SSE2, while VL-2 can be used to compute double-precision transforms with AVX *and* single-precision transforms with SSE. The method of vectorization presented in this chapter scales above VL-2, and has been successfully used to compute VL-4 single-precision transforms with AVX.

The leaf primitives were coded by hand in all cases; VL-1 required `L_2` and `L_4`, while VL-2 required `L_2_2`, `L_2_4`, `L_4_2` and `L_4_4`. In the case of VL-4, not all permutations of possible leaf primitive were required – only 11 out of 16 were needed for the transforms that were generated.

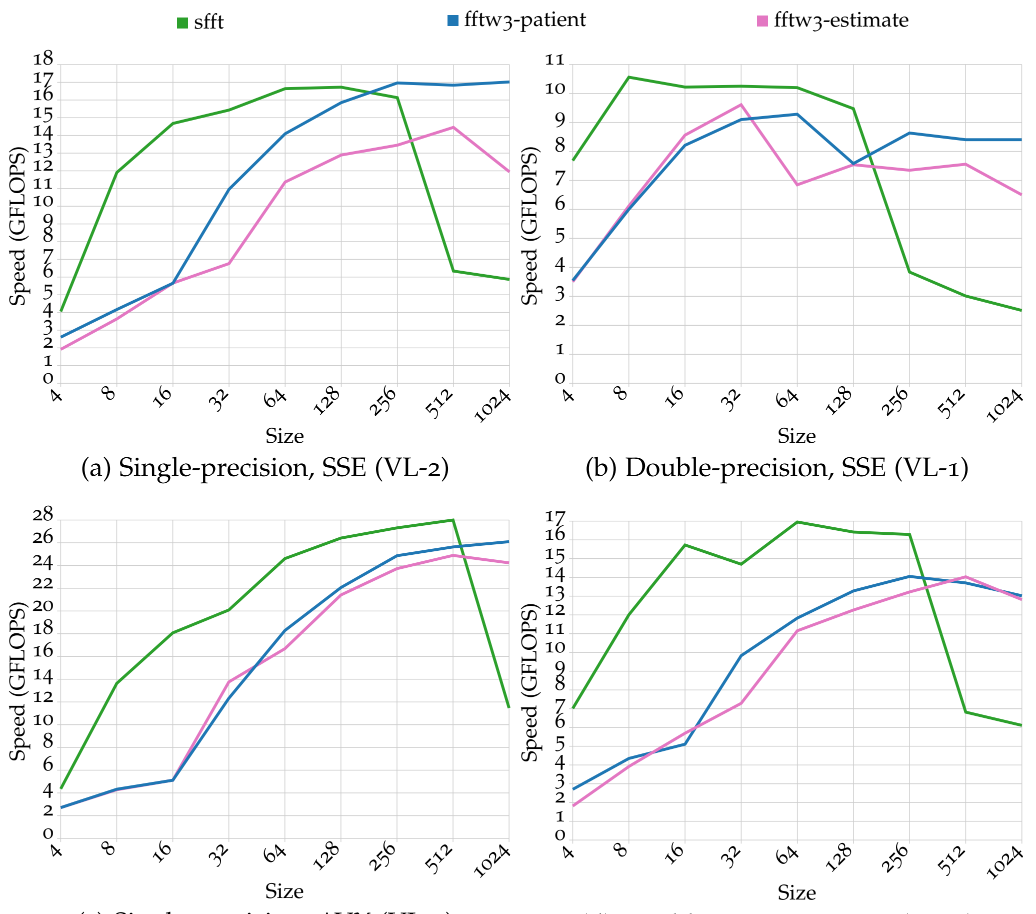
It is an easy exercise to code the leaf primitives for $\text{VL} \leqslant 4$ by hand, but for future machines that might feature vector lengths larger than 4, the leaf primitives could be automatically generated (in fact, Section 5.3.5 is concerned with automatic generation of leaf sub-transforms at another level of scale).

5.1.2.4 *Constraints*

For a transform of size N and leaf node size of S ($S = 4$ in the examples in this chapter), the following constraint must be satisfied:

$$N/\text{VL} \geqslant S \tag{35}$$

If this constraint is not satisfied, the size of either VL or S must be reduced. In practice, VL and S are small relative to the size of most transforms, and thus these corner cases typically only occur for very small sized transforms. Such an example is a size-2 transform when $\text{VL} = 2$ and $S = 4$, where in this case the transform is too small to be computed with SIMD operations and should be computed with scalar arithmetic instead.



(a) Single-precision, SSE (VL-2)

(b) Double-precision, SSE (VL-1)

(c) Single-precision, AVX (VL-4)

(d) Double-precision, AVX (VL-2)

Figure 7: Performance of hard-coded FFTs on a Macbook Air 4,2.

5.1.3 *Performance*

Figure 7 shows the results of a benchmark for transforms of size 4 through to 1024 running on a Macbook Air 4,2. The speed of FFTW 3.3 running in estimate and patient modes is also shown for comparison.

FFTW running in patient mode evaluates a huge configuration space of parameters, while the hard-coded FFT required no calibration.

A variety of vector lengths are represented, and the hard-coded FFTs have good performance while $N/VL \leqslant 128$. After this point, performance drops off and other techniques should be used. The following sections use the hard-coded FFT as a foundation for scaling to larger sizes of transforms.

5.2 HARD-CODED FOUR-STEP

This section presents an implementation of the four-step algorithm [8] that leverages hard-coded sub-transforms to compute larger transforms. The implementation uses an implicit memory transpose (along with vector register transposes) and scales particularly well with VL. In contrast to the fully hard-coded implementation in the previous section, the four-step implementation requires no new leaf primitives as VL increases, i.e., the code is much the same when $\mathrm{VL} > 1$ as it is when $\mathrm{VL} = 1$.

5.2.1 *The four-step algorithm*

A transform of size N is decomposed into a two-dimensional array of size $n_1 \times n_2$ where $N = n_1 n_2$. Selecting $n_1 = n_2 = \sqrt{N}$ (or close) often obtains the best performance results [8]. When either of the factors is larger than the other, it is the larger of the two factors that will determine performance, because the larger factor effectively brings the memory wall closer. The four steps of the algorithm are:

1. Compute n_1 FFTs of length n_2 along the columns of the array;
2. Multiply each element of the array with ω_N^{ij}, where i and j are the array coordinates;
3. Transpose the array;
4. Compute n_2 FFTs of length n_1 along the columns of the array.

For this out-of-place implementation, steps 2 and 3 are performed as part of step 1. Step 1 reads data from the input array and computes the FFTs, but before storing the data in the final pass, it is multiplied by the twiddle factors from step 2. After this, the data is stored to *rows* in the output array, and thus the transpose of step 3 is performed implicitly. Step 4 is then computed as usual: FFTs are computed along the columns of the output array.

This method of computing the four-step algorithm in two steps requires only minor modifications in order to support multiple vector lengths: with $\mathrm{VL} > 1$, multiple columns are read and computed in parallel without modification of the code, but before storing multiple columns of data to rows, a register transpose is required.

5.2.2 *Vector length 1*

When $\mathrm{VL} = 1$, three hard-coded FFTs are elaborated.

1. FFT of length n_2 with stride $n_1 \times 2$ for the first column of step 1;
2. FFT of length n_2 with stride $n_1 \times 2$ and twiddle multiplications on outputs – for all other columns of step 1;
3. FFT of length n_1 with stride $n_2 \times 2$ for columns in step 4.

In order to generate the code for the four-step sub-transforms, some minor modifications are made to the fully hard-coded code generator that was presented in the previous section.

The first FFT is used to handle the first column of step 1, where there are no twiddle factor multiplications because one of the array coordinates for step 2 is zero, and thus ω_N^0 is unity. This FFT may be elaborated as in Section 5.1.1 with the addition of a stride factor for the input address calculation. The second FFT is elaborated as per the first FFT, but with the addition of twiddle factor multiplications on each register prior to the store operations. The third FFT is elaborated as per the first FFT, but with strided input *and* output addresses.

5.2.2.1 *Example*

Listing 11 is a VL-1 size-64 hard-coded four-step FFT. Before it can be used, an initialization procedure (not shown) allocates and populates the LUT at line 1 with the twiddle factors that are required for the step 2 multiplications. Line 44 shows the main function that executes the first sub-transform on the first column (line 49), and the second sub-transform on all remaining columns (line 50). Finally, the sub-transforms corresponding to step 4 of the four-step algorithm are executed on all columns in line 51.

The twiddle factor multiplication that corresponds to step 2 of the four-step algorithm takes place in lines 21-23 and lines 26-29. The first register is not multiplied with a twiddle factor because the first row of twiddle factors are ω_N^0 (i.e., unity). The other registers are multiplied with two registers loaded from the LUT, which are the unpacked real and imaginary parts (see Section 3.3.1.3 for details about unpacked complex multiplication).

5.2.3 *Other vector lengths*

For $VL > 1$, the FFTs along the columns are computed in parallel. Thus, in step 1, n_1/VL FFTs are computed along the columns of the array with stride $= 2 \times VL$, and in step 4, n_2/VL FFTs are computed along the columns with stride $= 2 \times VL$.

Listing 11: Hard-coded four-step VL-1 size-64 FFT

```c
const SFFT_D __attribute__ ((aligned(32))) *LUT;
const SFFT_D *pLUT;
void sfft_dcf64_fs_x1_0(sfft_plan_t *p, const void *vin, void *vout){
  const SFFT_D *in = vin;
  SFFT_D *out = vout;
  SFFT_R r0,r1,r2,r3,r4,r5,r6,r7;
  L_4(in+0,in+64,in+32,in+96,&r0,&r1,&r2,&r3);
  L_2(in+16,in+80,in+112,in+48,&r4,&r5,&r6,&r7);
  K_0(&r0,&r2,&r4,&r6);
  S_4(r0,r2,r4,r6,out+0,out+4,out+8,out+12);
  K_N(VLIT2(0.7071,0.7071),VLIT2(0.7071,-0.7071),&r1,&r3,&r5,&r7);
  S_4(r1,r3,r5,r7,out+2,out+6,out+10,out+14);
}
void sfft_dcf64_fs_x1_n(sfft_plan_t *p, const void *vin, void *vout){
  const SFFT_D *in = vin;
  SFFT_D *out = vout;
  SFFT_R r0,r1,r2,r3,r4,r5,r6,r7;
  L_4(in+0,in+64,in+32,in+96,&r0,&r1,&r2,&r3);
  L_2(in+16,in+80,in+112,in+48,&r4,&r5,&r6,&r7);
  K_0(&r0,&r2,&r4,&r6);
  r2 = MUL(r2,LOAD(pLUT+4),LOAD(pLUT+6));
  r4 = MUL(r4,LOAD(pLUT+12),LOAD(pLUT+14));
  r6 = MUL(r6,LOAD(pLUT+20),LOAD(pLUT+22));
  S_4(r0,r2,r4,r6,out+0,out+4,out+8,out+12);
  K_N(VLIT2(0.7071,0.7071),VLIT2(0.7071,-0.7071),&r1,&r3,&r5,&r7);
  r1 = MUL(r1,LOAD(pLUT+0),LOAD(pLUT+2));
  r3 = MUL(r3,LOAD(pLUT+8),LOAD(pLUT+10));
  r5 = MUL(r5,LOAD(pLUT+16),LOAD(pLUT+18));
  r7 = MUL(r7,LOAD(pLUT+24),LOAD(pLUT+26));
  S_4(r1,r3,r5,r7,out+2,out+6,out+10,out+14);
  pLUT += 28;
}
void sfft_dcf64_fs_x2(sfft_plan_t *p, const void *vin, void *vout){
  const SFFT_D *in = vin;
  SFFT_D *out = vout;
  SFFT_R r0,r1,r2,r3,r4,r5,r6,r7;
  L_4(in+0,in+64,in+32,in+96,&r0,&r1,&r2,&r3);
  L_2(in+16,in+80,in+112,in+48,&r4,&r5,&r6,&r7);
  K_0(&r0,&r2,&r4,&r6);
  S_4(r0,r2,r4,r6,out+0,out+32,out+64,out+96);
  K_N(VLIT2(0.7071,0.7071),VLIT2(0.7071,-0.7071),&r1,&r3,&r5,&r7);
  S_4(r1,r3,r5,r7,out+16,out+48,out+80,out+112);
}
void sfft_dcf64_fs(sfft_plan_t *p, const void *vin, void *vout) {
  const SFFT_D *in = vin;
  SFFT_D *out = vout;
  pLUT =  LUT;
  int i;
  sfft_dcf64_fs_x1_0(p, in, out);
  for(i=1;i<8;i++) sfft_dcf64_fs_x1_n(p, in+(i*2), out+(i*16));
  for(i=0;i<8;i++) sfft_dcf64_fs_x2(p, out+(i*2), out+(i*2));
}
```

Listing 12: Hard-coded four-step VL-2 size-64 FFT

```
const SFFT_D __attribute__ ((aligned(32))) *LUT;
const SFFT_D *pLUT;
void sfft_fcf64_fs_x1(sfft_plan_t *p, const void *vin, void *vout) {
  const SFFT_D *in = vin;
  SFFT_D *out = vout;
  SFFT_R r0,r1,r2,r3,r4,r5,r6,r7;
  L_4(in+0,in+64,in+32,in+96,&r0,&r1,&r2,&r3);
  L_2(in+16,in+80,in+112,in+48,&r4,&r5,&r6,&r7);
  K_0(&r0,&r2,&r4,&r6);
  K_N(VLIT4(0.7071,0.7071,0.7071,0.7071),
      VLIT4(0.7071,-0.7071,0.7071,-0.7071),&r1,&r3,&r5,&r7);
  r1 = MUL(r1,LOAD(pLUT+0),LOAD(pLUT+4));
  TX2(r0,r1);
  r2 = MUL(r2,LOAD(pLUT+8),LOAD(pLUT+12));
  r3 = MUL(r3,LOAD(pLUT+16),LOAD(pLUT+20));
  TX2(r2,r3);
  r4 = MUL(r4,LOAD(pLUT+24),LOAD(pLUT+28));
  r5 = MUL(r5,LOAD(pLUT+32),LOAD(pLUT+36));
  TX2(r4,r5);
  r6 = MUL(r6,LOAD(pLUT+40),LOAD(pLUT+44));
  r7 = MUL(r7,LOAD(pLUT+48),LOAD(pLUT+52));
  TX2(r6,r7);
  S_4(r0,r2,r4,r6,out+0,out+4,out+8,out+12);
  S_4(r1,r3,r5,r7,out+16,out+20,out+24,out+28);
  pLUT += 56;
}
void sfft_fcf64_fs_x2(sfft_plan_t *p, const void *vin, void *vout) {
  const SFFT_D *in = vin;
  SFFT_D *out = vout;
  SFFT_R r0,r1,r2,r3,r4,r5,r6,r7;
  L_4(in+0,in+64,in+32,in+96,&r0,&r1,&r2,&r3);
  L_2(in+16,in+80,in+112,in+48,&r4,&r5,&r6,&r7);
  K_0(&r0,&r2,&r4,&r6);
  K_N(VLIT4(0.7071,0.7071,0.7071,0.7071),
      VLIT4(0.7071,-0.7071,0.7071,-0.7071),&r1,&r3,&r5,&r7);
  S_4(r0,r2,r4,r6,out+0,out+32,out+64,out+96);
  S_4(r1,r3,r5,r7,out+16,out+48,out+80,out+112);
}
void sfft_fcf64_fs(sfft_plan_t *p, const void *vin, void *vout) {
  const SFFT_D *in = vin;
  SFFT_D *out = vout;
  pLUT =  LUT;
  int i;
  for(i=0;i<4;i++) sfft_fcf64_fs_x1(p, in+(i*4), out+(i*32));
  for(i=0;i<4;i++) sfft_fcf64_fs_x2(p, out+(i*4), out+(i*4));
}
```

An implication of computing the first column in parallel with other columns is that the first column is now multiplied by unity twiddle factors, and thus only two sub-transforms are used instead of three.

The only other difference when $\mathrm{VL} > 1$ is that the registers need to be transposed before storing columns to rows (the implicit transpose that corresponds to step 3). To accomplish this when generating code, $n = \mathrm{VL}$ store operations are latched before the transpose and store code is emitted.

5.2.3.1 *Example*

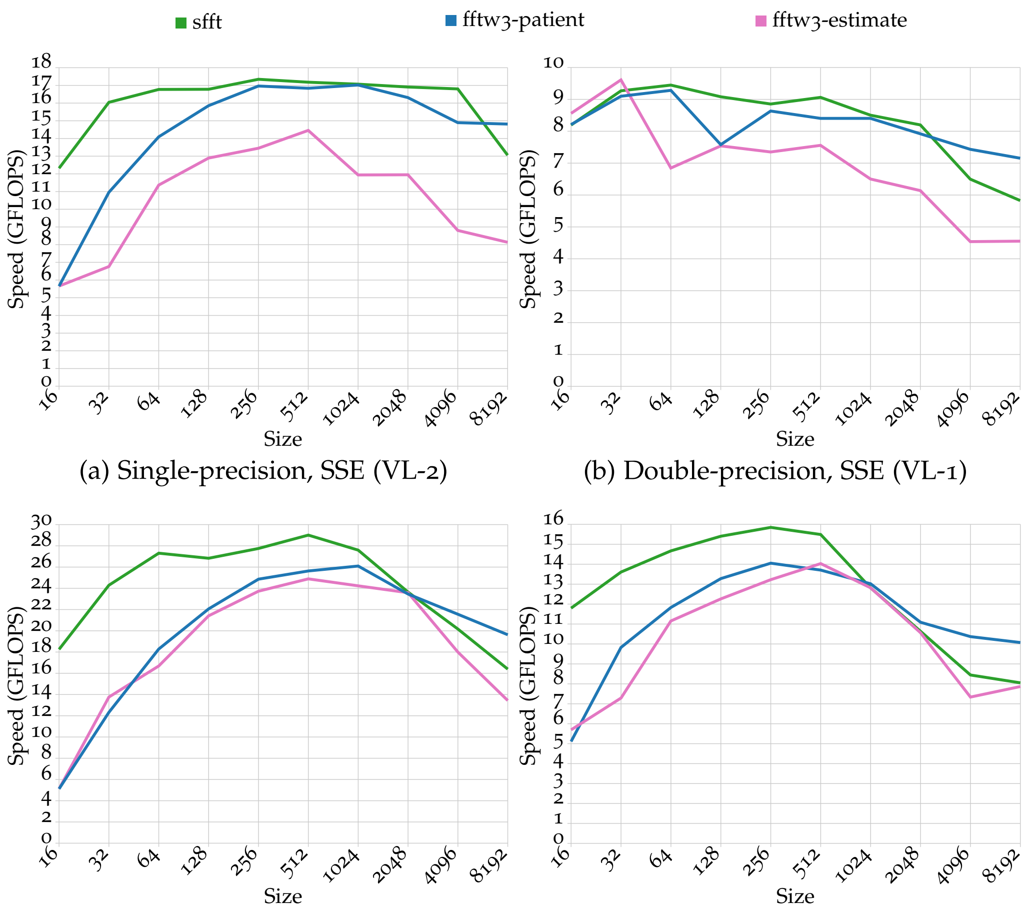
Listing 12 implements a VL-2 size-64 hard-coded four-step FFT. The main function (line 39) computes 8 FFTs along the columns for step 1 at line 44, and 8 FFTs along the columns for step 4 at line 45. There are only 4 iterations of the loop in each case because two sub-transforms are computed in parallel with each invocation of the sub-transform function.

In the function corresponding to the sub-transforms of step 1 (line 3), two store operations are latched (lines 23 and 24) before emitting code, which includes the preceding transposes (the `TX2` operations) and twiddle factor multiplications (lines 13–22).

5.2.4 *Performance*

Figure 9 shows the results of a benchmark for transforms of size 16 through to 8192 running on a Macbook Air 4,2. The speed of FFTW 3.3 running in estimate and patient modes is also shown for contrast.

The results show that the performance of the four-step algorithm improves as the length of the vector increases, but, as was the case with



(a) Single-precision, SSE (VL-2)

(b) Double-precision, SSE (VL-1)

(c) Single-precision, AVX (VL-4)

(d) Double-precision, AVX (VL-2)

Figure 9: Performance of hard-coded four-step FFTs on a Macbook Air 4,2.

the hard-coded FFTs in Section 5.1, the performance of the hard-coded four-step FFTs is limited to a certain range of transform size.

5.3 HARD-CODED LEAVES

The performance of the fully hard-coded transforms presented in Section 5.1 only scales while $N/VL \leqslant 128$. This section presents techniques that are similar to those found in the fully hard-coded transforms, but applied at another level of scale in order to scale performance to larger sizes.

5.3.1 *Vector length 1*

The fully hard-coded transforms in Section 5.1 used two primitives at the leaves: a size-4 sub-transform (`L_4`) and a double size-2 sub-transform (`L_2`). These sub-transforms loaded four elements of data from the input array, performed a small amount of computation, and stored the four results to the output array.

Performance is scaled to larger transforms by using larger sub-transforms at the leaves of the computation. These are automatically generated using fully hard-coded transforms, and thus the size of the leaf computations is easily parametrized, which is just as well, because the optimal leaf size is dependent on the size of the transform, the compiler, and the target machine.

The process of elaborating a topological ordering of nodes representing a hard-coded leaf transform of size N with leaf sub-transforms of size N_{leaf} is as follows:

1. Elaborate a size N_{leaf} sub-transform;
2. Elaborate a two size $N_{leaf}/2$ sub-transforms as one sub-transform;
3. Elaborate the main transform using the sub-transforms from steps 1 and 2 as the leaves of the computation.

The node lists for steps 1 and 2 are elaborated using the fully hard-coded `elaborate` function from Listing 6, but because the leaf sub-transform in step 2 is actually two sub-transforms of size $N_{leaf}/2$, the `elaborate` function is invoked twice with different offset parameters:

1. `elaborate(`$N_{leaf}/2$`, 0, 0, 1);`
2. `elaborate(`$N_{leaf}/2$`, −1, `$N_{leaf}/2$`, 1);`

The code corresponding to steps 1 and 2 is emitted slightly differently than was the case with the fully hard-coded transforms. Instead of hard coding the input array indices, the indices are themselves loaded from an array that is precomputed when the transform is initialized.

The node list corresponding to the main transform in step 3 is elaborated as in the function in Listing 6, but with some minor change. First, the recursion terminates with leaf nodes of size N_{leaf}. Second, because the loops in the body of the sub-transform will be at least $2 \times N_{leaf}$ iterations, the loop for the body sub-transforms (line 12 of Listing 6) is not statically unrolled. Instead only one node is added to the list of nodes, and the loop is computed dynamically.

5.3.1.1 *Example*

Listing 13 is a size-64 hard-coded leaf transform with size-16 leaves. The first function (lines 1–17) is a size-16 leaf sub-transform, while the second (lines 18–32) consists of two size-8 leaf sub-transforms in parallel. The main function (lines 36–46) invokes four leaf sub-transforms (lines 40, 41, 43 and 44), and two loops of body sub-transforms (lines 42 and 45).

The first parameter to the leaf functions (see lines 1 and 18) is a pointer into an array of precomputed indices for the input data array. At lines 41 and 43–44, the array is incremented before subsequent calls to the leaf functions, and at line 39 the pointer is reset to the base of the array so that the transform can be used repeatedly.

The function used for the body sub-transforms (lines 33–35) is a wrapper for a primitive that computes a radix-2/4 butterfly. The last parameter to this function is a pointer to a precomputed LUT of twiddle factors for a sub-transform of size N (the second parameter).

Listing 13: Hard-coded VL-1 size-64 FFT with size-16 leaves

```c
void sfft_dcf64_hcl16_4_e(offset_t *is,const SFFT_D *in,SFFT_D *out){
  SFFT_R r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14,r15;
  L_4(in+is[0],in+is[1],in+is[2],in+is[3],&r0,&r1,&r2,&r3);
  L_2(in+is[4],in+is[5],in+is[6],in+is[7],&r4,&r5,&r6,&r7);
  K_0(&r0,&r2,&r4,&r6);
  K_N(VLIT2(0.7071,0.7071),VLIT2(0.7071,-0.7071),&r1,&r3,&r5,&r7);
  L_4(in+is[8],in+is[9],in+is[10],in+is[11],&r8,&r9,&r10,&r11);
  L_4(in+is[12],in+is[13],in+is[14],in+is[15],&r12,&r13,&r14,&r15);
  K_0(&r0,&r4,&r8,&r12);
  S_4(r0,r4,r8,r12,out+0,out+8,out+16,out+24);
  K_N(VLIT2(0.9239,0.9239),VLIT2(0.3827,-0.3827),&r1,&r5,&r9,&r13);
  S_4(r1,r5,r9,r13,out+2,out+10,out+18,out+26);
  K_N(VLIT2(0.7071,0.7071),VLIT2(0.7071,-0.7071),&r2,&r6,&r10,&r14);
  S_4(r2,r6,r10,r14,out+4,out+12,out+20,out+28);
  K_N(VLIT2(0.3827,0.3827),VLIT2(0.9239,-0.9239),&r3,&r7,&r11,&r15);
  S_4(r3,r7,r11,r15,out+6,out+14,out+22,out+30);
}
void sfft_dcf64_hcl16_4_o(offset_t *is,const SFFT_D *in,SFFT_D *out){
  SFFT_R r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14,r15;
  L_4(in+is[0],in+is[1],in+is[2],in+is[3],&r0,&r1,&r2,&r3);
  L_2(in+is[4],in+is[5],in+is[6],in+is[7],&r4,&r5,&r6,&r7);
  K_0(&r0,&r2,&r4,&r6);
  S_4(r0,r2,r4,r6,out+0,out+4,out+8,out+12);
  K_N(VLIT2(0.7071,0.7071),VLIT2(0.7071,-0.7071),&r1,&r3,&r5,&r7);
  S_4(r1,r3,r5,r7,out+2,out+6,out+10,out+14);
  L_4(in+is[8],in+is[9],in+is[10],in+is[11],&r8,&r9,&r10,&r11);
  L_2(in+is[12],in+is[13],in+is[14],in+is[15],&r12,&r13,&r14,&r15);
  K_0(&r8,&r10,&r12,&r14);
  S_4(r8,r10,r12,r14,out+16,out+20,out+24,out+28);
  K_N(VLIT2(0.7071,0.7071),VLIT2(0.7071,-0.7071),&r9,&r11,&r13,&r15);
  S_4(r9,r11,r13,r15,out+18,out+22,out+26,out+30);
}
void sfft_dcf64_hcl16_4_X_4(SFFT_D *data, int N, SFFT_D *LUT){
  X_4(data, N, LUT);
}
void sfft_dcf64_hcl16_4(sfft_plan_t *p, const void *vin, void *vout){
  const SFFT_D *in = vin;
  SFFT_D *out = vout;
  p->is = p->is_base;
  sfft_dcf64_hcl16_4_e(p->is,in,out+0);
  p->is += 16;  sfft_dcf64_hcl16_4_o(p->is,in,out+32);
  sfft_dcf64_hcl16_4_X_4(out+0,32,p->ws[0]);
  p->is += 16;  sfft_dcf64_hcl16_4_e(p->is,in,out+64);
  p->is += 16;  sfft_dcf64_hcl16_4_e(p->is,in,out+96);
  sfft_dcf64_hcl16_4_X_4(out+0,64,p->ws[1]);
}
```

SIZE	INPUT ARRAY ADDRESSES
16	{0, 64, 32, 96, 16, 80, 112, 48, 8, 72, 40, 104, 120, 56, 24, 88}
8(x2)	{4, 68, 36, 100, 20, 84, 116, 52, 124, 60, 28, 92, 12, 76, 108, 44}
16	{2, 66, 34, 98, 18, 82, 114, 50, 10, 74, 42, 106, 122, 58, 26, 90}
16	{126, 62, 30, 94, 14, 78, 110, 46, 6, 70, 38, 102, 118, 54, 22, 86}
16	{1, 65, 33, 97, 17, 81, 113, 49, 9, 73, 41, 105, 121, 57, 25, 89}
8(x2)	{5, 69, 37, 101, 21, 85, 117, 53, 125, 61, 29, 93, 13, 77, 109, 45}
16	{127, 63, 31, 95, 15, 79, 111, 47, 7, 71, 39, 103, 119, 55, 23, 87}
8(x2)	{3, 67, 35, 99, 19, 83, 115, 51, 123, 59, 27, 91, 11, 75, 107, 43}

Table 5: Size-16 leaf nodes in VL-1 size-128 hard-coded leaf FFT

SIZE	INPUT ARRAY ADDRESSES
16	{0, 64, 32, 96, 16, 80, 112, 48, 8, 72, 40, 104, 120, 56, 24, 88}
16	{1, 65, 33, 97, 17, 81, 113, 49, 9, 73, 41, 105, 121, 57, 25, 89}
16	{2, 66, 34, 98, 18, 82, 114, 50, 10, 74, 42, 106, 122, 58, 26, 90}
8(x2)	{3, 67, 35, 99, 19, 83, 115, 51, 123, 59, 27, 91, 11, 75, 107, 43}
8(x2)	{4, 68, 36, 100, 20, 84, 116, 52, 124, 60, 28, 92, 12, 76, 108, 44}
8(x2)	{5, 69, 37, 101, 21, 85, 117, 53, 125, 61, 29, 93, 13, 77, 109, 45}
16	{126, 62, 30, 94, 14, 78, 110, 46, 6, 70, 38, 102, 118, 54, 22, 86}
16	{127, 63, 31, 95, 15, 79, 111, 47, 7, 71, 39, 103, 119, 55, 23, 87}

Table 6: Sorted size-16 leaf nodes in VL-1 size-128 hard-coded leaf FFT

5.3.2 *Improving memory locality in the leaves*

Table 5 lists the addresses of data loaded by each of the size-16 leaf nodes in a size-128 transform. It is difficult to improve the locality of accesses within a leaf sub-transform (doing so would require the use of expensive transposes), but the order of the leaf sub-transforms can be changed to yield better locality between sub-transforms.

Table 6 is the list of nodes from Table 5 after the rows have been sorted according to the minimum address in each row. There are now three distinct groups in the list: the first three sub-transforms of size-16, the second three sub-transforms of 2x size-8, and the final two sub-transforms of size-16. The memory accesses are now linear between consecutive sub-transforms, though the second and third groups operate on a permuted ordering of the addresses.

The pattern exhibited by Table 6 can be exploited to access the data stored in the input array with better locality, as Figures 10 and 11 show. Figure 10 depicts the memory access pattern of an FFT with size-16 hard-coded leaves, while Figure 11 depicts the same FFT with *sorted* hard-coded leaves.

To compute the FFT with sorted leaves, the leaf sub-transforms and the body sub-transforms are split into two separate lists, and the entire list of leaf sub-transforms is computed before any of the body sub-transforms. There is, however, a cost associated with this re-arrangement: each leaf sub-transform's offset into the output array is not easy to compute because the offsets are now essentially decimated-in-frequency, and thus they are now pre-computed. Overall, the trade-off is justified because the output memory accesses within each leaf sub-transform are still linear.

The leaf transforms can be computed in three loops. The first and third loops compute size-N_{leaf} sub-transforms, while the second loop computes size-$N_{leaf}/2$ sub-transforms. The size of the three loops i_0, i_1 and i_2 are:

$$i_0 = \left\lfloor \tfrac{N}{3 \times N_{leaf}} \right\rfloor + 1 \tag{36}$$

$$i_1 = \left\lfloor \frac{N}{3 \times N_{leaf}} \right\rfloor + \left\lfloor \left(\frac{N}{N_{leaf}} \mod 3 \right) \times \frac{1}{2} \right\rfloor \tag{37}$$

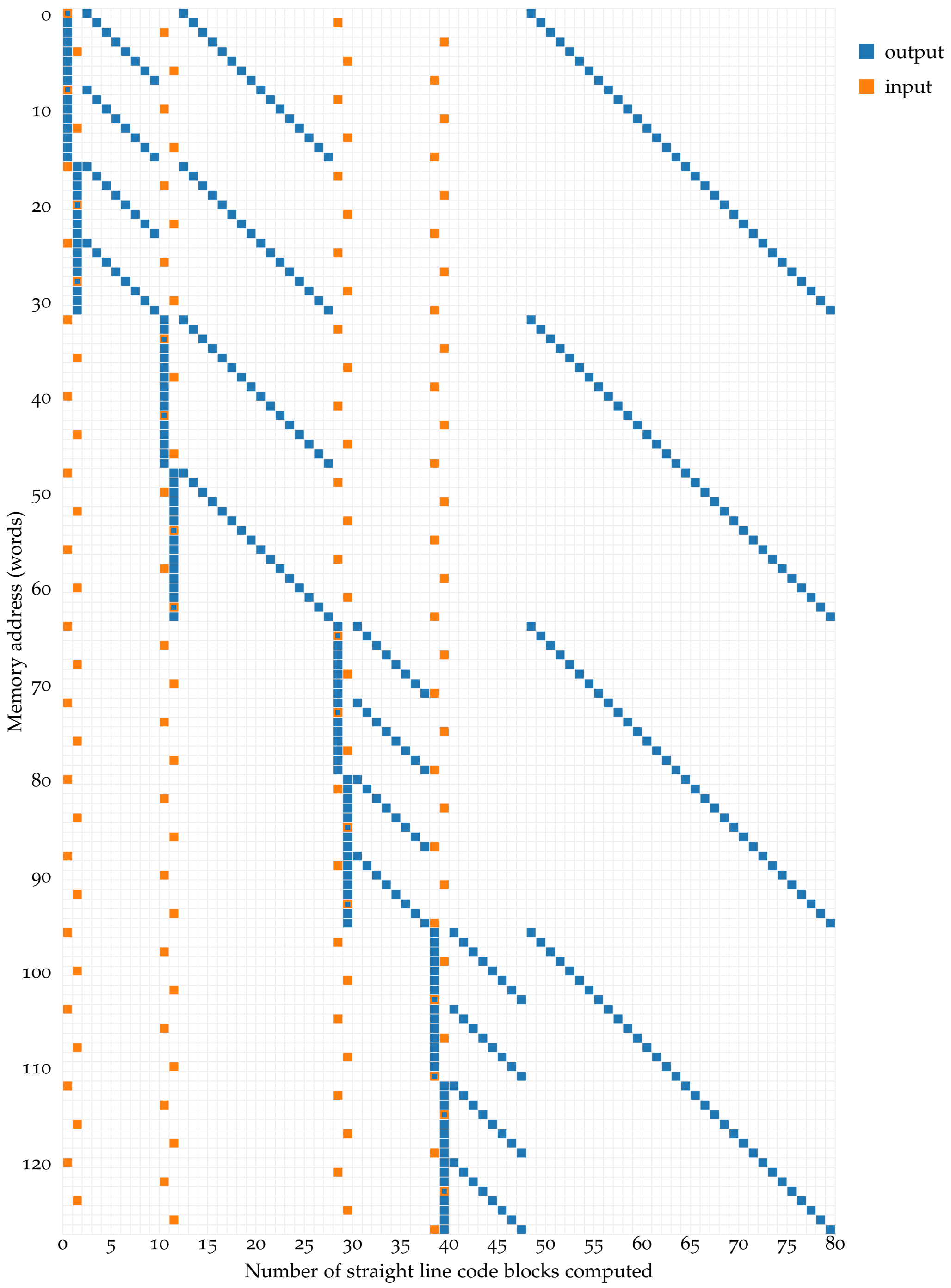


Figure 10: Memory access pattern of the straight line blocks of code in a VL-1 size-128 hard-coded leaf FFT

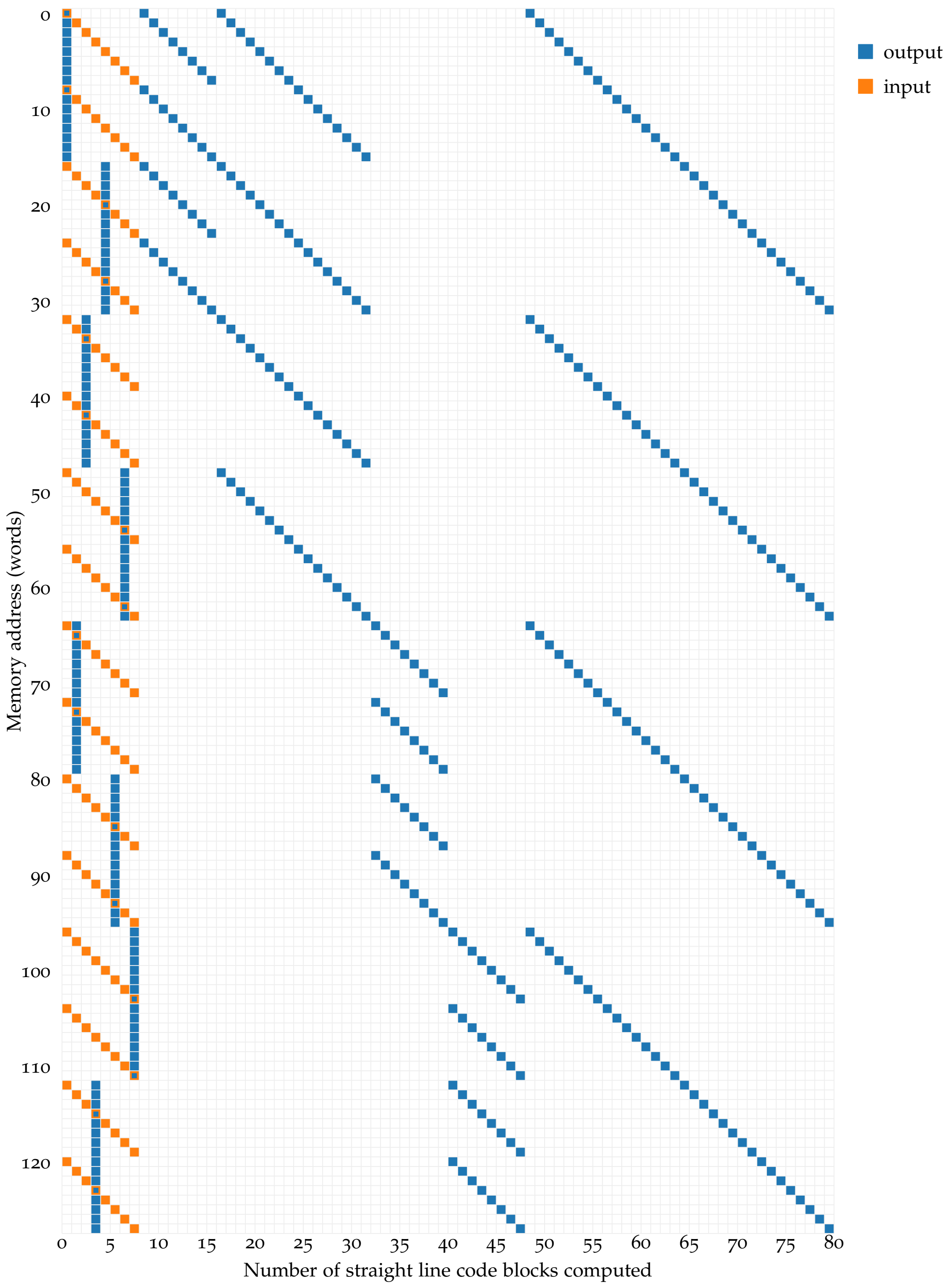


Figure 11: Memory access pattern of the straight line blocks of code in a VL-1 size-128 hard-coded leaf FFT after leaf node sorting

and

$$i_2 = \left\lfloor \frac{N}{3 \times N_{leaf}} \right\rfloor \tag{38}$$

The transform can now be elaborated *without* leaf nodes, and the code for the three loops emitted in the place of calls to individual leaf sub-transforms.

5.3.2.1 *Example*

Listing 14 is the main function for the FFT that corresponds to the leaf node list in Table 6. The first and third loops invoke size-16 sub-transforms at lines 8 and 16, and the second loop invokes 2x size-8 sub-transforms at line 12. Following the leaf sub-transforms, the body sub-transforms are called at lines 19-23.

5.3.2.2 *Scalability*

In terms of code size, computing the leaf sub-transforms with three loops is economical. As the size of the transform grows, the code size attributed to the leaf sub-transforms remains constant. However, as the size of the transform begins to grow large (e.g., $\geqslant 65,536$), the instructions required for the body sub-transform calls (lines 19-23 in Listing 14) begins to dominate the overall program size. Section 5.3.4 describes a method for compressing the code size of the body sub-transform calls while maintaining performance.

Because the input array references between consecutive leaves are now linear, and like types of leaf sub-transforms are grouped together, it is now possible to compute several leaf sub-transforms in parallel, which is fully described in Section 5.3.5.

Listing 14: Hard-coded VL-1 size-128 FFT with size-16 leaves (sub-transforms omitted)

```
void sfft_dcf128_shl16_4(sfft_plan_t *p,const void *vin,void *vout){
  const SFFT_D *in = vin;
  SFFT_D *out = vout;
  offset_t *is = p->is_base;
  offset_t *offsets = p->offsets_base;
  int i;
  for(i=3;i>0;--i) {
    sfft_dcf128_shl16_4_e(is, in, out+offsets[0]);
    is += 16; offsets += 1;
  }
  for(i=3;i>0;--i) {
    sfft_dcf128_shl16_4_o(is, in, out+offsets[0]);
    is += 16; offsets += 1;
  }
  for(i=2;i>0;--i) {
    sfft_dcf128_shl16_4_e(is, in, out+offsets[0]);
    is += 16; offsets += 1;
  }
  sfft_dcf128_shl16_4_X_4(out+0, 32, p->ws[0]);
  sfft_dcf128_shl16_4_X_4(out+0, 64, p->ws[1]);
  sfft_dcf128_shl16_4_X_4(out+128, 32, p->ws[0]);
  sfft_dcf128_shl16_4_X_4(out+192, 32, p->ws[0]);
  sfft_dcf128_shl16_4_X_4(out+0, 128, p->ws[2]);
}
```

5.3.3 *Body sub-transform radix*

The radix of the body sub-transforms can be increased in order to reduce the number of passes over the data and make better use of the cache. In practice, the body sub-transform radix is limited by the associativity of the cache as the size of the transform increases. If the radix is greater than the associativity of the nearest level of cache in which a sub-transform cannot fit, there will be cache misses for every iteration of the sub-transform's loop, resulting in severely degraded performance.

All Intel SIMD microprocessors since the Netburst micro-architecture have had at least 8-way associativity in all levels of cache, and thus increasing the radix from 4 to 8 is a sensible decision when targeting Intel machines.

Just as the split-radix 2/4 algorithm requires two different types of leaf sub-transforms, a split-radix 2/8 algorithm would require three, which increases the complexity of statically elaborating and generating code. There is an alternative that does not require implementing three types of leaf sub-transform: where a size-N body sub-transform divides into a size N/2 body sub-transform and two size N/4 sub-transforms, the size N and size N/2 sub-transforms may be collected together and computed as a size-8 sub-transform. Thus the transform is computed with two types of leaf sub-transform and two types of body sub-transform, instead of three types of leaf sub-transform and one type of body sub-transform, as with the standard split-radix 2/8 algorithm.

For the size-128 tranform in Listing 14, either the sub-transform at line 19 can be subsumed into the sub-transform at line 20, or the sub-transform at line 20 can be subsumed into the sub-transform at line 23 – but not both. The latter choice is better because it involves larger transforms.

The code in Listing 15 iterates in reverse over a list of sub-transforms and doubles the radix of the body sub-transforms. Because the list may include multiple types, type introspection at lines 6 and 20 filters out all types that are not body sub-transforms. For each body sub-transform, the `increase_body_radix` function searches upwards through the list for a subsumable body sub-transform (using `find_subsumable_sub_transform`) and if a match is found, the smaller sub-transform is removed from the list, and the size of the larger sub-transform is doubled.

Figure 12 depicts the memory access patterns of a size-128 transform where the outermost body sub-transform has subsumed a smaller sub-

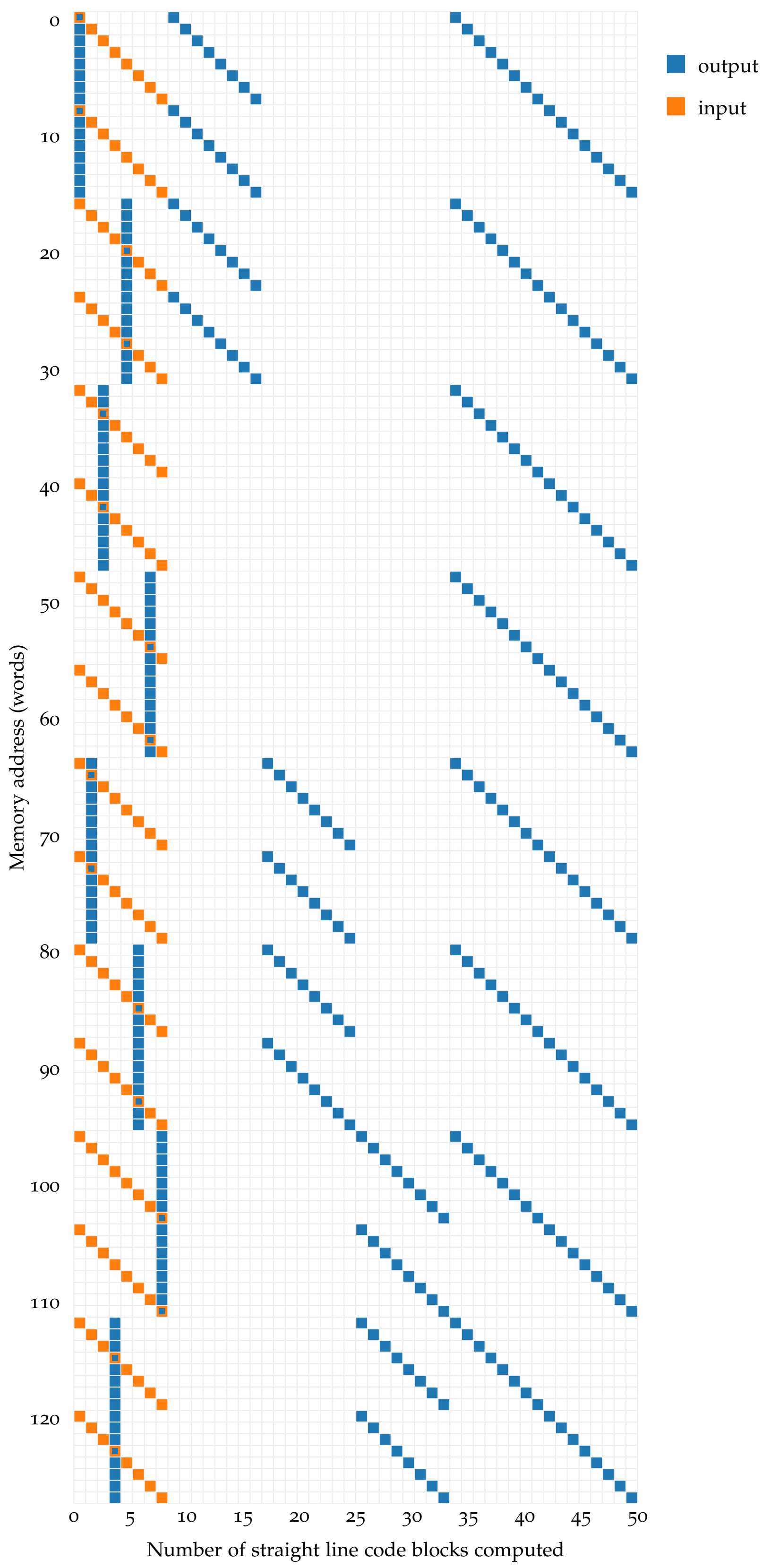


Figure 12: Memory access pattern of the straight line blocks of code in a VL-1 size-128 hard-coded leaf FFT with sorted radix-2/4 and size-8 body sub-transforms

Listing 15: Doubling the radix of body sub-transforms

```
CBody *CHardCodedLeaf::find_subsumable_sub_transform(
       vector<CNode *>::reverse_iterator i) {
  CBody *first = (CBody *)(*i);  i++;
  while(i != bs.rend()) {
    if(!((*i)->type().compare("body"))) {
      CBody *second = (CBody *)(*i);
      if(first->N == second->N*2 && first->offset == second->offset){
        bs.erase((++i).base());
        return second;
      }
    }
    ++i;
  }
  return NULL;
}
void CHardCodedLeaf::increase_body_radix(void) {
  vector<CNode *>::reverse_iterator ri;
  for(ri=bs.rbegin(); ri!=bs.rend(); ++ri) {
    if(!((*ri)->type().compare("body"))) {
      CBody *n1 = (CBody *)(*ri);
      CBody *n2 = find_subsumable_sub_transform(ri);
      if(n2) n1->size *= 2;
    }
  }
}
```

transform to become a size-8 sub-transform. The columns from 33 onwards show the sub-transform accessing eight elements in the output data array (cf. Figure 11, which shows the memory access patterns of the same transform prior to doubling the radix of the outer sub-transform).

5.3.4 *Optimizing the hierarchical structure*

The largest transform that has been considered so far is size-128. As it stands, the hard-coded leaf approach begins to generate code of unwieldy proportions as the size of the transform tends towards tens of thousands or hundreds of thousands of points. This is due to the lists of statically elaborated body sub-transform calls, e.g., a size-262,144 transform contains a lengthy list of 7279 such calls.

While long lists of statically elaborated calls are one extreme, the other is to compute the body sub-transforms with a recursive program. The former option degrades performance for larger transforms, while the latter option curbs performance for smaller transforms. A compromise is to somehow compress blocks of statically elaborated sub-transform calls.

The approach presented here extracts the hierarchical structure from the sequence of body sub-transforms and emits a set of functions that are neither too small (as in the case of a recursive program) nor too large (as is the case with full static elaboration). This is accomplished by adapting the Sequitur algorithm [63], which builds a grammar of rules from a sequence of symbols, and enforces two basic constraints:

1. no pair of adjacent symbols (referred to as a digram) appears more than once in the grammar;
2. every rule is used more than once.

The resulting grammar is an efficient hierarchical representation of the original sequence. Additional constraints can be imposed to limit the maximum or minimum size of each rule, which enable the size of the resulting functions to be tuned to be not too small and not too large.

To build the grammar, each body sub-transform is represented by a symbol consisting of the size and offset of the sub-transform. The radix is discarded, because it can be inferred from the size. Here are several other details relevant to this particular application of Sequitur:

- A digram of two sub-transforms is deemed to match another digram when the size of each sub-transform matches the size of the other digram's respective sub-transform *and* the relative offsets between sub-transforms within each digram match;

- Sub-transform offsets are maintained to be always relative to the base of the containing rule – when a rule is constructed, the offsets of the symbols within that rule are adjusted to be relative to the base of the new rule, and when a rule is subsumed (due to violation of constraint 2: every rule must be used more than once), the offsets are recomputed to be relative to the subsuming rule.

5.3.4.1 *Example*

A size-8192 hard-coded leaf FFT requires 229 calls to radix-2/4 and size-8 body sub-transforms. After optimizing the sequence of calls with Sequitur, the compact set of functions shown in Listing 16 replaces a sequence of 229 calls.

Compared to the full list of statically elaborated calls, the optimized set of functions requires less code space while achieving better performance; and compared to a recursive program, the optimized set of function calls is faster (due to lower call and stack overhead) while trading off an acceptably small amount of code space.

5.3.4.2 *Scalability*

The technique presented in this section has been verified for transforms ranging in size from 2^6 through to 2^{25} (32 mega) points. The technique works well up until sizes of about 2^{18} points, but for larger transforms the elaboration and compile times begin to exceed 1 second or so, and the code size again begins to grow large. For transforms larger than 2^{18} points, a recursive program can be used until leaves of size 2^{18} points are reached, at which point the technique presented in this section is used.

Listing 16: Optimized body sub-transforms for size-8192 FFT

```
void sfft_dcf8192_shl16_8_4(sfft_plan_t *p, SFFT_D *out) {
  X_4(out+0, 32, p->ws[0]);
  X_4(out+128, 32, p->ws[0]);
  X_4(out+192, 32, p->ws[0]);
  X_8(out+0, 128, p->ws[2]);
}
void sfft_dcf8192_shl16_8_5(sfft_plan_t *p, SFFT_D *out) {
  X_8(out+0, 64, p->ws[1]);
  X_8(out+128, 64, p->ws[1]);
}
void sfft_dcf8192_shl16_8_9(sfft_plan_t *p, SFFT_D *out) {
  X_8(out+0, 64, p->ws[1]);
  X_4(out+128, 32, p->ws[0]);
  X_4(out+192, 32, p->ws[0]);
  sfft_dcf8192_shl16_8_5(p, out+256);
  X_8(out+0, 256, p->ws[3]);
}
void sfft_dcf8192_shl16_8_13(sfft_plan_t *p, SFFT_D *out) {
  sfft_dcf8192_shl16_8_4(p, out+0);
  sfft_dcf8192_shl16_8_5(p, out+256);
  sfft_dcf8192_shl16_8_4(p, out+512);
  sfft_dcf8192_shl16_8_4(p, out+768);
  X_8(out+0, 512, p->ws[4]);
}
void sfft_dcf8192_shl16_8_14(sfft_plan_t *p, SFFT_D *out) {
  sfft_dcf8192_shl16_8_9(p, out+0);
  sfft_dcf8192_shl16_8_9(p, out+512);
}
void sfft_dcf8192_shl16_8_18(sfft_plan_t *p, SFFT_D *out) {
  sfft_dcf8192_shl16_8_9(p, out+0);
  sfft_dcf8192_shl16_8_4(p, out+512);
  sfft_dcf8192_shl16_8_4(p, out+768);
  sfft_dcf8192_shl16_8_14(p, out+1024);
  X_8(out+0, 1024, p->ws[5]);
}
void sfft_dcf8192_shl16_8_22(sfft_plan_t *p, SFFT_D *out) {
  sfft_dcf8192_shl16_8_13(p, out+0);
  sfft_dcf8192_shl16_8_14(p, out+1024);
  sfft_dcf8192_shl16_8_13(p, out+2048);
  sfft_dcf8192_shl16_8_13(p, out+3072);
  X_8(out+0, 2048, p->ws[6]);
}
void sfft_dcf8192_shl16_8_1(sfft_plan_t *p, SFFT_D *out) {
  sfft_dcf8192_shl16_8_22(p, out+0);
  sfft_dcf8192_shl16_8_18(p, out+4096);
  sfft_dcf8192_shl16_8_18(p, out+6144);
  sfft_dcf8192_shl16_8_22(p, out+8192);
  sfft_dcf8192_shl16_8_22(p, out+12288);
  X_8(out+0, 8192, p->ws[8]);
}
```

SIZE	INPUT ARRAY ADDRESSES
16	{0, 64, 32, 96, 16, 80, 112, 48, 8, 72, 40, 104, 120, 56, 24, 88}
16	{1, 65, 33, 97, 17, 81, 113, 49, 9, 73, 41, 105, 121, 57, 25, 89}
16	{2, 66, 34, 98, 18, 82, 114, 50, 10, 74, 42, 106, 122, 58, 26, 90}
8(x2)	{3, 67, 35, 99, 19, 83, 115, 51, 123, 59, 27, 91, 11, 75, 107, 43}
8(x2)	{4, 68, 36, 100, 20, 84, 116, 52, 124, 60, 28, 92, 12, 76, 108, 44}
8(x2)	{5, 69, 37, 101, 21, 85, 117, 53, 125, 61, 29, 93, 13, 77, 109, 45}
16	{126, 62, 30, 94, 14, 78, 110, 46, 6, 70, 38, 102, 118, 54, 22, 86}
16	{127, 63, 31, 95, 15, 79, 111, 47, 7, 71, 39, 103, 119, 55, 23, 87}

Table 7: Sorted size-16 leaf nodes in size-128 hard-coded leaf FFT, grouped for VL-2

5.3.5 *Other vector lengths*

The method of vectorizing the hard-coded leaf FFT is similar to that of the hard-coded FFT in Section 5.1.2; the only difference here is the level of scale.

The hard-coded FFT was vectorized by collecting together primitive leaf operations that loaded data from adjacent memory locations. The hard-coded leaf FFT has already been sorted such that consecutive leaf sub-transforms load data from adjacent memory locations (see Section 5.3.2), so the task is easier in this case – at least in one respect.

Table 7 shows the sorted size-16 leaf sub-transforms for a size-128 transform with the rows divided into VL-2 groups. Because each group of two leaf sub-transforms loads data from adjacent memory locations, the group of sub-transforms can be loaded in parallel with vector memory operations, and all (or some) of the computation done in parallel. The first, third and fourth groups in Table 7 contain leaf nodes of the same size/type; these are the easiest vector leaf sub-transforms to compute, as described in Section 5.3.5.1. The second group of rows con-

Listing 17: Homogeneous size-16 leaf sub-transform for VL-2 size-128 hard-coded leaf FFT

```c
 1  void
 2  sfft_fcf128_shl16_8_ee(offset_t *is,const SFFT_D *in,SFFT_D *out){
 3    SFFT_R r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14,r15;
 4    L_4(in+is[0],in+is[1],in+is[2],in+is[3],&r0,&r1,&r2,&r3);
 5    L_2(in+is[4],in+is[5],in+is[6],in+is[7],&r4,&r5,&r6,&r7);
 6    K_0(&r0,&r2,&r4,&r6);
 7    K_N(VLIT4(0.7071,0.7071,0.7071,0.7071),
 8        VLIT4(0.7071,-0.7071,0.7071,-0.7071),
 9        &r1,&r3,&r5,&r7);
10    L_4(in+is[8],in+is[9],in+is[10],in+is[11],&r8,&r9,&r10,&r11);
11    L_4(in+is[12],in+is[13],in+is[14],in+is[15],&r12,&r13,&r14,&r15);
12    K_0(&r0,&r4,&r8,&r12);
13    K_N(VLIT4(0.9239,0.9239,0.9239,0.9239),
14        VLIT4(0.3827,-0.3827,0.3827,-0.3827),
15        &r1,&r5,&r9,&r13);
16    TX2(&r0,&r1); TX2(&r4,&r5); TX2(&r8,&r9); TX2(&r12,&r13);
17    S_4(r0,r4,r8,r12,out0+0,out0+8,out0+16,out0+24);
18    S_4(r1,r5,r9,r13,out1+0,out1+8,out1+16,out1+24);
19    K_N(VLIT4(0.7071,0.7071,0.7071,0.7071),
20        VLIT4(0.7071,-0.7071,0.7071,-0.7071),
21        &r2,&r6,&r10,&r14);
22    K_N(VLIT4(0.3827,0.3827,0.3827,0.3827),
23        VLIT4(0.9239,-0.9239,0.9239,-0.9239),
24        &r3,&r7,&r11,&r15);
25    TX2(&r2,&r3); TX2(&r6,&r7); TX2(&r10,&r11); TX2(&r14,&r15);
26    S_4(r2,r6,r10,r14,out0+4,out0+12,out0+20,out0+28);
27    S_4(r3,r7,r11,r15,out1+4,out1+12,out1+20,out1+28);
28  }
```

tains leaf sub-transforms of differing size/type, and computing these sub-transforms is covered separately in Section 5.3.5.2.

5.3.5.1 *Homogeneous leaf sub-transform vectors*

The vector leaf sub-transforms of a single size/type are handled in the same way as a VL-1 sub-transform, with one difference: the vector registers must be transposed before the data is stored to memory in the output array. In the example shown in Listing 17, the transposes take place at lines 16 and 25.

Listing 18: Heterogeneous size-16 leaf sub-transform for VL-2 size-128 hard-coded leaf FFT

```
void
sfft_fcf128_shl16_8_eo(offset_t *is,const SFFT_D *in,SFFT_D *out){
  SFFT_R r0_1,r2_3,r4_5,r6_7,r8_9,r10_11,r12_13,r14_15,
  r16_17,r18_19,r20_21,r22_23,r24_25,r26_27,r28_29,r30_31;
  L_4_4(in+is[0],in+is[1],in+is[2],in+is[3],
      &r0_1,&r2_3,&r16_17,&r18_19);
  L_2_2(in+is[4],in+is[5],in+is[6],in+is[7],
      &r4_5,&r6_7,&r20_21,&r22_23);
  K_N(VLIT4(0.7071,0.7071,1,1),VLIT4(0.7071,-0.7071,0,-0),
      &r0_1,&r2_3,&r4_5,&r6_7);
  L_4_2(in+is[8],in+is[9],in+is[10],in+is[11],
        &r8_9,&r10_11,&r28_29,&r30_31);
  L_4_4(in+is[12],in+is[13],in+is[14],in+is[15],
        &r12_13,&r14_15,&r24_25,&r26_27);
  K_N(VLIT4(0.9239,0.9239,1,1),VLIT4(0.3827,-0.3827,0,-0),
      &r0_1,&r4_5,&r8_9,&r12_13);
  S_4(r0_1,r4_5,r8_9,r12_13,out0+0,out0+8,out0+16,out0+24);
  K_N(VLIT4(0.3827,0.3827,0.7071,0.7071),
      VLIT4(0.9239,-0.9239,0.7071,-0.7071),
      &r2_3,&r6_7,&r10_11,&r14_15);
  S_4(r2_3,r6_7,r10_11,r14_15,out0+4,out0+12,out0+20,out0+28);
  K_N(VLIT4(0.7071,0.7071,1,1),VLIT4(0.7071,-0.7071,0,-0),
      &r16_17,&r18_19,&r20_21,&r22_23);
  S_4(r16_17,r18_19,r20_21,r22_23,out1+0,out1+4,out1+8,out1+12);
  K_N(VLIT4(0.7071,0.7071,1,1),VLIT4(0.7071,-0.7071,0,-0),
      &r24_25,&r26_27,&r28_29,&r30_31);
  S_4(r24_25,r26_27,r28_29,r30_31,out1+16,out1+20,out1+24,out1+28);
}
```

Prior to the store operations, each position of the vector register (each position being a whole complex word) contains an element belonging to each of the leaf sub-transforms composing the vectorized sub-transform. Because each leaf sub-transform is stored sequentially to different locations in memory with aligned vector store operations, sets of registers are transposed such that each vector register contains elements from only one leaf sub-transform.

5.3.5.2 *Heterogeneous leaf sub-transform vectors*

In the case of a vector comprising heterogeneous leaf sub-transforms, the data is transposed into separate sub-transforms following the primitive leaf operations. The remainder of the computation is carried out separately for each leaf sub-transform in the vector, and no further transposes are required.

When elaborating and generating code for VL-2 transforms, there are only two heterogeneous leaf sub-transforms that might be required, but for other vector lengths the combinations are more complex. During the elaboration process, each unique combination that is encountered in the sorted list of leaf sub-transforms is elaborated into a function with repeated calls to the `elaborate` function, as was done in Section 5.3.1 in order to elaborate a sub-transform composed of two size $N_{\mathtt{leaf}}/2$ sub-transforms.

Listing 18 is an example of a heterogeneous size-16 VL-2 leaf sub-transform, where one size-16 leaf sub-transform is loaded into the lower halves of the vector registers, and the data from another leaf sub-transform composed of two size-8 sub-transforms is loaded into the upper halves. The primitive leaf operations at lines 5, 7, 11 and 13 transpose each sub-transform's data into separate vector registers, and the remainder of the computation is performed on each sub-transform separately. The size-16 sub-transform is stored to sequential locations in memory at lines 17 and 21, while the sub-transform composed of two size-8 leaf sub-transforms is stored to memory at lines 24 and 27.

5.3.6 *Streaming stores*

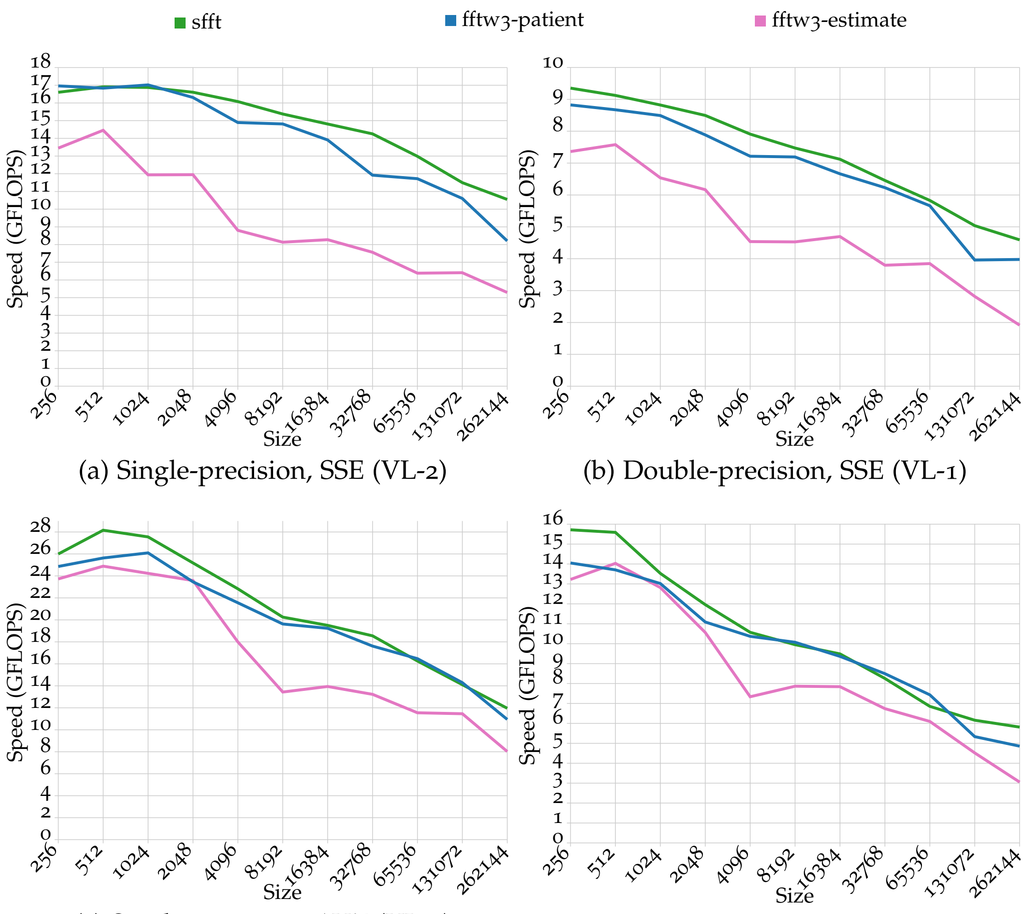
Some machines support streaming store or non-temporal store instructions; these instructions are used to store data to locations that do not have temporal locality, and thus the cache can be bypassed. The hard-coded leaf FFT described in the previous sections splits the computation into a pass of leaf sub-transforms and several passes of body sub-transforms. For large transforms where the size of the data exceeds the outermost level of cache, the non-temporal store instructions can be used in the leaf sub-transforms to bypass the cache when storing data to the output array; this can greatly improve performance by keeping other data in cache. The Intel SSE and AVX vector extensions both support streaming stores.

5.3.7 *Performance*

Figure 14 shows the results of a benchmark for transforms of size 256 through to 262,144 running on a Macbook Air 4,2. The speed of FFTW 3.3 running in estimate and patient modes is also shown for comparison.

For each size of transform, precision and vector length (i.e., either SSE or AVX), several configurations of hard-coded leaf FFT were generated: three configurations of leaf size (16, 32 and 64), and if the transform was larger than 32,768, an additional transform with size-16 leaves and streaming store instructions was also generated. Before running the benchmark, the library was calibrated and the fastest configuration selected (details of the calibration are described in section 5.4.1.3).

For most sizes of transform, precision and vector length, SFFT is faster than FFTW running in patient mode. For the transforms with memory



(a) Single-precision, SSE (VL-2)

(b) Double-precision, SSE (VL-1)

(c) Single-precision, AVX (VL-4)

(d) Double-precision, AVX (VL-2)

Figure 14: Performance of hard-coded leaf FFTs on a Macbook Air 4,2.

requirements that are approximately at the limits of the cache, FFTW running in patient mode is sometimes marginally faster than SFFT. Once the transforms exceed the size of the cache, SFFT is again the fastest.

It is important to note that FFTW running in patient mode evaluates a huge configuration space of parameters (and thus takes a long time to calibrate), while SFFT has, in this case, only evaluated either three or four configurations per transform.

5.4 IN PRACTICE

SFFT is not itself an FFT library; the name refers to the elaboration program that reads a configuration file and generates the code for an FFT library. The code for the FFT library is then built as any other library would be.

5.4.1 *Organization*

As well as the generated code, there is infrastructure code which is common to all libraries generated by SFFT. This can be broadly categorized into three parts: initialization, dispatch and calibration.

5.4.1.1 *Initialization*

Before an application can compute an FFT with SFFT, it must initialize a plan for the specific size, precision and direction of FFT. The library may have several FFTs and configurations that can compute the requested FFT, and it chooses the fastest option by timing each of the candidate configurations, which is at most 8 for any size of transform – a very small space compared to FFTW's exhaustive search of all possible FFT algorithms and configurations. Chapter 7 describes an alternative to calibration, where machine learning is used with data collected from benchmarks to build a model that predicts performance.

After determining which implementation and parameters will be used, the initialization code allocates memory and populates any lookup tables that may be required. Before returning the plan to the application, a function pointer in the plan is updated to point to the FFT that has just been initialized.

5.4.1.2 *Dispatch*

Applications do not invoke any of the FFTs within SFFT directly. Rather they invoke a dispatch function on an initialized plan, which in turn transfers control to the correct FFT code within SFFT. The use of a dispatch function is purely a matter of convenience, so that users only need to deal with a few simple functions.

5.4.1.3 *Calibration*

SFFT contains calibration code to measure the performance of the possible configurations of FFT on the target machine, which is at most 8 for each size of transform. Following calibration, the timing data is written to a file, which is then used by SFFT to select the fastest possible FFT for a given problem running on that machine.

5.4.2 *Usage*

SFFT is used much like other FFT libraries:

1. A plan for an FFT is initialized;
2. Using the plan, an FFT is computed (this step may be repeated many times);
3. The plan is destroyed.

The plan is initialized for a given size, precision and direction of transform, and may then be executed any number of times on any data. Any number of plans can be simultaneously created and used.

In Listing 19, a size-1024 transform is computed on double-precision data with AVX enabled. In lines 2-4, the input and output arrays are

Listing 19: SFFT example usage

```
 1  int n = 1024;
 2  double complex __attribute__ ((aligned(32))) *input, *output;
 3  input = _mm_malloc(n * sizeof(double complex), 32);
 4  output = _mm_malloc(n * sizeof(double complex), 32);
 5
 6  for(i=0;i<n;i++) input[i] = i;
 7
 8  sfft_plan_t *p = sfft_init(i, SFFT_FORWARD|SFFT_DOUBLE|SFFT_AVX);
 9
10  if(p) {
11
12    sfft_execute(p, input, output);
13    for(i=0;i<n;i++)
14      printf("%d %f %f\n", i, creal(output[i]), cimag(output[i]));
15    sfft_free(p);
16
17  }else{
18    printf("Plan unsupported\n");
19  }
```

allocated with 32 byte alignment, as is required for aligned AVX memory operations. The plan is initialized at line 8, used to compute an FFT at line 12 (provided the requested plan is supported), and finally freed at line 20.

5.4.3 *Other optimizations*

In addition to generating a general-purpose library that can be calibrated for a machine and application at runtime, there are several situations where the SFFT library can be specially optimized:

1. If the machine and application are fixed, a one time calibration can be performed and an optimized library containing only the fastest transforms specific to the application and machine is generated;

2. If the application is fixed, an optimized library containing only the transforms specific to the application is generated (and the library is calibrated the first time it is used on each machine);

3. If the machine is fixed, an optimized library containing only the transforms specific to the machine is generated (and an application can use any transform without calibration).

6 BENCHMARK METHODS

"If one takes care of the means, the end will take care of itself."

— Gandhi

This chapter describes the benchmarking methods used to evaluate the performance and accuracy of various FFT implementations throughout this thesis.

The two architectures of interest are the Intel x86 architecture and the ARM architecture. A comprehensive set of results collected from a wide range of machines implementing these architectures is presented in Chapter 7, but throughout the rest of the thesis, benchmarks are performed on an Apple Macbook Air 4,2; a widely available and currently state-of-the-art machine that is equipped with an Intel Core i5-2557M. Table 8 summarizes the specifications of the machine.

For the x86 benchmarks, an existing framework called *BenchFFT* [1] was used. For the ARM benchmarks, which were performed on iOS devices, there was no existing FFT benchmark software, and so an application was written for this purpose, which is described in Section 6.2.

6.1 x86 architecture

The x86 benchmarks were performed with BenchFFT, a collection of FFT libraries and benchmarking software assembled by Frigo and Johnson,

	MACBOOK AIR 4,2
CPU	Dual-core Intel Core i5 (i5-2557M)
CPU clock	1.7 GHz (turbo to 2.7GHz with one core)
L1 cache	32KB I-cache & 32KB D-cache
L2 cache	256KB
L3 cache	3MB shared
Memory	4 GB of 1333 MHz DDR3 SDRAM
OS	OS X 10.7.2
SIMD extensions	SSE and AVX

Table 8: Specifications of the primary test machine

the authors of FFTW [1]. The benchmarks in BenchFFT use timing and calibration code from `lmbench`, a performance analysis tool written by Larry McVoy and Carl Staelin [59].

6.1.1 *Timing*

BenchFFT measures the initialization time and runtime of an FFT separately. The initialization time is measured only once, and thus outliers due to effects from external factors such as OS scheduling are occasionally observed. Routines from `lmbench` are then used to calibrate the minimum number of FFT iterations required for accurate measurement using the `gettimeofday` function. Finally, the time taken to run the minimum number of iterations is measured eight times, from which the minimum time divided by the number of iterations is used, in order to factor out effects from external factors.

The minimum time for a transform is then used to determine a scaled inverse time measurement, sometimes known as CTGs. CTGs are defined as:

$$\mathrm{CTGs} = \frac{5N\log_2(N)}{10^9 t} \tag{39}$$

for complex transforms and

$$\mathrm{CTGs} = \frac{2.5N\log_2(N)}{10^9 t} \tag{40}$$

for real transforms, where t is the time taken to run one transform (in seconds). Unless the Cooley-Tukey radix-2 algorithm is used, a measurement expressed in CTGs is not an actual FLOP count – it is a rough measure of an algorithm's efficiency relative to the radix-2 algorithm and the clock speed of the machine.

When a transform has several variants (such as direction or radix), BenchFFT reports the speed of the FFT as being the *fastest* of the possible options.

6.1.2 *Accuracy*

To measure the accuracy of a transform, BenchFFT compares an FFT with an arbitrary-precision FFT computed on the same inputs, and reports the relative root-mean-square (RMS) error. The inputs are pseudo-random in the range $[0.5, 0.5)$ and the arbitrary-precision FFT has over 40 decimal places of accuracy.

When a transform has several variants (such as direction or radix), BenchFFT reports the accuracy as being *worst* of the results.

6.1.3 *Compiling*

Except where otherwise noted, Intel C compiler (ICC) version 12.1.0 for OS X was used to compile 64-bit code. For OS X builds, the compiler flags used were "-O3", while "-O3 -msse2" (or equivalent) was used for

Linux builds. In the cases where the FFT uses AVX, the code is compiled with “-xAVX” or “-mavx” (depending on compiler).

Some libraries included in the BenchFFT software have their own compilation scripts which override the defaults, and in the case of commercial libraries (such as Intel IPP and Apple vDSP), the compiler flags are of little consequence because the libraries are distributed in binary form.

6.1.4 *Data format*

FFT libraries use interleaved format and/or complex format to store the data. In the case of interleaved format, the real and imaginary parts of complex numbers are stored adjacently in memory, while in the case of split format, the real and imaginary parts are stored in separate arrays.

The majority of FFT libraries use interleaved format to store data. In the case where the library supports interleaved *or* split format, BenchFFT uses interleaved format. However there are a few libraries that only support split format, and in theses cases it should be noted the results are not strictly comparable (Apple vDSP is one such case).

6.2 ARM ARCHITECTURE

There was no existing FFT benchmarking software for iOS on ARM devices, and so a benchmarking tool was written. The tool runs the benchmarking in a thread of normal priority.

6.2.1 *Compiling*

The code was compiled with Apple clang compiler 3.0 for ARMv7 targets running iOS 5.0. The compiler flags used were "-O3 -mfpu=neon".

6.2.2 *Timing*

The Apple A4 and A5 system on chips (SoCs) are built around the ARM Cortex-A8 and Cortex-A9 cores, which have hardware cycle counters that can be used for precise timing. The cycle counter control registers can only be accessed in kernel mode, and so the high resolution timer available through the `mach_absolute_time` function was used instead.

For a given size of transform, a calibration routine determines the number of iterations that must be run such that the total runtime is approximately one second. After calibration, each FFT to be evaluated is run for the pre-determined number of iterations – this loop is run eight times, and the fastest time divided by the number of iterations is taken to be the FFTs runtime. By running each FFT for approximately one second, and repeating the measurement eight times to find the best time, the effects from external factors such as OS scheduling are minimized. As with BenchFFT, the time is expressed in CTGs.

7 RESULTS AND DISCUSSION

"There is no data that can be displayed in a pie chart, that cannot be displayed BETTER in some other type of chart."

— John Tukey

In order to test the hypotheses set out in Chapter 1, SFFT was benchmarked alongside FFTW and other libraries on a wide range of machines, as per the methods set out in Chapter 6. The majority of the data was collected on Linux machines populated with SSE capable Intel microprocessors, with some additional data collected on small set of AVX and ARM NEON machines. The results are divided into three sections: speed, accuracy and setup time, with an additional section detailing a model that predicts SFFT's performance for different configurations. Finally, the chapter concludes by relating the results to other work.

Table 9 presents a summary of the Linux machines that were used to run benchmarks. The majority of the machines were functioning as either lab workstations or servers in a University environment. The benchmarks took approximately 12 hours to run, and while efforts were made to reduce each machine's load to a minimum, there were still transient system processes, such as log rotations and backups during the night that have introduced noise into the results.

For the Linux benchmarks, both 32-bit and 64-bit statically-linked binaries for SFFT, FFTW 3.3 and SPIRAL were compiled with icc 12.0.5, gcc 4.4.5 and clang 1.1. For the OS X benchmarks, 32-bit and 64-bit binaries for SFFT, FFTW 3.3 and SPIRAL were compiled with icc 12.1.0,

MODELSTRING	L1D	L2	L3
Intel(R) Pentium(R) 4 CPU 2.80GHz	16	512	-
Intel(R) Pentium(R) D CPU 3.00GHz	16	1024	-
Intel(R) Pentium(R) M processor 1000MHz	32	1024	-
Intel(R) Xeon(TM) CPU 2.40GHz	16	2048	-
Intel(R) Xeon(R) CPU E5335 @ 2.00GHz	32	4096	-
Intel(R) Xeon(R) CPU X5355 @ 2.66GHz	32	8192	-
Intel(R) Xeon(R) CPU E5430 @ 2.66GHz	32	6144	-
Intel(R) Xeon(R) CPU X5560 @ 2.80GHz	32	256	8192
Intel(R) Core(TM)2 CPU 6600 @ 2.40GHz	32	4096	-
Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz	32	4096	-
Intel(R) Core(TM)2 Duo CPU E6850 @ 3.00GHz	32	4096	-
Intel(R) Core(TM)2 Duo CPU E8400 @ 3.00GHz	32	6144	-
Intel(R) Core(TM)2 Duo CPU P8600 @ 2.40GHz	32	3072	-
Intel(R) Core(TM) i5 CPU 660 @ 3.33GHz	32	256	4096
Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz	32	256	8192

Table 9: Linux benchmark machines, listed with the size of each level of cache (in kilobytes)

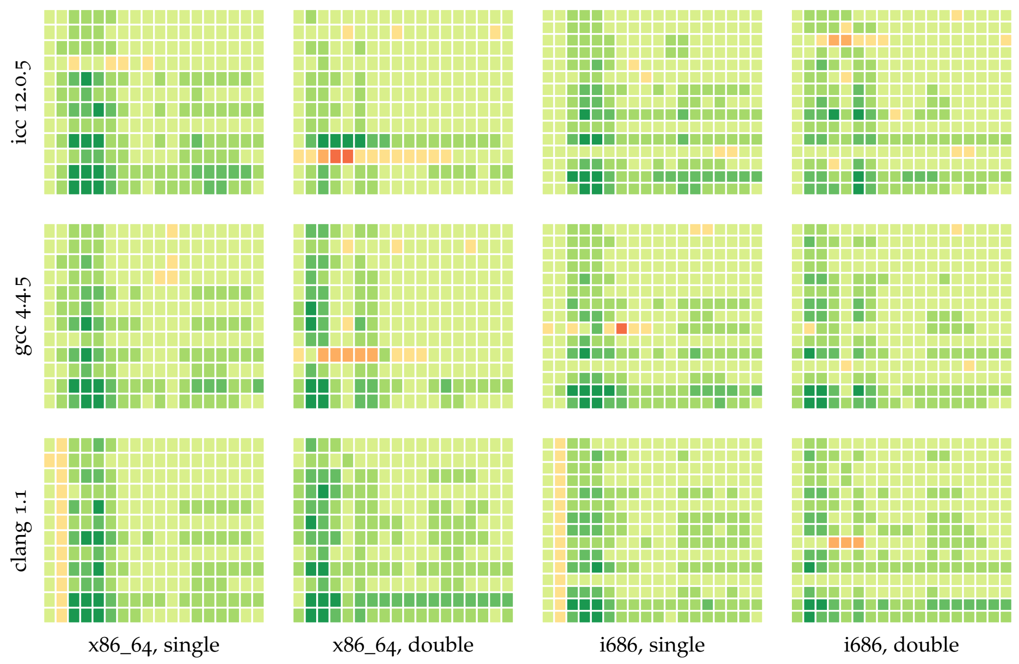


Figure 15: Performance comparison between SFFT and FFTW 3.3 in estimate mode on SSE machines

llvm-gcc 4.2.1 and clang 3.0. The builds of SFFT and FFTW 3.3.1 for iOS 5 on ARM NEON were compiled with Apple clang 3.0.

Several binary libraries were also benchmarked: Intel IPP 7 and Apple Accelerate. Because these libraries are only available in binary form, they are compared against the icc builds of SFFT, FFTW 3.3 and SPIRAL, because icc generally produced the fastest code.

7.1 SPEED

The speed results are presented in subsections according to the SIMD extensions: SSE, AVX and ARM NEON.

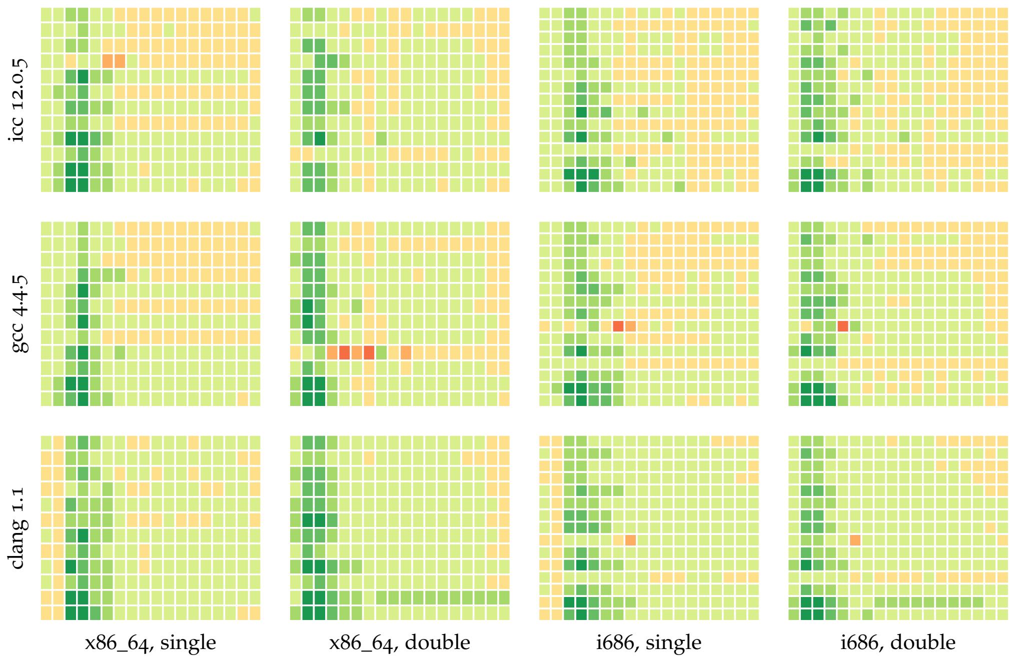


Figure 16: Performance comparison between SFFT and FFTW 3.3 in patient mode on SSE machines

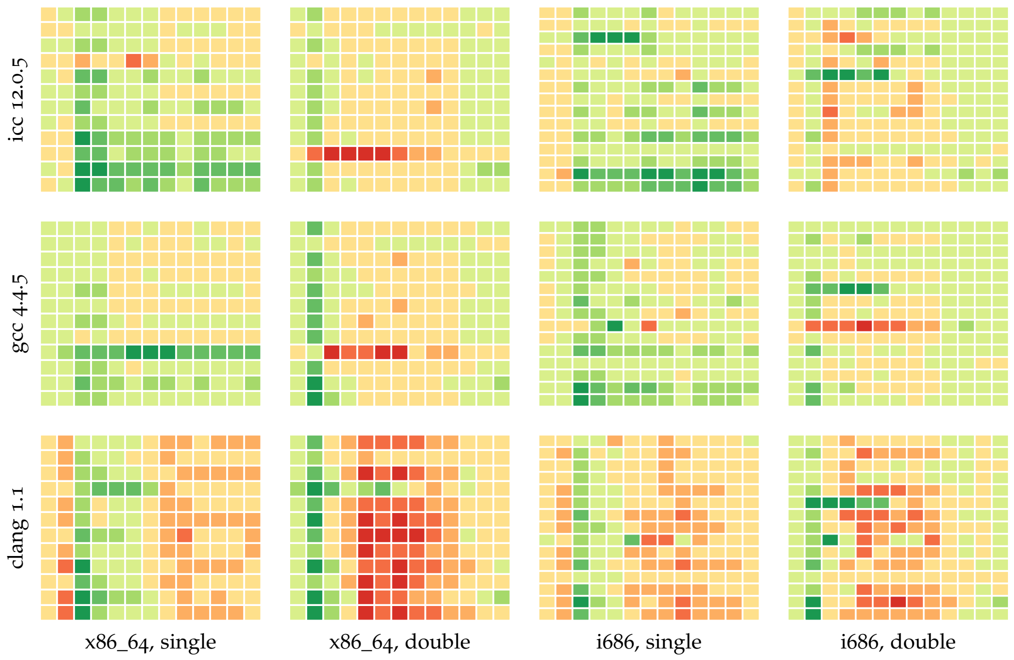


Figure 17: Performance comparison between SFFT and SPIRAL on SSE machines. Although SPIRAL is faster when compiled with clang 1.1, Figure 19 shows that SFFT is faster than SPIRAL when compiled with clang 3.0

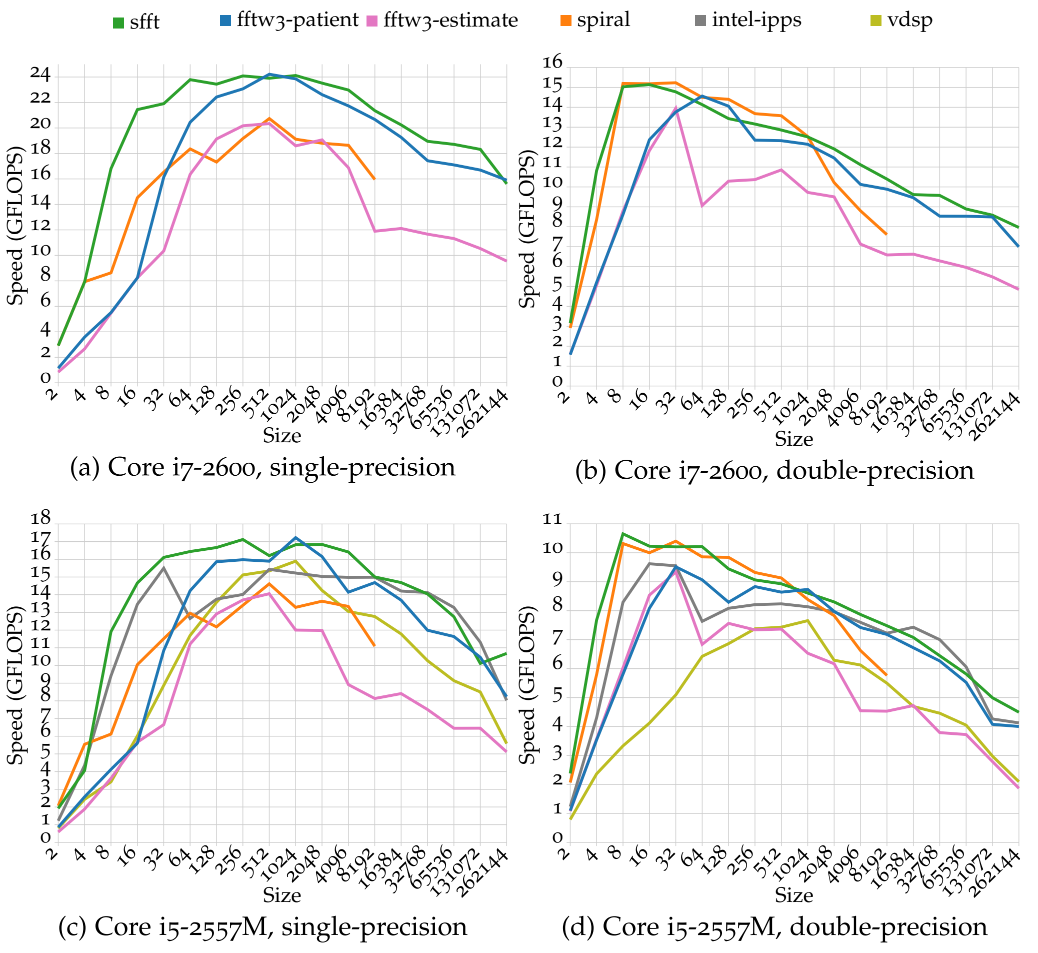
7.1.1 *SSE*

Figure 15 summarizes the speed performance of SFFT against FFTW 3.3 running in estimate mode on Linux machines with SSE. Twelve heatmaps are used to present data from different configurations. The three rows in the grid correspond to the three different compilers used, while the four columns correspond to the four different architecture and floating-point precision pairs. Within each heatmap, the rows correspond to different machines, and the columns correspond to different sizes of transform (2^1 through to 2^{18}). Shades of green indicate that SFFT is faster for a particular point of data, while shades of yellow through to red indicate that FFTW is faster; lighter shades indicate a small difference, while darker shades indicate a bigger difference in performance. The scale for the colour map is computed separately for each of the 12 heatmaps in the grid, so a particular colour in one heatmap is not directly comparable to the same colour in another heatmap; the colours are only meant to indicate differences within each heatmap.

Similarily, Figure 16 compares SFFT to FFTW 3.3 running in patient mode, and Figure 17 compares SFFT to SPIRAL. There are fewer columns in the heatmaps of Figure 17 because SPIRAL only computes single-threaded FFTs for sizes 2^1 through to 2^{13}.

7.1.1.1 *FFTW 3.3 in estimate mode*

Figure 15 shows that SFFT is faster than FFTW 3.3 running in estimate mode in almost all cases over a range of Intel x86 machines that implement SSE. The horizontal streaks of yellow-red that can be seen in some heatmaps are outliers and likely caused by transient system processes



(a) Core i7-2600, single-precision (b) Core i7-2600, double-precision

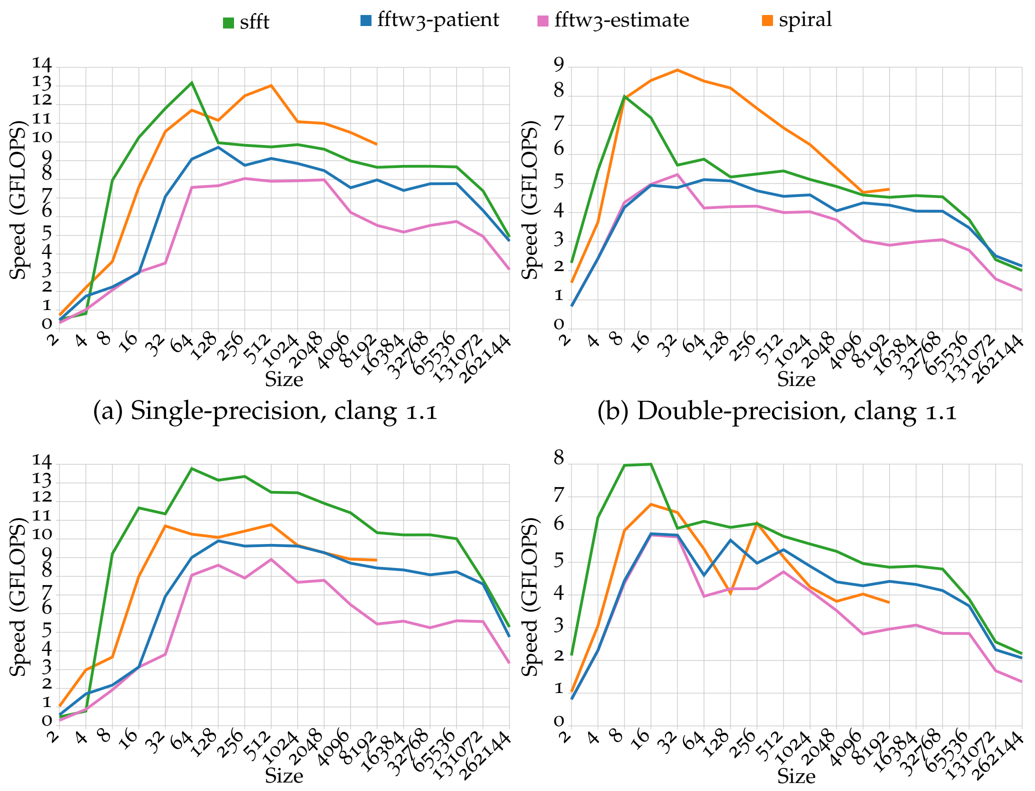
(c) Core i5-2557M, single-precision (d) Core i5-2557M, double-precision

Figure 18: Performance of FFTs on recent Sandy Bridge machines, with x86_64 SSE binaries. Compiler: icc

that were running while SFFT was being benchmarked. Similar streaks appear at the same locations in Figures 16 and 17.

7.1.1.2 *FFTW 3.3 in patient mode*

Figure 16 shows that SFFT is faster than FFTW 3.3 running in patient mode in the majority of cases over a range of Intel x86 machines that implement SSE. SFFT was generally slightly slower than fftw3-patient on older machines such as the Pentium 4's and the 1GHz Pentium M, while on the newer machines such as the Sandy Bridge based Core i7-2600 and the Nehalem based Core i5-660, SFFT was clearly faster than



(a) Single-precision, clang 1.1

(b) Double-precision, clang 1.1

(c) Single-precision, clang 3.0

(d) Double-precision, clang 3.0

Figure 19: Performance of clang-compiled x86_64 SSE FFTs on an Intel Core2 Duo P8600

FFTW (see Figure 18). This could be explained by the fact that FFTW performs extensive instruction level optimizations, such as scheduling, and that the older processors have smaller instruction and trace caches.

7.1.1.3 *SPIRAL*

The last row of Figure 17 shows that SFFT is generally slower than SPIRAL when both libraries are compiled with clang 1.1. However, with more recent releases of clang, which do much more code optimization, the situation is reversed, as shown in Figure 19. In some cases SPIRAL compiled with clang 3.0 is slower than SPIRAL compiled with clang 1.1, while SFFT is generally faster when compiled with clang 3.0. This demonstrates that the speed of automatically tuned SPIRAL code is specific to certain compilers.

SPIRAL's double-precision performance is slightly better than SFFT when compiled with icc or gcc, while SFFT's single-precision code is faster than SPIRAL on recent machines, and of similar speed on older machines.

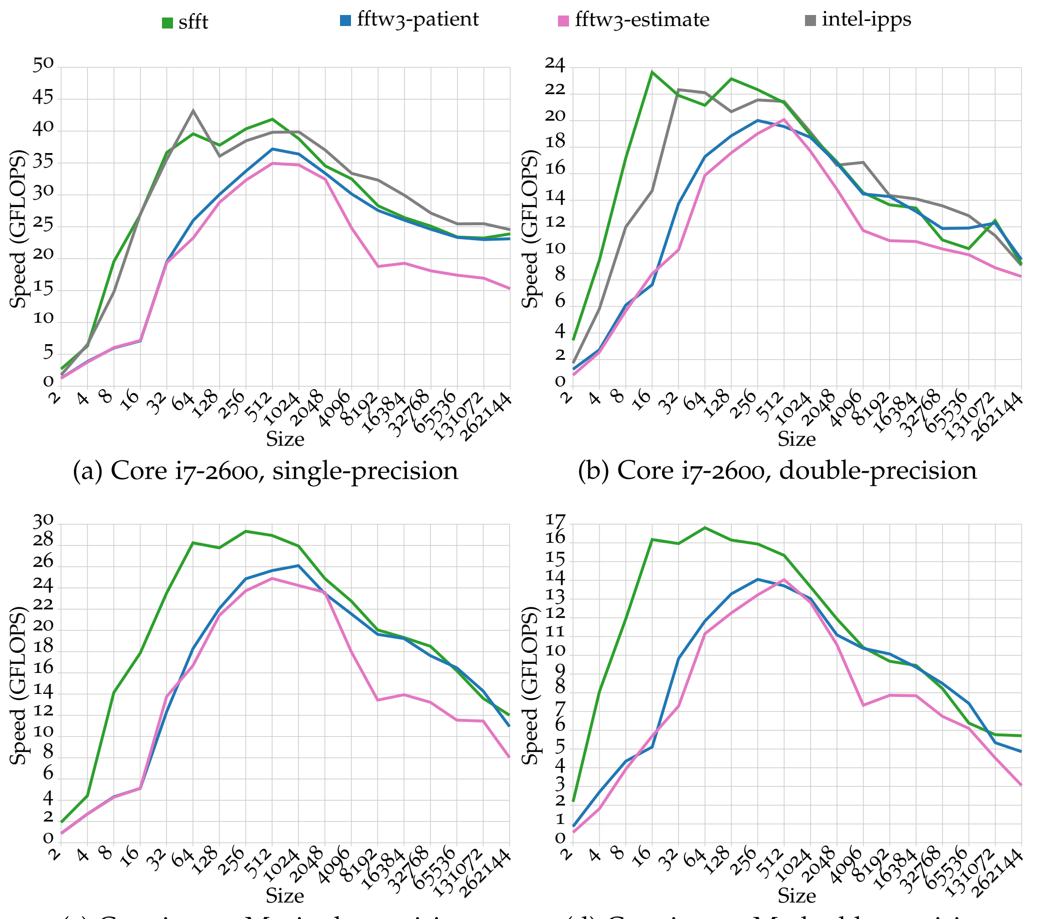
7.1.2 *AVX*

Of the machines that were used for benchmarks, only two supported AVX: the Macbook Air 4,2 with an Intel Core i5-2557M, and a Linux machine with an Intel Core i7-2600. Figure 20 shows that SFFT is clearly faster than FFTW up until about 1024 points, while performance between the two is similar for larger transforms.

Results for Intel IPP are also plotted in Figure 20, but only for the Core i7-2600. IPP did not detect the existence of AVX on the Core i5-2557M, and instead used SSE, as plotted in Figure 18. Apple vDSP does not support AVX, and so SSE vDSP results for the Macbook Air 4,2's Core i5-2557M are also plotted in Figure 18.

7.1.3 *ARM NEON*

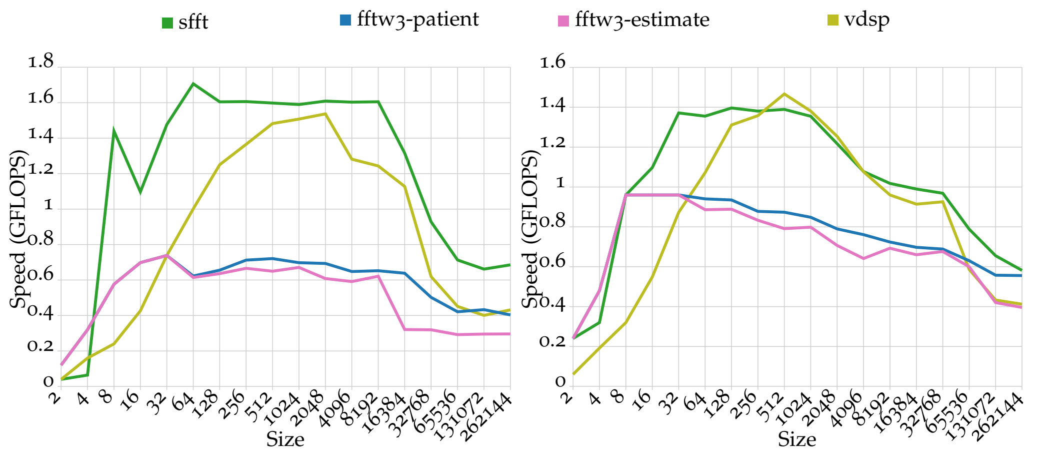
SFFT and FFTW 3.3.1 were compiled with Apple clang 3.0 and benchmarked on an Apple iPod touch 4G and an Apple iPad 2, which contain the Apple A4 and A5 SoCs respectively. The A4 implements the ARM Cortex-A8, while the A5 implements the ARM Cortex-A9, both of which support ARM NEON.



(a) Core i7-2600, single-precision (b) Core i7-2600, double-precision

(c) Core i5-2557M, single-precision (d) Core i5-2557M, double-precision

Figure 20: Performance of FFTs on recent Sandy Bridge machines, with x86_64 AVX binaries. Compiler: icc



(a) Apple A4 (ARM Cortex-A8) (b) Apple A5 (ARM Cortex-A9)

Figure 21: Performance of single-precision FFTs on ARM NEON devices running iOS. Compiler: Apple clang 3.0

Figure 21 shows that SFFT is easily faster than FFTW on both devices. This contradicts Frigo and Johnson's claim that the performance of FFTW is portable, and tends to support the idea that it is possible to write fast and portable code without exhaustive searches through the configuration space of all possible FFTs.

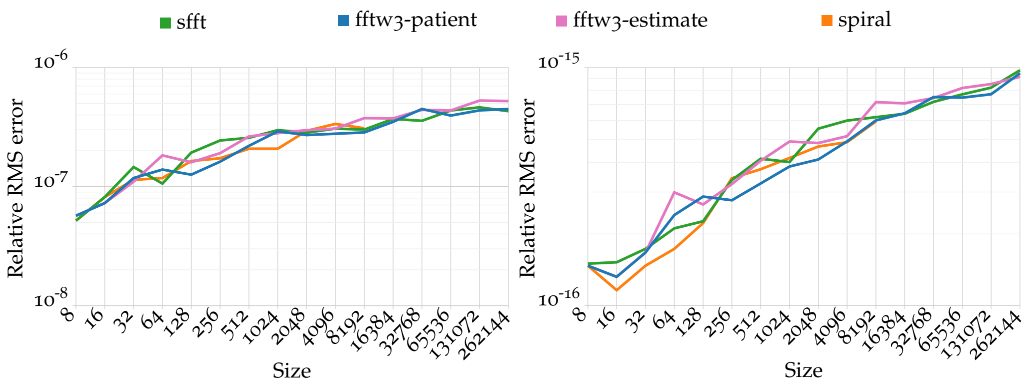
A considerable amount of effort was needed to work around several problems that were encountered when targeting ARM NEON with Apple clang 3.0, and many of SFFT's primitive macros for NEON were written in inline assembly code. Among the problems encountered when targeting ARM NEON with Apple clang 3.0:

1. There is no way of explicitly specifying memory alignment when using vector intrinsics;
2. Fused multiply-add/subtract intrinsics do not currently compile to the correct instructions because of a bug in clang;
3. Clang's inline assembly front-end lacks the syntax and semantics to properly address the dual-size aliased vector registers.

The above problems affect all FFT libraries equally, and it seems that portability depends critically on the quality of the machine specific code and macros.

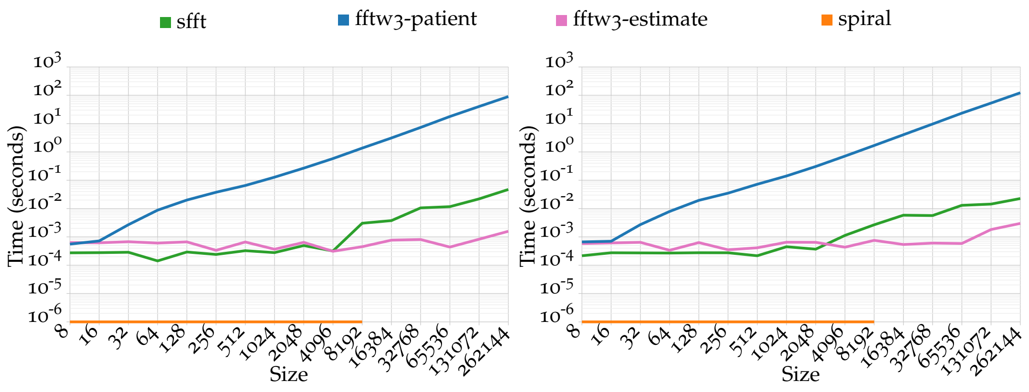
7.2 ACCURACY

The accuracy of each FFT was measured as per the methods in Chapter 6. The accuracy of single and double precision FFTs on an Intel Core i7-2600 is plotted in Figure 22, and shows that the relative RMS error for FFTW, SFFT and SPIRAL is within an acceptable range. Graphs for all other machines are similar.



(a) SSE, single-precision (b) SSE, double-precision

Figure 22: Accuracy of FFTs on an Intel Core i7-2600. SFFT, FFTW and SPIRAL were compiled for x86_64 with icc



(a) SSE, single-precision (b) SSE, double-precision

Figure 23: Setup times of FFTs on an Intel Core i7-2600. SFFT, FFTW and SPIRAL were compiled for x86_64 with icc

7.3 setup time

Figure 23 shows that FFTW, in patient mode, requires several orders of magnitude more time to initialize as it searches for a fast FFT configuration. SPIRAL has a very fast setup time, because it is entirely statically elaborated and needs no dynamic initialization. The setup time for SFFT is comparable to FFTW in estimate mode, though SFFT's setup time begins to increase for transforms larger than 8192 points. This is likely because of repeated calls to the complex exponential function as twiddle factor LUTs are elaborated; no effort was made to optimize this setup code, and it is likely that it would be much faster if the calls to the complex exponential function were optimized.

Graphs for all other machines are similar.

7.4 BINARY SIZE

Compared to other libraries, SFFT produced larger binaries for the benchmarks, because there is currently no optimization performed between transforms contained in the same library. For 64-bit single precision binaries on OS X with AVX, the size of the SFFT benchmark was approximately 2.8 megabytes while the size of the FFTW benchmark was 1.8 megabytes.

7.5 PREDICTING PERFORMANCE

For each size of transform on a particular machine, SFFT chooses the fastest configuration from a set of up to eight possible configurations. Small transforms have only one option, which is a fully hard-coded transform, while larger transforms have up to eight, which could include the four-step transform, and several variants of the hard-coded leaf transform, where each variant corresponds to a particular size of leaf sub-transform and size of body sub-transform, and for size-16 leaf sub-transforms, a streaming store variant is included too. The decision of exactly which configuration to use depends on the size of transform, the compiler, and the characteristics of the host machine.

For the benchmarks in this chapter, SFFT used a calibration routine to choose the fastest configuration. The calibration data was collected, along with some data about the machine and the compiler, and used to train a classifier.

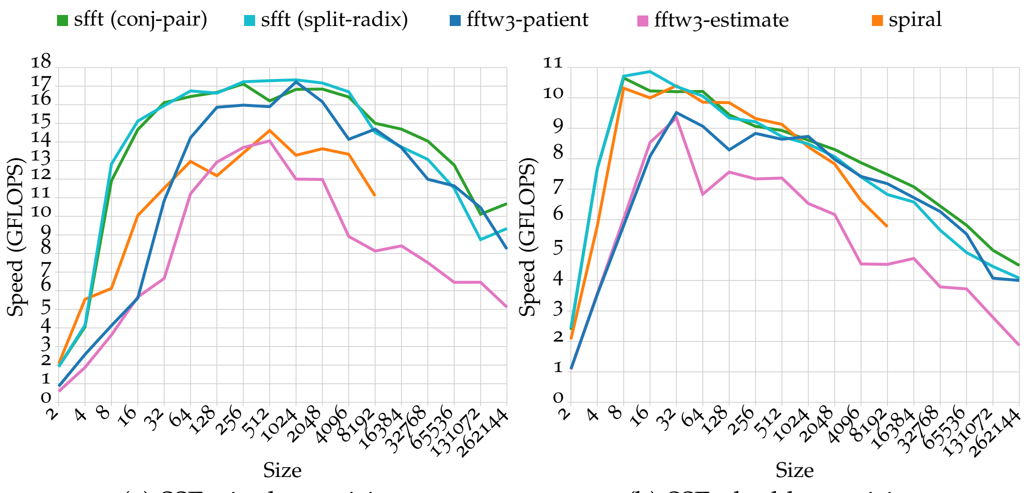
The data was processed into instances, with each instance having attributes for the size of the transform and the precision, the size of each level of cache, the architecture and micro-architecture of the machine, the SIMD extensions, the OS, the compiler used, and the CPU frequency. In total there were 3348 instances of data, each of which had 12 attributes.

Weka [77] was used to experiment with several classifiers, and a REPTree classifier with bagging was used to train a model. Using 10-fold cross-validation, the model correctly classified 76.1% of the instances with a weighted average precision of 74.8%, which tends to confirm the existence of a relationship between the characteristics of the machine and the performance of a particular FFT configuration.

The accuracy of the classifier is promising, and it has the potential to replace the calibration code in SFFT. It is highly likely that if the noise in the data was reduced through the use of an isolated benchmarking environment, the accuracy of the classifier would increase. The accuracy would also likely benefit from a larger dataset collected from a larger range of benchmark machines.

7.6 SPLIT-RADIX VS. CONJUGATE-PAIR

In order to quantify the gain in performance that might be attributable to the use of the conjugate-pair algorithm, SFFT was retrospectively modified to compute the FFT using the ordinary split-radix algorithm as well as the conjugate-pair algorithm. The results of benchmarks between the two algorithms, as well as FFTW and SPIRAL, are plotted in Figure 24.



(a) SSE, single-precision (b) SSE, double-precision

Figure 24: Ordinary split-radix versus conjugate-pair split-radix on an Intel Core i5-2557M. SFFT, FFTW and SPIRAL were compiled for x86_64 with icc

Unexpectedly, the ordinary split-radix algorithm is faster than the conjugate-pair algorithm for some smaller sizes of transform, but for transforms above a certain size, the conjugate-pair algorithm is faster by a few hundred MFLOPS.

The performance advantage of the ordinary split-radix algorithm for smaller sizes of transforms is likely due to shorter chains of dependent instructions where twiddle factors are loaded and used. Consider that the ordinary split-radix algorithm separately loads two twiddle factors into two registers, and there are no dependencies between these instructions, while the conjugate-pair algorithm must load one twiddle factor and then duplicate it into another register, which does result in dependent instructions. Thus the ordinary split-radix algorithm is faster for smaller transforms where memory bandwidth is not the limiting factor, but when memory bandwidth does become the limiting factor, the conjugate-pair algorithm is faster.

In future, SFFT could exploit the performance advantage of the ordinary split-radix algorithm when computing smaller sizes of transforms.

7.7 APPLICATIONS OF THIS WORK

This section provides an overview of how the techniques presented in this thesis may be applied to the prime-factor algorithm, sparse Fourier transforms, and multi-threaded transforms.

7.7.1 *Prime-factor algorithm*

The techniques presented in this work rely on the fact that FFTs operating on signal lengths that are a power-of-two can be factored into smaller power-of-two length components, which are computed in parallel by being evenly divided into a number of SIMD vector registers that are a power-of-two length.

The prime-factor algorithm factors other lengths of FFTs into components that are co-prime in length, and ultimately small prime components, which do not evenly divide into the power-of-two length SIMD registers, except in the special case where a SIMD register contains only one complex element (such is the case with double-precision on SSE machines).

Because the prime components do not evenly divide into power-of-two length SIMD registers, the algorithm level vectorization techniques presented in this work are not directly applicable. In contrast, the auto-vectorization techniques used in SPIRAL [29, 51, 52] are performed at the instruction level, and are applicable to the prime-factor algorithm, but as the results in Figure 18 show, the downside of SPIRAL's lower level approach is that performance for power-of-two transforms scales poorly with the length of the SIMD register.

7.7.2 *Sparse Fourier transforms*

The recently published Sparse FFT [39, 38] will benefit from the techniques presented in this work because the inner loops use small DFTs (e.g, 512 point for a certain 256k point sparse FFT), which are currently computed with FFTW. Replacing FFTW with SFFT will almost certainly result in improved performance, because SFFT is faster than both FFTW and Intel IPP for the applicable small sizes of transform on an Intel Core i7-2600 (see Figure 20).

Version 2.0 of the Sparse FFT code is scalar, and would benefit greatly from explicitly describing the computation with SIMD intrinsics. However, a key difference between the sparse Fourier transform and other FFTs is the use of conditional branches on the input signal data. This has performance implications on all machines, but it is worth noting that some machines will be drastically affected by this, such as the ARM Cortex-A8, where the SIMD pipeline is located behind the main pipeline, resulting in fast transfers from the main CPU unit to the SIMD pipeline, but large penalties when SIMD registers or flags are accessed by the main CPU unit.

7.7.3 *Multi-threaded transforms*

MatrixFFT has recently shown that the four-step algorithm [8], designed to efficiently use hierarchical or external memory on Cray machines in the 1980's, is useful for computing large multi-threaded transforms on modern machines, providing performance far surpassing that of FFTW's multi-threaded performance [69].

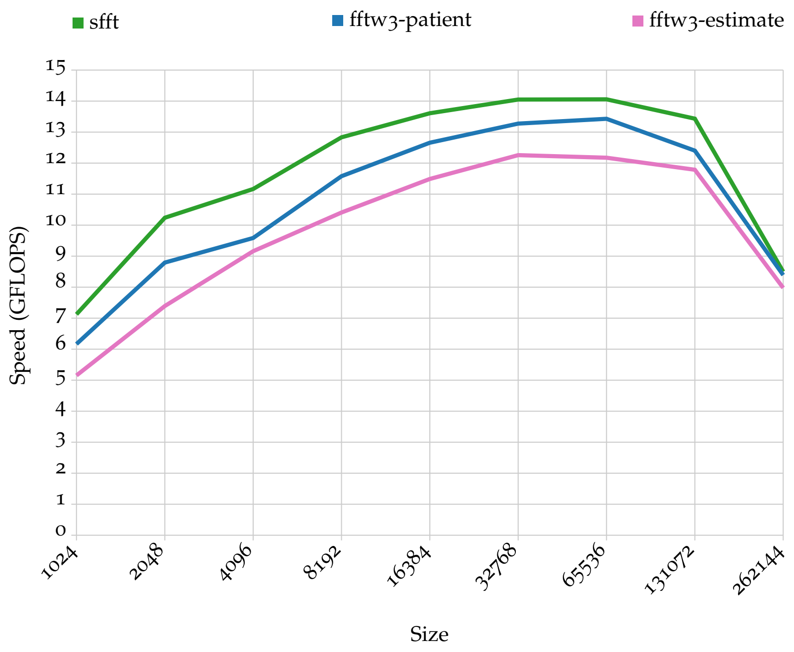


Figure 25: Speed of multi-threaded four-step algorithm running on an Intel Core i5-2557M with four threads. The algorithm decomposes transforms into smaller single-threaded components, which are computed above with three different implementations. All code was compiled with icc for x86_64 with SSE.

The four-step algorithm decomposes a transform of size N into a two-dimensional array of size $n_1 \times n_2$ where $N = n_1 n_2$, and $n_1 = n_2 = \sqrt{N}$ (or close) often obtains the best performance.

The four-steps of the algorithm are:

1. Compute n_1 FFTs of length n_2 along the columns of the array;
2. Multiply each element of the array with ω_N^{ij}, where i and j are the array coordinates;
3. Transpose the array;
4. Compute n_2 FFTs of length n_1 along the columns of the array.

Each step can be divided amongst a pool of threads, with a synchronisation barrier between the third and fourth steps. The transforms in steps one and four operate on sequential data, and if they are small enough, they are not subject to bandwidth limitations (and if they are not small enough, they can be further decomposed with the four-step algorithm until they are small enough). The bandwidth bottleneck does not disappear, but it is factored out into the transpose in step three, and because of this, the performance of the small single-threaded 1D transforms used in steps one and four correlate with the overall multi-threaded performance. A simple multi-threaded implementation of the four-step algorithm was benchmarked with SFFT and FFTW transforms, and the results are shown in Figure 25, which tends to confirm that the performance of single-threaded transforms for steps one and four translates to the overall multi-threaded performance when using the four-step algorithm.

7.8 SIMILAR WORK

Aside from Bernstein's FFT library, which was designed in the days of scalar microprocessors and has not been updated since 1999, there have been a few other challenges to the automatically adaptive approach of FFTW, but none present concrete results that definitively dismiss the idea. Most recently, Vasilios et al. presented an approach that uses the characteristics of the host machine to choose good FFT parameters at run time [49], but their approach has several issues that render it almost irrelevant. First, the approach uses optimizations that only apply to scalar machines, viz. twiddle factor symmetries are exploited to compress the twiddle LUTs, and arithmetic is avoided when twiddle factors contains zeros or ones. The vast majority of microprocessors, even those found in mobile devices such as phones, feature SIMD extensions, and so an approach that is limited to scalar arithmetic is of little consequence.

Second, they benchmark the FFTs in a most unusual way. Rather than repeat a large number of iterations of the FFT, they repeat a large number of iterations of a binary that initializes and then executes only one FFT; such an approach is by no means representative of applications where the performance of the FFT is a concern, and is more a measurement of the initialization time rather than the FFT.

8 CONCLUSIONS AND FUTURE WORK

"... programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself, but where to find it."

— John W. Backus [7, 19]

The results presented in this thesis show that vectorization at the algorithm level of abstraction produces good performance results, the conjugate-pair algorithm is in many cases faster than the ordinary split-radix algorithm, and that there are good heuristics for predicting the performance of the FFT on SIMD microprocessors (i.e., the need for empirical optimization may be overstated).

This work concludes with a review of the hypotheses, a summary of the contributions, some ideas for directions that future work might take, and a few final remarks.

8.1 REVISITING THE HYPOTHESES

This section discusses the hypotheses of Section 1.1 with reference to the experiments in Chapters 3 and 5 and the results in Chapter 7.

Hypothesis 1: Accessing memory in sequential "streams" is critical for best performance

The simple implementation in Section 3.2 used a LUT to store precomputed coefficients, but for every size of sub-transform that composes a particular transform, the LUT is accessed non-contiguously, with vector gather operations of varying strides. In Section 3.3, vector intrinsics and a sequentially accessed LUT for each size of sub-transform are shown to improve performance. Although the set of LUTs increases the memory footprint, the speed improves markedly, by over 30% in many cases.

In Section 5.3.2, a DAG representing the computation was topologically sorted so that accesses to the input data, which are effectively pseudo-random for a decimation-in-time decomposition, are ordered into sequential streams. Benchmark results in Figure 14 show that this technique, in tandem with several others, achieves good results, being faster than FFTW in many cases.

The results from the above two cases confirm the idea that accessing data in sequential streams provides big performance gains, even in the somewhat counter-intuitive case where data is duplicated and more memory is required.

Hypothesis 2: The conjugate-pair algorithm is faster than the ordinary split-radix algorithm

Hypothesis 2 is based on the idea that memory bandwidth is a bottleneck, and on the fact that the conjugate-pair algorithm requires only half the number of twiddle factor loads.

In Section 7.6, a highly optimized implementation of the conjugate-pair algorithm is benchmarked against an equally highly optimized implementation of the ordinary split-radix algorithm. For smaller sizes of transform, the ordinary split-radix algorithm is faster, but above a certain size (4096 in this case), the conjugate-pair algorithm is faster.

Thus, Hypothesis 2 is confirmed with the proviso that the transform is larger than a particular size.

Hypothesis 3: The performance of an FFT can be predicted based on characteristics of the underlying machine and the compiler

In Chapter 7, SFFT and FFTW were benchmarked on sixteen x86 machines and two ARM NEON machines, and SFFT was found to be as fast as, or faster than FFTW, suggesting that the performance of an FFT running on a certain machine can be predicted and reasoned about, and that extensive machine calibration might not be required.

In Section 7.5, a model was evaluated with 10-fold cross-validation to have 74.8% precision when using characteristics of the underlying machine and the compiler to predict performance, further supporting the idea that the performance of the FFT on SIMD microprocessors can be predicted and reasoned about.

8.2 contributions

The contributions of this work are summarized as follows:

1. Three methods of computing the conjugate-pair algorithm on SIMD microprocessors are presented in Chapter 5. The three techniques are suited for different sizes of transform, but in general, all tech-

niques are amenable to algorithm level vectorization, and latency and memory locality optimizations. These techniques are shown to produce results that are, in many cases, faster than state of the art libraries such as FFTW and SPIRAL, but without extensive machine calibration;

2. The source code for the library developed in this thesis, SFFT, is publicly available under a permissive open source licence at `https://github.com/anthonix/sfft`. A permissive open source licence will hopefully ensure that SFFT is developed further.

8.3 FUTURE WORK

This section presents some ideas for future work that can be divided into four categories: measurement, modelling, systems and applications.

8.3.1 *Measurement*

FFTW could be instrumented to collect data on the huge space of transforms it evaluates, which could then be used to build more accurate models.

The existing FFT benchmarking infrastructure could be improved by detecting interruption by other system processes and re-running the affected results. Benchmarks could then be run on a much wider range of machines, under more controlled conditions, which would increase the accuracy of models built from the data.

8.3.2 *Modelling*

It might be possible to build a classifier that predicts whether a transform is likely, given some threshold, to be the fastest. The fastest is then selected from a subset of those that are likely to be the fastest, and thus the number of transforms that must be evaluated during calibration is reduced, while sacrificing little or no performance.

8.3.3 *Systems*

SFFT could be extended to multi-dimensional, multi-threaded, real, large (megapoint and above) and arbitrary sized transforms. Additionally, support for other architectures such as POWER and Cell B.E. could be added. Code could be optimized between transforms in a library, which would reduce binary size, but there may be other effects.

8.3.4 *Algorithms*

So far, there have been no known attempts to seriously optimize the tangent FFT, and the results of optimizing the tangent FFT to the same degree as the conjugate-pair FFT in this thesis would be very interesting.

SFFT could be utilized in the sparse FFT algorithms which have recently been published, perhaps improving their performance even further.

8.3.5 *Applications*

Applications such as the SETI@home client could be patched to support SFFT. The results of benchmarks between SFFT, FFTW and other libraries, when used in real world applications such as SETI@home, would be of great interest.

8.4 FINAL REMARKS

This thesis showed that high-performance computation of the FFT is by no means a solved problem, and it is hoped that this work will serve as a catalyst or foundation for future efforts that push efficiency and performance even further.

BIBLIOGRAPHY

[1] BenchFFT, a program to compare the performance and accuracy of many different FFT implementations. URL: `http://www.fftw.org/benchfft/`, November 2011.

[2] SPIRAL: A generator for platform-adapted libraries of signal processing alogorithms. URL: `http://www.spiral.net/index.html`, November 2011.

[3] A. Ali and L. Johnsson. UHFFT: A high performance DFT framework. 2006.

[4] A. Ali, L. Johnsson, and D. Mirkovic. Empirical auto-tuning code generator for FFT and trignometric transforms. In *ODES: 5th Workshop on Optimizations for DSP and Embedded Systems, in conjunction with International Symposium on Code Generation and Optimization (CGO)*, 2007.

[5] A. Ali, L. Johnsson, and J. Subhlok. Adaptive computation of self sorting in-place FFTs on hierarchical memory architectures. *High Performance Computing and Communications*, pages 372–383, 2007.

[6] A. Ali, L. Johnsson, and J. Subhlok. Scheduling FFT computation on SMP and multicore systems. In *Proceedings of the 21st annual international conference on Supercomputing*, pages 293–301. ACM, 2007.

[7] J. Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[8] D.H. Bailey. FFTs in external or hierarchical memory. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 234–242. ACM, 1989.

[9] D. Bernstein. The tangent FFT. *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 291–300, 2007.

[10] D. Bernstein. djbfft. URL: `http://cr.yp.to/djbfft.html`, August 2011.

[11] L. Bluestein. A linear filtering approach to the computation of discrete fourier transform. *Audio and Electroacoustics, IEEE Transactions on*, 18(4):451–455, 1970.

[12] Mark Buchanan. Heated debate in different dimensions. *Nat Phys*, 1(2):71–71, 11 2005. URL `http://dx.doi.org/10.1038/nphys157`.

[13] CSS Burrus and T.W. Parks. *DFT/FFT and Convolution Algorithms: theory and Implementation*. John Wiley & Sons, Inc., 1991.

[14] F. Carl. Werke, band 3, Königlichen Gesellschaft der Wissenschaften, Göttingen, 1866. pages 308–310.

[15] W.H. Chang and T.Q. Nguyen. On the fixed-point accuracy analysis of FFT algorithms. *Signal Processing, IEEE Transactions on*, 56(10): 4673–4682, 2008.

[16] J.W. Cooley and J.W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90): 297–301, 1965.

[17] R.E. Crochiere and A.V. Oppenheim. Analysis of linear digital networks. *Proceedings of the IEEE*, 63(4):581–595, 1975.

[18] G.C. Danielson and C. Lanczos. Some improvements in practical Fourier analysis and their application to x-ray scattering from liquids. *Journal of the Franklin Institute*, 233(5):435–452, 1942.

[19] E.W. Dijkstra. A review of the 1977 Turing Award Lecture by John Backus. 1978.

[20] J. Dongarra and F. Sullivan. Guest editors' introduction: The top 10 algorithms. *Computing in Science & Engineering*, pages 22–23, 2000.

[21] P. Duhamel. Algorithms meeting the lower bounds on the multiplicative complexity of length-2n DFTs and their connection with practical algorithms. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 38(9):1504–1511, 1990.

[22] P. Duhamel and H. Hollmann. Split radix FFT algorithm. *Electronics Letters*, 20(1):14–16, 1984.

[23] P. Duhamel and M. Vetterli. Fast Fourier transforms: a tutorial review and a state of the art. *Signal Processing*, 19(4):259–299, 1990.

[24] R. Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, 1990.

[25] C.M. Fiduccia. Polynomial evaluation via the division algorithm the fast Fourier transform revisited. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 88–93. ACM, 1972.

[26] Jason Fitzpatrick. An interview with Steve Furber. *Communications of the ACM*, 54(5):34–39, 2011.

[27] M.J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972.

[28] F. Franchetti. *Performance portable short vector transforms*. PhD thesis, Technische Universität Wien, 2003.

[29] F. Franchetti and M. Puschel. A SIMD vectorizing compiler for digital signal processing algorithms. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 20–26. IEEE, 2002.

[30] F. Franchetti and M. Puschel. Short vector code generation and adaptation for DSP algorithms. In *Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03). 2003 IEEE International Conference on*, volume 2, pages II–537. IEEE, 2003.

[31] F. Franchetti, S. Kral, J. Lorenz, and C.W. Ueberhuber. Efficient utilization of SIMD extensions. *Proceedings of the IEEE*, 93(2):409–425, 2005.

[32] F. Franchetti, Y. Voronenko, and M. Puschel. FFT program generation for shared memory: SMP and multicore. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages 51–51. IEEE, 2006.

[33] F. Franchetti, Y. Voronenko, and M. Püschel. A rewriting system for the vectorization of signal transforms. *High Performance Computing for Computational Science-VECPAR 2006*, pages 363–377, 2007.

[34] M. Frigo. A fast Fourier transform compiler. In *ACM SIGPLAN Notices*, volume 34, pages 169–180. ACM, 1999.

[35] M. Frigo and S.G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[36] R. Garg and R. Telang. Estimating app demand from publicly available data. 2011.

[37] R. A. Gopinath. Conjugate pair fast Fourier transform. *Electronics Letters*, 25:1084, 1989.

[38] H. Hassanieh, P. Indyk, D. Katabi, and E. Price. Nearly optimal sparse Fourier transform. *Arxiv preprint arXiv:1201.2501*, 2012.

[39] H. Hassanieh, P. Indyk, D. Katabi, and E. Price. Simple and practical algorithm for sparse Fourier transform. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1183–1194. SIAM, 2012.

[40] M. Heideman and C. Burrus. On the number of multiplications necessary to compute a length-2n DFT. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 34(1):91–95, 1986.

[41] M.T. Heideman, D.H. Johnson, and C.S. Burrus. Gauss and the history of the fast Fourier transform. *Archive for history of exact sciences*, 34(3):265–277, 1985.

[42] Apple Inc. Using the Accelerate framework for data processing, 09 2010. URL `http://developer.apple.com/library/mac/#featuredarticles/AccelerateFrameworkData/_index.html`.

[43] Apple Inc. Advanced Computation Group. URL: `http://www.apple.com/acg/`, November 2011.

[44] Intel Inc. Intel IPP. URL: `http://software.intel.com/en-us/articles/intel-ipp/`, November 2011.

[45] JR Johnson and RW Johnson. Challenges of computing the fast Fourier transform. In *DARPA Conference*. Citeseer, 1997.

[46] S. G. Johnson and M. Frigo. Implementing FFTs in practice. In C. S. Burrus, editor, *Fast Fourier Transforms*, Connexions, chapter 11. Rice University, Houston TX, September 2008.

[47] S.G. Johnson and M. Frigo. A modified split-radix FFT with fewer arithmetic operations. *Signal Processing, IEEE Transactions on*, 55(1): 111–119, 2006. ISSN 1053-587X.

[48] I. Kamar and Y. Elcherif. Conjugate pair fast Fourier transform. *Electronics Letters*, 25:324, 1989.

[49] Vasilios I. Kelefouras, George Athanasiou, Nikolaos Alachiotis, Harris E. Michail, Angeliki Kritikakou, and Costas E. Goutis. A methodology for speeding up fast Fourier transform focusing on memory architecture utilization. *IEEE Transactions on Signal Processing*, 59 (12):6217–6226, 2011.

[50] D.E. Knuth. *The art of computer programming. Vol. 3: sorting and searching*, volume 307. Addison Wesley, 1973.

[51] S. Kral, F. Franchetti, J. Lorenz, and C. Ueberhuber. SIMD vectorization of straight line FFT code. *Euro-Par 2003 Parallel Processing*, pages 251–260, 2003.

[52] S. Kral, F. Franchetti, J. Lorenz, C. Ueberhuber, and P. Wurzinger. FFT compiler techniques. In *Compiler Construction*, pages 2725–2725. Springer, 2004.

[53] A. M. Krot and H. B. Minervina. Conjugate pair fast Fourier transform. *Electronics Letters*, 28:1143, 1992.

[54] J. Lorenz, S. Kral, F. Franchetti, and C.W. Ueberhuber. Vectorization techniques for the Blue Gene/L double FPU. *IBM Journal of Research and Development*, 49(2.3):437–446, 2005.

[55] T. Lundy and J. Van Buskirk. A new matrix approach to real FFTs and convolutions of length 2 k. *Computing*, 80(1):23–45, 2007.

[56] J.B. Martens. Recursive cyclotomic factorization–A new algorithm for calculating the discrete Fourier transform. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 32(4):750–761, 1984.

[57] T. Mateer. *Fast Fourier transform algorithms with applications*. PhD thesis, Clemson University, 2008.

[58] Dylan McGrath. IDC cuts PC microprocessor forecast. *EETimes*, September 2011.

[59] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pages 23–23. Usenix Association, 1996.

[60] D. Mirkovic and S.L. Johnsson. Automatic performance tuning in the UHFFT library. *Lecture notes in computer science*, pages I–71, 2001.

[61] D. Mirkovic, R. Mahasoom, and L. Johnsson. Adaptive software library for fast Fourier transforms. In *2000 International Conference on Supercomputing*, pages 215–224, 2000.

[62] J.E. Moreira, G. Almási, C. Archer, R. Bellofatto, P. Bergner, J.R. Brunheroto, M. Brutman, J.G. Castanos, PG Crumley, M. Gupta, et al. Blue Gene/L programming and operating environment. *IBM Journal of Research and Development*, 49(2.3):367–376, 2005.

[63] C.G. Nevill-Manning and I.H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[64] A.V. Oppenheim, R.W. Schafer, J.R. Buck, et al. *Discrete-time signal processing*, volume 2. Prentice hall Upper Saddle Riverˆ eN. JNJ, 1989.

[65] T. Pitkanen, R. Makinen, J. Heikkinen, T. Partanen, and J. Takala. Low-power, high-performance TTA processor for 1024-point fast Fourier transform. *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 227–236, 2006.

[66] M. Puschel, J.M.F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R.W. Johnson. SPIRAL: A generator for platform-adapted libraries of signal processing alogorithms. *International Journal of High Performance Computing Applications*, 18(1):21, 2004.

[67] M. Puschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, et al. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2): 232–275, 2005.

[68] H-S. Qian and Z-J. Zhao. Conjugate pair fast Fourier transform. *Electronics Letters*, 26:541, 1990.

[69] J. Klivington R. Crandall and D. Mitchell. Large-scale FFTs and convolutions on Apple hardware. Technical report, Apple Inc Advanced Computation Group, 2009.

[70] L. Rabiner, R. Schafer, and C. Rader. The chirp z-transform algorithm. *Audio and Electroacoustics, IEEE Transactions on*, 17(2):86–92, 1969.

[71] C.M. Rader. Discrete Fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56(6):1107–1108, 1968.

[72] A. Robinson. *Handbook of automated reasoning*, volume 2. Elsevier, 2001.

[73] Richard C. Singleton. On computing the fast Fourier transform. *Commun. ACM*, 10:647–654, October 1967. doi: http://doi.acm.org/10.1145/363717.363771.

[74] C. Temperton. Note on prime factor FFT algorithms. *J. Comput. Phys.*, 1, 1983.

[75] M. Vetterli, H.J. Nussbaumer, et al. Simple FFT and DCT algorithms with reduced number of operations. *Signal Processing*, 6(4):267–278, 1984.

[76] S. Winograd. On computing the discrete Fourier transform. *Mathematics of Computation*, 32(141):175–199, 1978.

[77] I.H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[78] R. Yavne. An economical method for calculating the discrete Fourier transform. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 115–125. ACM, 1968.

Part III

APPENDICES

A

SIMPLE FFTS

This Appendix contains source code listings corresponding to the simple implementations in Section 3.1.

Listing 20: Simple radix-2 FFT

```c
#include <complex.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

typedef complex float data_t;

#define W(N,k) (cexp(-2.0f * M_PI * I * (float)k / (float)N))

void ditfft2(data_t *in, data_t *out, int stride, int N) {
  if(N == 2) {
    out[0]   = in[0] + in[stride];
    out[N/2] = in[0] - in[stride];
  }else{
    ditfft2(in, out, stride << 1, N >> 1);
    ditfft2(in+stride, out+N/2, stride << 1, N >> 1);

    {  /* k=0 -> no multiplication */
      data_t Ek = out[0];
      data_t Ok = out[N/2];
      out[0]   = Ek + Ok;
      out[N/2] = Ek - Ok;
    }

    int k;
    for(k=1;k<N/2;k++) {
      data_t Ek = out[k];
      data_t Ok = out[(k+N/2)];
      out[k]         = Ek + W(N,k) * Ok;
      out[(k+N/2) ] = Ek - W(N,k) * Ok;
    }
  }
}
```

Listing 21: Simple split-radix FFT

```c
#include <complex.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

typedef complex float data_t;

#define W(N,k) (cexp(-2.0f * M_PI * I * (float)k / (float)N))

void splitfft(data_t *in, data_t *out, int stride, int N) {
  if(N == 1) {
    out[0] = in[0];
  }else if(N == 2) {
    out[0]   = in[0] + in[stride];
    out[N/2] = in[0] - in[stride];
  }else{
    splitfft(in, out, stride << 1, N >> 1);
    splitfft(in+stride, out+N/2, stride << 2, N >> 2);
    splitfft(in+3*stride, out+3*N/4, stride << 2, N >> 2);

    {
      data_t Uk  = out[0];
      data_t Zk  = out[0+N/2];
      data_t Uk2 = out[0+N/4];
      data_t Zdk = out[0+3*N/4];
      out[0]       = Uk  + (Zk + Zdk);
      out[0+N/2]   = Uk  - (Zk + Zdk);
      out[0+N/4]   = Uk2 - I*(Zk - Zdk);
      out[0+3*N/4] = Uk2 + I*(Zk - Zdk);
    }
    int k;
    for(k=1;k<N/4;k++) {
      data_t Uk  = out[k];
      data_t Zk  = out[k+N/2];
      data_t Uk2 = out[k+N/4];
      data_t Zdk = out[k+3*N/4];
      out[k]       = Uk  + (W(N,k)*Zk + W(N,3*k)*Zdk);
      out[k+N/2]   = Uk  - (W(N,k)*Zk + W(N,3*k)*Zdk);
      out[k+N/4]   = Uk2 - I*(W(N,k)*Zk - W(N,3*k)*Zdk);
      out[k+3*N/4] = Uk2 + I*(W(N,k)*Zk - W(N,3*k)*Zdk);
    }
  }
}
```

Listing 22: Simple conjugate-pair FFT

```
#include <complex.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

typedef complex float data_t;

#define W(N,k) (cexp(-2.0f * M_PI * I * (float)k / (float)N))

void conjfft(data_t *base, int TN,
  data_t *in, data_t *out, int stride, int N) {
  if(N == 1) {
    if(in < base) in += TN;
    out[0] = in[0];
  }else if(N == 2) {
    data_t *i0 = in, *i1 = in + stride;
    if(i0 < base) i0 += TN;
    if(i1 < base) i1 += TN;
    out[0]   = *i0 + *i1;
    out[N/2] = *i0 - *i1;
  }else{
    conjfft(base, TN, in, out, stride << 1, N >> 1);
    conjfft(base, TN, in+stride, out+N/2, stride << 2, N >> 2);
    conjfft(base, TN, in-stride, out+3*N/4, stride << 2, N >> 2);

    {
      data_t Uk  = out[0];
      data_t Zk  = out[0+N/2];
      data_t Uk2 = out[0+N/4];
      data_t Zdk = out[0+3*N/4];
      out[0]       = Uk  + (Zk + Zdk);
      out[0+N/2]   = Uk  - (Zk + Zdk);
      out[0+N/4]   = Uk2 - I*(Zk - Zdk);
      out[0+3*N/4] = Uk2 + I*(Zk - Zdk);
    }
    int k;
```

```c
    for(k=1;k<N/4;k++) {
      data_t Uk  = out[k];
      data_t Zk  = out[k+N/2];
      data_t Uk2 = out[k+N/4];
      data_t Zdk = out[k+3*N/4];
      data_t w = W(N,k);
      out[k]       = Uk  + (w*Zk + conj(w)*Zdk);
      out[k+N/2]   = Uk  - (w*Zk + conj(w)*Zdk);
      out[k+N/4]   = Uk2 - I*(w*Zk - conj(w)*Zdk);
      out[k+3*N/4] = Uk2 + I*(w*Zk - conj(w)*Zdk);
    }
  }
}
```

Listing 23: Simple tangent FFT

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <complex.h>

typedef complex float data_t;

#define W(N,k) (cexp(-2.0f * M_PI * I * (float)(k) / (float)(N)))

float
s(int n, int k) {
  if (n <= 4) return 1.0f;

  int k4 = k % (n/4);

  if (k4 <= n/8) return (s(n/4,k4) * cosf(2.0f * M_PI * (float)k4 / (float)n));
  return (s(n/4,k4) * sinf(2.0f * M_PI * (float)k4 / (float)n));
}

void tangentfft8(data_t *base, int TN, data_t *in, data_t *out, int stride, int N) {
  if(N == 1) {
    if(in < base) in += TN;
    out[0] = in[0];
  }else if(N == 2) {
    data_t *i0 = in, *i1 = in + stride;
    if(i0 < base) i0 += TN;
    if(i1 < base) i1 += TN;
    out[0]   = *i0 + *i1;
    out[N/2] = *i0 - *i1;
  }else if(N == 4) {
    tangentfft8(base, TN, in, out, stride << 1, N >> 1);
    tangentfft8(base, TN, in+stride, out+2, stride << 1, N >> 1);

    data_t temp1 = out[0] + out[2];
    data_t temp2 = out[0] - out[2];
    out[0] = temp1;
    out[2] = temp2;
    temp1 = out[1] - I*out[3];
    temp2 = out[1] + I*out[3];
    out[1] = temp1;
    out[3] = temp2;

  }else{
    tangentfft8(base, TN, in, out, stride << 2, N >> 2);
    tangentfft8(base, TN, in+(stride*2), out+2*N/8, stride << 3, N >> 3);
    tangentfft8(base, TN, in-(stride*2), out+3*N/8, stride << 3, N >> 3);
    tangentfft8(base, TN, in+(stride), out+4*N/8, stride << 2, N >> 2);
    tangentfft8(base, TN, in-(stride), out+6*N/8, stride << 2, N >> 2);
    int k;
    for(k=0;k<N/8;k++) {
      float s4 = s(N/4,k)/s(N,k);
      float s4_n8 = s(N/4,k+N/8)/s(N,k+N/8);

      float s2    = s(N/2,k)/s(N,k);
      float s2_n8 = s(N/2,k+N/8)/s(N,k+N/8);

      data_t w0 = W(N,k)*s4;
      data_t w1 = W(N,k+N/8)*s4_n8;
      data_t w2 = W(N,2*k)*s(N/8,k)/s(N/2,k);

      data_t zk_p   = w0       * out[k+4*N/8];
      data_t zk_n   = conj(w0) * out[k+6*N/8];
      data_t zk2_p  = w1       * out[k+5*N/8];
      data_t zk2_n  = conj(w1) * out[k+7*N/8];
      data_t uk     = out[k]                  * s4;
      data_t uk2    = out[k+N/8]              * s4_n8;
      data_t yk_p   = w2       * out[k+2*N/8];
      data_t yk_n   = conj(w2) * out[k+3*N/8];

      data_t y0 = (yk_p + yk_n)*s2;
      data_t y1 = (yk_p - yk_n)*I*s2_n2;

      out[k]       = uk + y0 + (zk_p + zk_n);
      out[k+4*N/8] = uk + y0 - (zk_p + zk_n);
      out[k+2*N/8] = uk - y0 - I*(zk_p - zk_n);
      out[k+6*N/8] = uk - y0 + I*(zk_p - zk_n);
      out[k+1*N/8] = uk2 - y1 +   (zk2_p + zk2_n);
      out[k+3*N/8] = uk2 + y1 - I*(zk2_p - zk2_n);
      out[k+5*N/8] = uk2 - y1 -   (zk2_p + zk2_n);
      out[k+7*N/8] = uk2 + y1 + I*(zk2_p - zk2_n);
    }
  }

}

void tangentfft4(data_t *base, int TN,
                 data_t *in, data_t *out, int stride, int N) {
  if(N == 1) {
    if(in < base) in += TN;
    out[0] = in[0];
  }else if(N == 2) {
    data_t *i0 = in, *i1 = in + stride;
    if(i0 < base) i0 += TN;
    if(i1 < base) i1 += TN;
    out[0]   = *i0 + *i1;
    out[N/2] = *i0 - *i1;
  }else{
    tangentfft4(base, TN, in, out, stride << 1, N >> 1);
    tangentfft8(base, TN, in+stride, out+N/2, stride << 2, N >> 2);
    tangentfft8(base, TN, in-stride, out+3*N/4, stride << 2, N >> 2);

    {
      data_t Uk  = out[0];
      data_t Zk  = out[0+N/2];
      data_t Uk2 = out[0+N/4];
      data_t Zdk = out[0+3*N/4];
      out[0]       = Uk  + (Zk + Zdk);
      out[0+N/2]   = Uk  - (Zk + Zdk);
      out[0+N/4]   = Uk2 - I*(Zk - Zdk);
      out[0+3*N/4] = Uk2 + I*(Zk - Zdk);
    }
    int k;
    for(k=1;k<N/4;k++) {
      data_t Uk  = out[k];
      data_t Zk  = out[k+N/2];
      data_t Uk2 = out[k+N/4];
      data_t Zdk = out[k+3*N/4];
      data_t w = W(N,k)*s(N/4,k);
      out[k]       = Uk  + (w*Zk + conj(w)*Zdk);
      out[k+N/2]   = Uk  - (w*Zk + conj(w)*Zdk);
      out[k+N/4]   = Uk2 - I*(w*Zk - conj(w)*Zdk);
      out[k+3*N/4] = Uk2 + I*(w*Zk - conj(w)*Zdk);
    }
  }
}
```

B

FFTS WITH PRECOMPUTED LUTS

This Appendix contains source code listings corresponding to the FFT implementations with precomputed coefficients in Section 3.2.

Listing 24: Simple radix-2 FFT with precomputed LUT

```c
#include <math.h>
#include <complex.h>
#include <stdio.h>
#include <stdlib.h>

typedef complex float data_t;

#define W(N,k) (cexp(-2.0f * M_PI * I * (float)k / (float)N))

data_t *LUT;

void ditfft2(data_t *in, data_t *out, int log2stride, int stride, int N) {
  if(N == 2) {
    out[0]   = in[0] + in[stride];
    out[N/2] = in[0] - in[stride];
  }else{
    ditfft2(in, out, log2stride+1, stride << 1, N >> 1);
    ditfft2(in+stride, out+N/2, log2stride+1, stride << 1, N >> 1);

    {  /* k=0 -> no multiplication */
      data_t Ek = out[0];
      data_t Ok = out[N/2];
      out[0]   = Ek + Ok;
      out[N/2] = Ek - Ok;
    }

    int k;
    for(k=1;k<N/2;k++) {
      data_t Ek = out[k];
      data_t Ok = out[(k+N/2)];
      data_t w = LUT[k<<log2stride];
      out[k]        = Ek + w * Ok;
      out[(k+N/2) ] = Ek - w * Ok;
    }
  }
}

void fft_init(int N) {
  LUT = malloc(N/2 * sizeof(data_t));
  int i;
  for(i=0;i<N/2;i++) LUT[i] = W(N,i);
}
```

Listing 25: Simple split-radix FFT with precomputed LUTs

```c
#include <math.h>
```

```
#include <complex.h>
#include <stdio.h>
#include <stdlib.h>

typedef complex float data_t;

#define W(N,k) (cexp(-2.0f * M_PI * I * (float)k / (float)N))
data_t *LUT1;
data_t *LUT3;

void splitfft(data_t *in, data_t *out, int log2stride, int stride, int N) {
  if(N == 1) {
    out[0] = in[0];
  }else if(N == 2) {
    out[0]   = in[0] + in[stride];
    out[N/2] = in[0] - in[stride];
  }else{
    splitfft(in, out, log2stride+1, stride << 1, N >> 1);
    splitfft(in+stride, out+N/2, log2stride+2, stride << 2, N >> 2);
    splitfft(in+3*stride, out+3*N/4, log2stride+2, stride << 2, N >> 2);

    {
      data_t Uk  = out[0];
      data_t Zk  = out[0+N/2];
      data_t Uk2 = out[0+N/4];
      data_t Zdk = out[0+3*N/4];
      out[0]       = Uk  + (Zk + Zdk);
      out[0+N/2]   = Uk  - (Zk + Zdk);
      out[0+N/4]   = Uk2 - I*(Zk - Zdk);
      out[0+3*N/4] = Uk2 + I*(Zk - Zdk);
    }
    int k;
    for(k=1;k<N/4;k++) {
      data_t Uk  = out[k];
      data_t Zk  = out[k+N/2];
      data_t Uk2 = out[k+N/4];
      data_t Zdk = out[k+3*N/4];
      data_t w1 = LUT1[k<<log2stride];
      data_t w3 = LUT3[k<<log2stride];
      out[k]       = Uk  + (w1*Zk + w3*Zdk);
      out[k+N/2]   = Uk  - (w1*Zk + w3*Zdk);
      out[k+N/4]   = Uk2 - I*(w1*Zk - w3*Zdk);
      out[k+3*N/4] = Uk2 + I*(w1*Zk - w3*Zdk);
    }
  }
}

void fft_init(int N) {
  LUT1 = malloc(N/4 * sizeof(data_t));
  LUT3 = malloc(N/4 * sizeof(data_t));
  int i;
  for(i=0;i<N/4;i++) LUT1[i] = W(N,i);
  for(i=0;i<N/4;i++) LUT3[i] = W(N,3*i);
}
```

Listing 26: Simple conjugate-pair FFT with precomputed LUT

```
#include <math.h>
#include <complex.h>
#include <stdio.h>
#include <stdlib.h>

typedef complex float data_t;

#define W(N,k) (cexp(-2.0f * M_PI * I * (float)k / (float)N))
data_t *LUT;

void conjfft(data_t *base, int TN,
  data_t *in, data_t *out, int log2stride, int stride, int N) {
  if(N == 1) {
    if(in < base) in += TN;
    out[0] = in[0];
```

```c
  }else if(N == 2) {
    data_t *i0 = in, *i1 = in + stride;
    if(i0 < base) i0 += TN;
    if(i1 < base) i1 += TN;
    out[0]   = *i0 + *i1;
    out[N/2] = *i0 - *i1;
  }else{
    conjfft(base, TN, in, out, log2stride+1, stride << 1, N >> 1);
    conjfft(base, TN, in+stride, out+N/2, log2stride+2, stride << 2, N >> 2);
    conjfft(base, TN, in-stride, out+3*N/4, log2stride+2, stride << 2, N >> 2);

    {
      data_t Uk  = out[0];
      data_t Zk  = out[0+N/2];
      data_t Uk2 = out[0+N/4];
      data_t Zdk = out[0+3*N/4];
      out[0]       = Uk  + (Zk + Zdk);
      out[0+N/2]   = Uk  - (Zk + Zdk);
      out[0+N/4]   = Uk2 - I*(Zk - Zdk);
      out[0+3*N/4] = Uk2 + I*(Zk - Zdk);
    }
    int k;
    for(k=1;k<N/4;k++) {
      data_t Uk  = out[k];
      data_t Zk  = out[k+N/2];
      data_t Uk2 = out[k+N/4];
      data_t Zdk = out[k+3*N/4];
      data_t w = LUT[k<<log2stride];
      out[k]       = Uk  + (w*Zk + conj(w)*Zdk);
      out[k+N/2]   = Uk  - (w*Zk + conj(w)*Zdk);
      out[k+N/4]   = Uk2 - I*(w*Zk - conj(w)*Zdk);
      out[k+3*N/4] = Uk2 + I*(w*Zk - conj(w)*Zdk);
    }
  }
}

void fft_init(int N) {
  LUT = malloc(N/4 * sizeof(data_t));
  int i;
  for(i=0;i<N/4;i++) LUT[i] = W(N,i);
}
```

Listing 27: Simple tangent FFT with precomputed LUTs

```c
#include <math.h>
#include <complex.h>
#include <stdio.h>
#include <stdlib.h>

typedef complex float data_t;

#define W(N,k) (cexp(-2.0f * M_PI * I * (float)(k) / (float)(N)))

float
s(int n, int k) {
  if (n <= 4) return 1.0f;

  int k4 = k % (n/4);

  if (k4 <= n/8)
    return (s(n/4,k4) * cosf(2.0f * M_PI * (float)k4 / (float)n));
  return (s(n/4,k4) * sinf(2.0f * M_PI * (float)k4 / (float)n));
}

data_t *LUT, *LUT0, *LUT1, *LUT2;
float *s2, *s4;

void tangentfft8(data_t *base, int TN, data_t *in, data_t *out, int log2stride,
    int stride, int N) {
  if(N == 1) {
    if(in < base) in += TN;
    out[0] = in[0];
  }else if(N == 2) {
    data_t *i0 = in, *i1 = in + stride;
    if(i0 < base) i0 += TN;
    if(i1 < base) i1 += TN;
    out[0]   = *i0 + *i1;
    out[N/2] = *i0 - *i1;
  }else if(N == 4) {
    tangentfft8(base, TN, in, out, log2stride+1, stride << 1, N >> 1);
    tangentfft8(base, TN, in+stride, out+2, log2stride+1, stride << 1, N >> 1);

    data_t temp1 = out[0] + out[2];
    data_t temp2 = out[0] - out[2];
    out[0] = temp1;
    out[2] = temp2;
    temp1 = out[1] - I*out[3];
    temp2 = out[1] + I*out[3];
    out[1] = temp1;
    out[3] = temp2;

  }else{
    tangentfft8(base, TN, in, out, log2stride+2, stride << 2, N >> 2);
    tangentfft8(base, TN, in+(stride*2), out+2*N/8, log2stride+3, stride << 3, N >> 3);
    tangentfft8(base, TN, in-(stride*2), out+3*N/8, log2stride+3, stride << 3, N >> 3);
    tangentfft8(base, TN, in+(stride), out+4*N/8, log2stride+2, stride << 2, N >> 2);
    tangentfft8(base, TN, in-(stride), out+6*N/8, log2stride+2, stride << 2, N >> 2);
    int k;
    for(k=0;k<N/8;k++) {
      data_t w0 = LUT0[k<<log2stride];
      data_t w1 = LUT1[k<<log2stride];
      data_t w2 = LUT2[k<<log2stride];

      data_t zk_p   = w0       * out[k+4*N/8];
      data_t zk_n   = conj(w0) * out[k+6*N/8];
      data_t zk2_p  = w1       * out[k+5*N/8];
      data_t zk2_n  = conj(w1) * out[k+7*N/8];
      data_t uk     = out[k]                * s4[k<<log2stride];
      data_t uk2    = out[k+N/8]            * s4[k+N/8 << log2stride];
      data_t yk_p   = w2       * out[k+2*N/8];
      data_t yk_n   = conj(w2) * out[k+3*N/8];

      data_t y0 = (yk_p + yk_n)*s2[k<<log2stride];
      data_t y1 = (yk_p - yk_n)*I*s2[k+N/8 << log2stride];

      out[k]       = uk + y0 + (zk_p + zk_n);
      out[k+4*N/8] = uk + y0 - (zk_p + zk_n);
      out[k+2*N/8] = uk - y0 - I*(zk_p - zk_n);
      out[k+6*N/8] = uk - y0 + I*(zk_p - zk_n);
      out[k+1*N/8] = uk2 - y1 +   (zk2_p + zk2_n);
      out[k+3*N/8] = uk2 + y1 - I*(zk2_p - zk2_n);
      out[k+5*N/8] = uk2 - y1 -   (zk2_p + zk2_n);
      out[k+7*N/8] = uk2 + y1 + I*(zk2_p - zk2_n);
    }
  }

}

void tangentfft4(data_t *base, int TN, data_t *in, data_t *out, int log2stride,
    int stride, int N) {
  if(N == 1) {
    if(in < base) in += TN;
    out[0] = in[0];
  }else if(N == 2) {
    data_t *i0 = in, *i1 = in + stride;
    if(i0 < base) i0 += TN;
    if(i1 < base) i1 += TN;
    out[0]   = *i0 + *i1;
    out[N/2] = *i0 - *i1;
  }else{
    tangentfft4(base, TN, in, out, log2stride+1, stride << 1, N >> 1);
    tangentfft8(base, TN, in+stride, out+N/2, log2stride+2, stride << 2, N >> 2);
    tangentfft8(base, TN, in-stride, out+3*N/4, log2stride+2, stride << 2, N >> 2);

    {
      data_t Uk  = out[0];
      data_t Zk  = out[0+N/2];
      data_t Uk2 = out[0+N/4];
      data_t Zdk = out[0+3*N/4];
      out[0]       = Uk  + (Zk + Zdk);
      out[0+N/2]   = Uk  - (Zk + Zdk);
      out[0+N/4]   = Uk2 - I*(Zk - Zdk);
      out[0+3*N/4] = Uk2 + I*(Zk - Zdk);
    }
    int k;
    for(k=1;k<N/4;k++) {
      data_t Uk  = out[k];
      data_t Zk  = out[k+N/2];
      data_t Uk2 = out[k+N/4];
      data_t Zdk = out[k+3*N/4];
      data_t w = LUT[k<<log2stride];
      out[k]       = Uk  + (w*Zk + conj(w)*Zdk);
      out[k+N/2]   = Uk  - (w*Zk + conj(w)*Zdk);
      out[k+N/4]   = Uk2 - I*(w*Zk - conj(w)*Zdk);
      out[k+3*N/4] = Uk2 + I*(w*Zk - conj(w)*Zdk);
    }
  }
}

void fft_init(int N) {
  LUT0 = malloc(N/8 * sizeof(data_t));
  LUT1 = malloc(N/8 * sizeof(data_t));
  LUT2 = malloc(N/8 * sizeof(data_t));
  LUT = malloc(N/4 * sizeof(data_t));

  s2 = malloc(N/4 * sizeof(float));
  s4 = malloc(N/4 * sizeof(float));

  int i;
  for(i=0;i<N/8;i++) LUT0[i] = W(N,i)*s(N/4,i)/s(N,i);
  for(i=0;i<N/8;i++) LUT1[i] = W(N,i+N/8)*s(N/4,i+N/8)/s(N,i+N/8);
  for(i=0;i<N/8;i++) LUT2[i] = W(N,2*i)*s(N/8,i)/s(N/2,i);
  for(i=0;i<N/4;i++) LUT[i] = W(N,i)*s(N/4,i);
  for(i=0;i<N/4;i++) s4[i] = s(N/4,i)/s(N,i);
  for(i=0;i<N/4;i++) s2[i] = s(N/2,i)/s(N,i);
}
```

C

FFTS WITH VECTORIZED LOOPS

This Appendix contains source code listings corresponding to the vectorized FFT implementations in Section 3.3.

Listing 28: Radix-2 FFT with vectorized loops

```c
#include <math.h>
#include <complex.h>
#include <stdio.h>
#include <stdlib.h>
#include <xmmintrin.h>

typedef complex float data_t;

#define W(N,k) (cexp(-2.0f * M_PI * I * (float)(k) / (float)(N)))

data_t **LUT;

void ditfft2(data_t *in, data_t *out, int log2stride, int stride, int N) {
  if(N == 2) {
    out[0]   = in[0] + in[stride];
    out[N/2] = in[0] - in[stride];
  }else if(N == 4){
    ditfft2(in, out, log2stride+1, stride << 1, N >> 1);
    ditfft2(in+stride, out+N/2, log2stride+1, stride << 1, N >> 1);

    data_t temp0 = out[0] + out[2];
    data_t temp1 = out[0] - out[2];
    data_t temp2 = out[1] - I*out[3];
    data_t temp3 = out[1] + I*out[3];
    if(log2stride) {
      out[0] = creal(temp0) + creal(temp2)*I;
      out[1] = creal(temp1) + creal(temp3)*I;
      out[2] = cimag(temp0) + cimag(temp2)*I;
      out[3] = cimag(temp1) + cimag(temp3)*I;
    }else{
      out[0] = temp0;
      out[2] = temp1;
      out[1] = temp2;
      out[3] = temp3;
    }
  }else if(!log2stride){
    ditfft2(in, out, log2stride+1, stride << 1, N >> 1);
    ditfft2(in+stride, out+N/2, log2stride+1, stride << 1, N >> 1);

    int k;
    for(k=0;k<N/2;k+=4) {
      __m128 Ok_re = _mm_load_ps((float *)&out[k+N/2]);
      __m128 Ok_im = _mm_load_ps((float *)&out[k+N/2+2]);
      __m128 w_re = _mm_load_ps((float *)&LUT[log2stride][k]);
      __m128 w_im = _mm_load_ps((float *)&LUT[log2stride][k+2]);
      __m128 Ek_re = _mm_load_ps((float *)&out[k]);
      __m128 Ek_im = _mm_load_ps((float *)&out[k+2]);
      __m128 wOk_re = _mm_sub_ps(_mm_mul_ps(Ok_re,w_re),_mm_mul_ps(Ok_im,w_im));
      __m128 wOk_im = _mm_add_ps(_mm_mul_ps(Ok_re,w_im),_mm_mul_ps(Ok_im,w_re));
      __m128 out0_re = _mm_add_ps(Ek_re, wOk_re);
      __m128 out0_im = _mm_add_ps(Ek_im, wOk_im);
      __m128 out1_re = _mm_sub_ps(Ek_re, wOk_re);
      __m128 out1_im = _mm_sub_ps(Ek_im, wOk_im);
      _mm_store_ps((float *)(out+k), _mm_unpacklo_ps(out0_re, out0_im));
      _mm_store_ps((float *)(out+k+2), _mm_unpackhi_ps(out0_re, out0_im));
      _mm_store_ps((float *)(out+k+N/2), _mm_unpacklo_ps(out1_re, out1_im));
      _mm_store_ps((float *)(out+k+N/2+2), _mm_unpackhi_ps(out1_re, out1_im));
    }
  }else{
    ditfft2(in, out, log2stride+1, stride << 1, N >> 1);
    ditfft2(in+stride, out+N/2, log2stride+1, stride << 1, N >> 1);

    int k;
    for(k=0;k<N/2;k+=4) {
      __m128 Ok_re = _mm_load_ps((float *)&out[k+N/2]);
      __m128 Ok_im = _mm_load_ps((float *)&out[k+N/2+2]);
      __m128 w_re = _mm_load_ps((float *)&LUT[log2stride][k]);
      __m128 w_im = _mm_load_ps((float *)&LUT[log2stride][k+2]);
      __m128 Ek_re = _mm_load_ps((float *)&out[k]);
      __m128 Ek_im = _mm_load_ps((float *)&out[k+2]);
      __m128 wOk_re = _mm_sub_ps(_mm_mul_ps(Ok_re,w_re),_mm_mul_ps(Ok_im,w_im));
      __m128 wOk_im = _mm_add_ps(_mm_mul_ps(Ok_re,w_im),_mm_mul_ps(Ok_im,w_re));
      _mm_store_ps((float *)(out+k), _mm_add_ps(Ek_re, wOk_re));
      _mm_store_ps((float *)(out+k+2), _mm_add_ps(Ek_im, wOk_im));
      _mm_store_ps((float *)(out+k+N/2), _mm_sub_ps(Ek_re, wOk_re));
      _mm_store_ps((float *)(out+k+N/2+2), _mm_sub_ps(Ek_im, wOk_im));
    }
  }
}

void fft_init(int N) {
  int i;

  #define log2(x) ((int)(log(x)/log(2)))

  int n_luts = log2(N)-2;
  LUT = malloc(n_luts * sizeof(data_t *));
  for(i=0;i<n_luts;i++) {
    int n = N / pow(2,i);
    LUT[i] = _mm_malloc(n/2 * sizeof(data_t), 16);


    int j;
    for(j=0;j<n/2;j+=4) {
      data_t w[4];
      int k;
      for(k=0;k<4;k++) w[k] = W(n,j+k);

      LUT[i][j]   = creal(w[0]) + creal(w[1])*I;
      LUT[i][j+1] = creal(w[2]) + creal(w[3])*I;
      LUT[i][j+2] = cimag(w[0]) + cimag(w[1])*I;
      LUT[i][j+3] = cimag(w[2]) + cimag(w[3])*I;
    }
  }

}
```

Listing 29: Vectorized math functions for split-radix implementations

```

typedef struct _reg_t {
  __m128 re, im;
} reg_t;

static inline reg_t MUL(reg_t a, reg_t b) {
  reg_t r;
  r.re = _mm_sub_ps(_mm_mul_ps(a.re,b.re),_mm_mul_ps(a.im,b.im));
  r.im = _mm_add_ps(_mm_mul_ps(a.re,b.im),_mm_mul_ps(a.im,b.re));
  return r;
}
static inline reg_t MULJ(reg_t a, reg_t b) {
  reg_t r;
  r.re = _mm_add_ps(_mm_mul_ps(a.re,b.re),_mm_mul_ps(a.im,b.im));
  r.im = _mm_sub_ps(_mm_mul_ps(a.im,b.re),_mm_mul_ps(a.re,b.im));
  return r;
}

static inline reg_t ADD(reg_t a, reg_t b) {
  reg_t r;
  r.re = _mm_add_ps(a.re,b.re);
  r.im = _mm_add_ps(a.im,b.im);
  return r;
}
static inline reg_t SUB(reg_t a, reg_t b) {
  reg_t r;
  r.re = _mm_sub_ps(a.re,b.re);
  r.im = _mm_sub_ps(a.im,b.im);
  return r;
}
static inline reg_t ADD_I(reg_t a, reg_t b) {
  reg_t r;
  r.re = _mm_sub_ps(a.re,b.im);
  r.im = _mm_add_ps(a.im,b.re);
  return r;
}
static inline reg_t SUB_I(reg_t a, reg_t b) {
  reg_t r;
  r.re = _mm_add_ps(a.re,b.im);
  r.im = _mm_sub_ps(a.im,b.re);
  return r;
}

static inline reg_t LOAD(float *a) {
  reg_t r;
  r.re = _mm_load_ps(a);
  r.im = _mm_load_ps(a+4);
  return r;
}
static inline void STORE(float *a, reg_t r) {
  _mm_store_ps(a, r.re);
  _mm_store_ps(a+4, r.im);
}
static inline void STOREIL(float *a, reg_t r) {
  _mm_store_ps(a, _mm_unpacklo_ps(r.re, r.im));
  _mm_store_ps(a+4, _mm_unpackhi_ps(r.re, r.im));
}
```

Listing 30: Split-radix FFT with vectorized loops

```
#include <math.h>
#include <complex.h>
#include <stdio.h>
#include <stdlib.h>
#include <xmmintrin.h>

typedef complex float data_t;

#define W(N,k) (cexp(-2.0f * M_PI * I * (float)(k) / (float)(N)))
data_t **LUT1;
data_t **LUT3;

#include "vecmath.h"

void splitfft(data_t *in, data_t *out,
          int log2stride, int stride, int N) {

  if(N == 1) {
    out[0] = in[0];
  }else if(N == 2) {
    out[0] = in[0] + in[stride];
    out[1] = in[0] - in[stride];
  }else if(N == 4) {
    splitfft(in, out, log2stride+1, stride << 1, N >> 1);
    splitfft(in+stride, out+N/2, log2stride+2, stride << 2, N >> 2);
    splitfft(in+3*stride, out+3*N/4, log2stride+2, stride << 2, N >> 2);
```

```

      data_t temp0 = out[0]  + (out[2] + out[3]);
      data_t temp1 = out[0]  - (out[2] + out[3]);
      data_t temp2 = out[1] - I*(out[2] - out[3]);
      data_t temp3 = out[1] + I*(out[2] - out[3]);
    if(log2stride) {
      out[0] = creal(temp0) + creal(temp2)*I;
      out[1] = creal(temp1) + creal(temp3)*I;
      out[2] = cimag(temp0) + cimag(temp2)*I;
      out[3] = cimag(temp1) + cimag(temp3)*I;
    }else{
      out[0] = temp0;
      out[2] = temp1;
      out[1] = temp2;
      out[3] = temp3;
    }

  }else if(N == 8) {
    splitfft(in, out, log2stride+1, stride << 1, N >> 1);
    splitfft(in+stride, out+N/2, log2stride+2, stride << 2, N >> 2);
    splitfft(in+3*stride, out+3*N/4, log2stride+2, stride << 2, N >> 2);

    data_t o[8];
    {
      data_t Uk  = creal(out[0]) + creal(out[2])*I;
      data_t Zk  = out[4];
      data_t Uk2  = creal(out[1]) + creal(out[3])*I;
      data_t Zdk = out[6];

      o[0] = Uk  + (Zk + Zdk);
      o[4] = Uk  - (Zk + Zdk);
      o[2] = Uk2 - I*(Zk - Zdk);
      o[6] = Uk2 + I*(Zk - Zdk);
    }
    {
      data_t Uk = cimag(out[0]) + cimag(out[2])*I;
      data_t Zk  = out[5];
      data_t Uk2 = cimag(out[1]) + cimag(out[3])*I;
      data_t Zdk = out[7];
      data_t w1 = LUT1[log2stride][1];
      data_t w3 = LUT3[log2stride][1];

      o[1] = Uk  + (w1*Zk + w3*Zdk);
      o[5] = Uk  - (w1*Zk + w3*Zdk);
      o[3] = Uk2 - I*(w1*Zk - w3*Zdk);
      o[7] = Uk2 + I*(w1*Zk - w3*Zdk);
    }
    if(log2stride) {
      out[0] = creal(o[0]) + creal(o[1])*I;
      out[1] = creal(o[2]) + creal(o[3])*I;
      out[2] = cimag(o[0]) + cimag(o[1])*I;
      out[3] = cimag(o[2]) + cimag(o[3])*I;
      out[4] = creal(o[4]) + creal(o[5])*I;
      out[5] = creal(o[6]) + creal(o[7])*I;
      out[6] = cimag(o[4]) + cimag(o[5])*I;
      out[7] = cimag(o[6]) + cimag(o[7])*I;
    }else{
      int i;
      for(i=0;i<8;i++) out[i] = o[i];
    }
  }else if(!log2stride){

    splitfft(in, out, log2stride+1, stride << 1, N >> 1);
    splitfft(in+stride, out+N/2, log2stride+2, stride << 2, N >> 2);
    splitfft(in+3*stride, out+3*N/4, log2stride+2, stride << 2, N >> 2);
    int k;
    for(k=0;k<N/4;k+=4) {
      reg_t Uk  = LOAD((float *)&out[k]);
      reg_t Zk  = LOAD((float *)&out[k+N/2]);
      reg_t Uk2 = LOAD((float *)&out[k+N/4]);
      reg_t Zdk = LOAD((float *)&out[k+3*N/4]);
      reg_t w1 = LOAD((float *)&LUT1[log2stride][k]);
      reg_t w3 = LOAD((float *)&LUT3[log2stride][k]);

      reg_t w3Zdk = MUL(w3, Zdk);
      reg_t w1Zk = MUL(w1, Zk);
      reg_t sum = ADD(w1Zk, w3Zdk);
      reg_t dif = SUB(w1Zk, w3Zdk);

      STOREIL((float *)&out[k], ADD(Uk, sum));
      STOREIL((float *)&out[k+N/2], SUB(Uk, sum));
      STOREIL((float *)&out[k+N/4], SUB_I(Uk2, dif));
      STOREIL((float *)&out[k+3*N/4], ADD_I(Uk2, dif));
    }

  }else{
    splitfft(in, out, log2stride+1, stride << 1, N >> 1);
    splitfft(in+stride, out+N/2, log2stride+2, stride << 2, N >> 2);
    splitfft(in+3*stride, out+3*N/4, log2stride+2, stride << 2, N >> 2);

    int k;
    for(k=0;k<N/4;k+=4) {
      reg_t Uk  = LOAD((float *)&out[k]);
      reg_t Zk  = LOAD((float *)&out[k+N/2]);
      reg_t Uk2 = LOAD((float *)&out[k+N/4]);
      reg_t Zdk = LOAD((float *)&out[k+3*N/4]);
      reg_t w1 = LOAD((float *)&LUT1[log2stride][k]);
      reg_t w3 = LOAD((float *)&LUT3[log2stride][k]);

      reg_t w3Zdk = MUL(w3, Zdk);
      reg_t w1Zk = MUL(w1, Zk);
      reg_t sum = ADD(w1Zk, w3Zdk);
      reg_t dif = SUB(w1Zk, w3Zdk);

      STORE((float *)&out[k], ADD(Uk, sum));
      STORE((float *)&out[k+N/2], SUB(Uk, sum));
      STORE((float *)&out[k+N/4], SUB_I(Uk2, dif));
      STORE((float *)&out[k+3*N/4], ADD_I(Uk2, dif));
    }
  }
}

void fft_init(int N) {
  #define log2(x) ((int)(log(x)/log(2)))

  int n_luts = log2(N)-1;
  LUT1 = malloc(n_luts * sizeof(data_t *));
  LUT3 = malloc(n_luts * sizeof(data_t *));
  int i;
  for(i=0;i<n_luts;i++) {
    int n = N / pow(2,i);
    LUT1[i] = _mm_malloc(n/4 * sizeof(data_t),16);
    LUT3[i] = _mm_malloc(n/4 * sizeof(data_t),16);

    if(n == 8) {
      int j;
      for(j=0;j<n/4;j++) {
        LUT1[i][j] = W(n,j);
        LUT3[i][j] = W(n,3*j);
      }
    }else{
      int j;
      for(j=0;j<n/4;j+=4) {
        data_t w1[4], w3[4];
        int k;
        for(k=0;k<4;k++) w1[k] = W(n,j+k);
        for(k=0;k<4;k++) w3[k] = W(n,3*(j+k));

        LUT1[i][j]   = creal(w1[0]) + creal(w1[1])*I;
        LUT1[i][j+1] = creal(w1[2]) + creal(w1[3])*I;
        LUT1[i][j+2] = cimag(w1[0]) + cimag(w1[1])*I;
        LUT1[i][j+3] = cimag(w1[2]) + cimag(w1[3])*I;
        LUT3[i][j]   = creal(w3[0]) + creal(w3[1])*I;
        LUT3[i][j+1] = creal(w3[2]) + creal(w3[3])*I;
        LUT3[i][j+2] = cimag(w3[0]) + cimag(w3[1])*I;
        LUT3[i][j+3] = cimag(w3[2]) + cimag(w3[3])*I;
      }
    }
  }
}
```

Listing 31: Conjugate-pair FFT with vectorized loops

```c
#include <math.h>
#include <complex.h>
#include <stdio.h>
#include <stdlib.h>
#include <xmmintrin.h>

typedef complex float data_t;

#define W(N,k) (cexp(-2.0f * M_PI * I * (float)(k) / (float)(N)))
data_t **LUT1;

#include "vecmath.h"

data_t *base;
int TN;

void conjfft(data_t *in, data_t *out,
          int log2stride, int stride, int N) {

  if(N == 1) {
    if(in < base) in += TN;
    out[0] = in[0];
  }else if(N == 2) {
    data_t *i0 = in, *i1 = in + stride;
    if(i0 < base) i0 += TN;
    if(i1 < base) i1 += TN;
    out[0]   = *i0 + *i1;
    out[N/2] = *i0 - *i1;
  }else if(N == 4) {
    conjfft(in, out, log2stride+1, stride << 1, N >> 1);
    conjfft(in+stride, out+N/2, log2stride+2, stride << 2, N >> 2);
    conjfft(in-stride, out+3*N/4, log2stride+2, stride << 2, N >> 2);

      data_t temp0 = out[0]  + (out[2] + out[3]);
      data_t temp1 = out[0]  - (out[2] + out[3]);
      data_t temp2 = out[1] - I*(out[2] - out[3]);
      data_t temp3 = out[1] + I*(out[2] - out[3]);
    if(log2stride) {
      out[0] = creal(temp0) + creal(temp2)*I;
      out[1] = creal(temp1) + creal(temp3)*I;
      out[2] = cimag(temp0) + cimag(temp2)*I;
      out[3] = cimag(temp1) + cimag(temp3)*I;
    }else{
      out[0] = temp0;
      out[2] = temp1;
      out[1] = temp2;
      out[3] = temp3;
    }

  }else if(N == 8) {
    conjfft(in, out, log2stride+1, stride << 1, N >> 1);
    conjfft(in+stride, out+N/2, log2stride+2, stride << 2, N >> 2);
    conjfft(in-stride, out+3*N/4, log2stride+2, stride << 2, N >> 2);

    data_t o[8];
    {
      data_t Uk  = creal(out[0]) + creal(out[2])*I;
      data_t Zk  = out[4];
      data_t Uk2  = creal(out[1]) + creal(out[3])*I;
      data_t Zdk = out[6];

      o[0] = Uk  + (Zk + Zdk);
      o[4] = Uk  - (Zk + Zdk);
      o[2] = Uk2 - I*(Zk - Zdk);
      o[6] = Uk2 + I*(Zk - Zdk);
    }
    {
      data_t Uk = cimag(out[0]) + cimag(out[2])*I;
      data_t Zk  = out[5];
      data_t Uk2 = cimag(out[1]) + cimag(out[3])*I;
      data_t Zdk = out[7];
      data_t w1 = LUT1[log2stride][1];

      o[1] = Uk  + (w1*Zk + conj(w1)*Zdk);
      o[5] = Uk  - (w1*Zk + conj(w1)*Zdk);
      o[3] = Uk2 - I*(w1*Zk - conj(w1)*Zdk);
      o[7] = Uk2 + I*(w1*Zk - conj(w1)*Zdk);
    }
    if(log2stride) {
      out[0] = creal(o[0]) + creal(o[1])*I;
      out[1] = creal(o[2]) + creal(o[3])*I;
      out[2] = cimag(o[0]) + cimag(o[1])*I;
      out[3] = cimag(o[2]) + cimag(o[3])*I;
      out[4] = creal(o[4]) + creal(o[5])*I;
      out[5] = creal(o[6]) + creal(o[7])*I;
      out[6] = cimag(o[4]) + cimag(o[5])*I;
      out[7] = cimag(o[6]) + cimag(o[7])*I;
    }else{
      int i;
      for(i=0;i<8;i++) out[i] = o[i];
    }
  }else if(!log2stride){

    conjfft(in, out, log2stride+1, stride << 1, N >> 1);
    conjfft(in+stride, out+N/2, log2stride+2, stride << 2, N >> 2);
    conjfft(in-stride, out+3*N/4, log2stride+2, stride << 2, N >> 2);
    int k;
    for(k=0;k<N/4;k+=4) {
      reg_t Uk  = LOAD((float *)&out[k]);
      reg_t Zk  = LOAD((float *)&out[k+N/2]);
      reg_t Uk2 = LOAD((float *)&out[k+N/4]);
      reg_t Zdk = LOAD((float *)&out[k+3*N/4]);
      reg_t w1 = LOAD((float *)&LUT1[log2stride][k]);

      reg_t w3Zdk = MULJ(Zdk, w1);
      reg_t w1Zk = MUL(w1, Zk);
      reg_t sum = ADD(w1Zk, w3Zdk);
      reg_t dif = SUB(w1Zk, w3Zdk);

      STOREIL((float *)&out[k], ADD(Uk, sum));
      STOREIL((float *)&out[k+N/2], SUB(Uk, sum));
      STOREIL((float *)&out[k+N/4], SUB_I(Uk2, dif));
      STOREIL((float *)&out[k+3*N/4], ADD_I(Uk2, dif));
    }

  }else{
    conjfft(in, out, log2stride+1, stride << 1, N >> 1);
    conjfft(in+stride, out+N/2, log2stride+2, stride << 2, N >> 2);
    conjfft(in-stride, out+3*N/4, log2stride+2, stride << 2, N >> 2);

    int k;
    for(k=0;k<N/4;k+=4) {
      reg_t Uk  = LOAD((float *)&out[k]);
      reg_t Zk  = LOAD((float *)&out[k+N/2]);
      reg_t Uk2 = LOAD((float *)&out[k+N/4]);
      reg_t Zdk = LOAD((float *)&out[k+3*N/4]);
      reg_t w1 = LOAD((float *)&LUT1[log2stride][k]);

      reg_t w3Zdk = MULJ(Zdk, w1);
      reg_t w1Zk = MUL(w1, Zk);
      reg_t sum = ADD(w1Zk, w3Zdk);
      reg_t dif = SUB(w1Zk, w3Zdk);

      STORE((float *)&out[k], ADD(Uk, sum));
      STORE((float *)&out[k+N/2], SUB(Uk, sum));
      STORE((float *)&out[k+N/4], SUB_I(Uk2, dif));
      STORE((float *)&out[k+3*N/4], ADD_I(Uk2, dif));
    }
  }
}

void fft_init(int N) {
  #define log2(x) ((int)(log(x)/log(2)))
  int n_luts = log2(N)-1;
  LUT1 = malloc(n_luts * sizeof(data_t *));
  int i;
  for(i=0;i<n_luts;i++) {
    int n = N / pow(2,i);
    LUT1[i] = _mm_malloc(n/4 * sizeof(data_t),16);

    if(n == 8) {
      int j;
      for(j=0;j<n/4;j++) {
        LUT1[i][j] = W(n,j);
      }
    }else{
      int j;
      for(j=0;j<n/4;j+=4) {
        data_t w1[4];
        int k;
        for(k=0;k<4;k++) w1[k] = W(n,j+k);

        LUT1[i][j]   = creal(w1[0]) + creal(w1[1])*I;
        LUT1[i][j+1] = creal(w1[2]) + creal(w1[3])*I;
        LUT1[i][j+2] = cimag(w1[0]) + cimag(w1[1])*I;
        LUT1[i][j+3] = cimag(w1[2]) + cimag(w1[3])*I;
      }
    }
  }
  TN = N;

}
```