# Asymmetrical Load-Balancing for Incremental Fast Fourier Transform on Multi-Core Processors

By

**Todor Pandeliev**

A thesis submitted to
The Faculty of Graduate Studies and Research
in partial fulfilment of
the degree requirements of
**Master of Science in Information and Systems Science – Computer Science**

Ottawa-Carleton Institute of
Computer Science

Department of Computer Science
Carleton University
Ottawa, Ontario, Canada
September, 2009

*Your file Votre référence*
*ISBN:* 978-0-494-64431-7
*Our file Notre référence*
*ISBN:* 978-0-494-64431-7

# Abstract

The Fast Fourier Transform (FFT) is a powerful method in contemporary computing, with lots of practical applications – from signal processing to cryptography. On multi-core platforms, symmetrical load distribution prevails but has efficiency issues with locality creating a gap between theoretical arithmetic complexity and actual performance.

Some proposed re-evaluations of Amdahl's law for parallel speed-up favour asymmetric multi-processing. The approach taken in this work is therefore to let the individual processors/cores specialise in parts of the FFT such as butterfly operation, permuting/transposing, calculating complex roots of unity; thus computation is asymmetrical even on SMP/CMP. Inter-thread synchronisation employed is spin-lock. This research contributes: the notion of incrementality inherent in the application, innovative usage of a shared heap to hold twiddle-factors, and best known arithmetic complexity of computing these. The solution suits hard to predict problem sizes, on the up to 9 cores that the multi-core industry delivers nowadays (9 in IBM's CellBE).

TO THE EUROPEAN COUNTRY THAT RAISED ME ACADEMICALLY,
BULGARIA,
AND TO CANADA FOR ALLOWING ME TO EXCEL.

# Acknowledgements

I would specifically like to express appreciation to my co-supervisors Dr. Michiel Smid and Dr. Richard Dansereau, for their participation in the initial discussion of the topic, as well as for logistical support. Dr. Smid contributed a lot to shaping up this research through valuable concise remarks, to the point, all along throughout the work.

My family comprising my wife Valeria Pandelieva, son Velian and daughter Antonia, deserve thanks for their patience, understanding and encouragement during my studies at Carleton University.

The overall academic atmosphere at the School of Computer Science, and the excellent level of the seminars in particular, set the context that made this possible. Many thanks go to the School's graduate director (until recently) Dr. J.-P. Corriveau for his support in my transition into the department, and to Claire Ryan for her assistance with admin matters. Dr. Kranakis contributed to improving the clarity of the narrative.

Dr. Panario from the Department of Mathematics helped to fix and improve the mathematics in this work.

# Table of Contents

# List of Figures

# List of Acronyms

ADC.....Analog-to-Digital Converter

AM.....Amplitude Modulation

ALU.....Arithmetical-Logical Unit

AMP.....Asymmetrical Multi-Processing

API.....Application Programming Interface

BCE.....Base Core Entity

CMP.....Chip Multi-Processing (SMP on a single chip)

CPU.....Central Processing Unit

DMA.....Direct Memory Access

DFT.....Discrete Fourier Transform

DIF.....Decimation In Frequency

DIT.....Decimation In Time

DMA.....Direct Memory Access, copying without processor participation

DSP.....Digital Signal Processing/ Digital Signal Processor

FFT.....Fast Fourier Transform

FFTW.....Fastest Fourier Transform in the West (a planner/executor tool)

FLOP.....FLoating-point OPeration

FM.....Frequency Modulation

FPGA.....Field-Programmable Gate Array

IDCT.....Inverse Discrete Cosine Transform

IFFT.....Inverse Fast Fourier Transform

MAC.....Multiply-and-ACcumulate

MKL.....Math Kernel Library by Intel

OFDM.....Orthogonal Frequency Division Multiplexing

PPU.....PowerPC core on the IBM CellBE processor

PVR.....Point-Value Representation (of polynomials)

SIMD.....Single Instruction Multiple Data

SMP.....Symmetrical Multi-Processing

SPU.....Synergistic Processing Unit core on the IBM CellBE processor

WHT.....Walsh-Hadamard Transform

WLOG.....Without Loss Of Generality

# Chapter 1: Introduction

## 1.1 Terminology Used

Multi-core processor technology is a modern version of parallel multi-processing, in which several but not numerous, usually a single-digit number of cores are laid out by industry on the same processor. One of the aims is to distribute the generated heat more evenly across the integrated circuit, so that higher performance is achieved without increasing the processor's clock rate, which inevitably creates heat dissipation issues.

In the title of this work, "asymmetrical load-balancing" is used in a sense similar to the well-known meaning in AMP (Asymmetrical Multi-Processing). AMP refers to processor designs with cores consisting of different/specialised architectures, as well as to running different applications on each core – possibly optimised for different hardware features, but also because each core is assigned its own thread. Symmetrical on the other hand, when referring to hardware, means that the processor cores are identical. When used in conjunction with Operating Systems/Software, this usually means that the different cores do not run the same code (on different data) in parallel – a multi-core variant of vectorization and SIMD, which some compiler code-generation optimisers and libraries like Intel's MKL are capable of achieving transparently for the programmer.

The title uses the word "incremental" to indicate that not all input data may be available at once at the start of the computational process. Indeed most signal-processing applications, FFT being part of these, rely on sampling voltage at regular intervals.

Except for a couple of newly introduced terms e.g. "middle-heap", the rest of the wording in this work is intended to conform completely to the widely-accepted terms and their meanings in contemporary literature on technology. The only peculiarity worth mentioning is that sometimes terms from the area of algorithms and computer science are less known to engineering and electronics people, e.g. "heap", and vice-versa. Among the possible stumbling blocks for people outside electrical and electronics engineering could be DMA (Direct Memory Access) – copying of values in memory without the participation of the processor; DSP (Digital Signal Processing/Digital Signal Processor) denoting a serially manufactured but specialised processor for computationally-intensive, usually embedded, applications. Along the same lines are FPGA (Field-Programmable Gate Array) – an integrated circuit whose logic is programmable rather than hard-wired, MAC (Multiply-and-Accumulate) – a computational pattern in DSP technology.

There are also terms that are known to any professional in the industry of software development, but may not be very commonly used by academic researchers. Among these is API (Application Programming Interface) meaning a function/functions to be called, or "spin-lock" – computationally efficient but crude inter-thread synchronisation where one core is in an unproductive loop while waiting to be released by another.

Key to this work is also the notion of space locality meaning the aim to achieve few cache misses – see section 1.2.4, and time locality, explained in detail in section 2.3.1.

## 1.2 State of the Art in FFT Efficiency on Multi-core

### 1.2.1 Amdahl's law

Almost forty years ago in [3] Gene Amdahl argued in favour of a single-processor approach for achieving large-scale computing capabilities, as opposed to multiprocessing. In the process, he defined his law for the case of using $n$ processors (cores) in parallel. Assuming that fraction $p$ of a program's execution time is parallelisable (ignoring scheduling overhead), while $1-p$ is strictly sequential, the speedup on $n$ processors is:

$$S_{parallel} = \frac{1}{(1-p) + \frac{p}{n}}$$

Amdahl's law has a few corollaries, one of which, namely that when $p$ is small optimisations have little effect, was in support of his argument that high performance computing should rely on single processors. The most important corollary is this: as $n$ approaches infinity, speedup is bound by $1/(1-p)$.

### 1.2.2 Gustafson's law

In 1988 [11] was published, and later became known as the Gustafson(-Barsis) Law. Here is how he himself summarises it: "The model is not a contradiction of Amdahl's law as some have stated, but an observation that Amdahl's assumptions don't match the way people use parallel processors. People scale their problems to match the power available, in contrast to Amdahl's assumption that the problem is always the same no matter how capable the computer." For $N$ processors, the parallel part $p$ will scale to $p' \times N$:

$S_{scaled} = (s'+p' \times N)/(s'+p') = s'+p' \times N = N + (1-N) \times s' = N - (N-1) \times s'$, where $s' = 1-p'$.

As a result, few today claim that parallel processing is not viable.

In [12] Hill and Marty argue two important results (among others), quoted exactly:

**Result 1.** Amdahl's law (still) applies to multicore chips because achieving the best speedup $S$ requires $p$ to be close to 1. Thus, finding parallelism is still critical.

**Result 2.** Asymmetric multicore chips can offer potential speedups that are much greater than symmetric multicore chips (and never worse).

Since by asymmetric they mean fitting a varying number of Base Core Elements (BCEs) of the *same*, not different, architecture in each core, in this work we adopt the idea of asymmetric loads for CMP. A spin-lock is an efficient idle loop in one core until another core is ready and releases it. While the ad-hoc load-balancing will not be perfect, the simplicity of spin-locks will minimise the inherently serial synchronisation overhead.

This section further focuses on literature covering the particular task of optimising the FFT and/or similar computational problems, with respect to locality and parallelism.

### 1.2.3 Input/Output Complexity

Efficient algorithms have to consider the use of slower memory and external storage, along with the counts of operations. While today memory has even more layers, taking into account cache at various levels, some earlier results on efficiency of algorithms still apply – whether cache versus main memory is considered, or main memory versus disk can be immaterial. In [1] Aggrawal and Vitter consider a model with these parameters:

- $N$ = # records to handle;
- $M$ = # records that can fit into internal memory;
- $B$ = # records that can be transferred in a single block;
- $P$ = # blocks that can be transferred concurrently;

where $1 \le P \le M < N$ and $1 \le P \le \lfloor M/B \rfloor$. The parameters *N, M,* and *B* are the file size, memory size, and block size, respectively.

For FFT, the asymptotically-optimal algorithm based on Radix-2 DIF recursion, is shown to require $\Theta\left(\frac{N}{PB}\frac{\log(1+N/B)}{\log(1+M/B)}\right)$ I/O operations (no tight lower bound is known). This is achieved through so called pebbling, and bringing the records into memory in transposition permutations, *M* at a time in log*N*/log*M* stages (assume log*M* divides log*N*).

### 1.2.4 Cache Obliviousness

The ideal cache model is defined as follows: the CPU only uses words that are in cache; if the referenced word is already in cache, a cache hit occurs, and the word is used; else a cache miss causes a fetch from memory, possibly "optimally" evicting from the cache. Cache normally consists of cache lines, each containing *L* consecutive words copied together to and from main memory, $L>1$ – counting on data space locality for efficiency. Algorithms are cache oblivious when no parameters dependent on the hardware platform, such as cache size or cache-line length, need tuning to achieve asymptotical optimality.

In their landmark paper [9] Frigo et al. prove the following for FFT: with cache of size *Z* and cache-line length *L*, when $Z = \Omega(L^2)$ – defined as the tall cache assumption, their 6-step version (see section 2.3) of the *n*-point FFT with factorisation $\sqrt{n} \times \sqrt{n}$ is cache-oblivious (if transpose is cache-oblivious), and incurs $\Theta\left(1+\frac{n}{L}(1+\log_Z n)\right)$ cache misses. They also prove that cache-obliviousness is preserved with multiple cache levels.

### 1.2.5 Reconciling Space and Time Locality with FFT

From among the planner-oriented solutions, the most unique one is in [16] , and uses a learning strategy and heuristics. The authors Singer and Veloso describe a space of different decimation trees for a given FFT, using Kronecker products with permutation matrices and twiddle-factor/Vandermonde matrices. They maintain that the complexity of modern processors makes it difficult to predict analytically, or to model by hand, the performance of a formula on a particular architecture. Also, that the differences between current processors lead to very different optimal formulae from machine to machine. They employ a black-box approach by running the planner software tool described in [7] on different platforms, gathering performance statistics. Their research reveals clear clusters in the histograms of cache miss counts vs. runtimes for each platform, as well as interesting patterns common across the board. Also, they apply clever heuristics to limit the combinatorial burst of the solution space, and use particular decomposition trees. Their approach seems to focus on, and work slightly better for, the similar to DFT but real-valued, Walsh-Hadamard Transform (WHT) – outside the scope of this work.

Frigo and Johnson have addressed similar goals in the FFTW software (1998): it uses binary dynamic programming to search for the optimal FFT implementation (see [8] ).

In [2] Ali et al. address scheduling on $p$ cores, by factoring a $\sqrt{n} \times \sqrt{n}$-sized FFT, $p \mid \sqrt{n}$. Earlier, [2] co-author Johnsson contributed to the development of the popular planner tool UHFFT. Using it, the parallel FFT schedules in OpenMP and PThreads are compared to that of the best sequential FFT plan, and the speedup for various number of

processors is reported. Reasonable speedup is achieved for sizes between $2^{12}$ and $2^{14}$, on 2 to 8 cores. However outside those sizes the speedup seems to follow Amdahl's law.

In [7] Franchetti, Voronenko and Püschel discuss parallelising the FFT under the assumption of shared memory among the cores. They maintain: "The major problem with using the standard Cooley-Tukey FFT algorithm on shared memory machines is its memory access pattern: large strides, and consecutive loop iterations touch the same cache lines, which leads to false sharing." Their effort is thus aimed to fight false sharing. They describe their existing Spiral planner tool, then propose extensions to it that allow for embarrassingly parallel (i.e. no mutual data dependencies exist between the threads) computations of the FFT that also avoid false sharing. Their implementation appears to be the best fit for CMP: while on SMP platforms the performances are comparable, the "break-even point" of parallelised FFT for Spiral on CMP is at size $2^7$, while the competition (FFTW and Intel MKL) achieves it earliest at $2^{14}$.

An exotic algorithmic path is presented in [13] : van der Hoeven argues that the stride from $2^n$ to $2^{n+1}$ is too large, so truncate the FFT to obtain $<2^{n+1}$ entries in the result vector.

In [4] there is a good summary and bibliography of techniques for efficient twiddle-factor computation. The main approaches are CORDIC algorithms, polynomial approximation of trigonometry, and the recursive sine-function generator technique. CORDIC implements fixed-point arithmetic for butterfly rotation (which is what a multiply by a twiddle-factor is) in fast embedded/FPGA systems, virtually eliminating the need to compute and store twiddle-factors separately; polynomial approximation is common in digital frequency synthesis (DDFS); recursive sine-function generation has

accuracy issues, which the authors of [4] attempt to counter. The best result quoted is with the recursive sine-function generator – 2 adds and 2 multiplies, 4 FLOPs per entry.

Finally, attempts are made to design new computing platforms so that they are also optimised for applications similar to the FFT. In [10] Guo et al. elaborate on desirable features of a universal multi-core processor with respect to its memory interface. Although their requirements are not explicitly stated to favour the FFT, 2 of the 7 tests carried out on their simulation are FFT and the essentially computationally equivalent Inverse Discrete Cosine Transform (IDCT). Another two are Finite Impulse Response filter and Adaptive Differential Pulse-Code Modulation coder - both signal processing applications too, making these more than half of the tested ones. The most revolutionary idea proposed is the usage of cache for instructions only, while data goes into local memory for each core, similar to the CellBE processor (except the latter uses local memory for instructions too).

## 1.3 Motivation for This Research

### 1.3.1 Incrementality of Important Applications

Most signal processing applications involve sampling at regular intervals. Prevailing research so far has concentrated on quickly and otherwise efficiently computing the FFT, once all input data is in main memory (all samples have been taken).

Interestingly, the twiddle factors, constant for any given problem size, are almost never kept in storage or a database (except in some embedded/FPGA applications), even when the algorithm only deals with power-of-two sizes. Instead, API is provided to

calculate them, the idea being that the programmer calls this once then uses the values in repeated Fourier transforms of the same size.

Herein we assume that there exist signal processing applications where the problem size is not known before the arrival of some samples. While this may not always be the case, it seems like a terrible waste anyway to stay idle computationally while sampling, and until the last sample arrives. If the latter were to contain a sentinel tag identifying it as the last (a natural assumption), even the twiddle factors will not yet have been computed, which could have happened if the problem size had been known in advance.

We present a parallelisable method to work incrementally, as the samples arrive.

### 1.3.2 Multi-core specifics

What remains to be addressed is parallelisation. Earlier research has exclusively addressed it via decimation of FFT into smaller sizes, each assigned to a separate core. That may be the only reasonable approach when many parallel processor units are available. However recently multi-cores of 2 to maximum 8 units have become popular. On them, the prevailing approach is to search experimentally a space of (usually six-step) solutions looking for the best execution time of a specific size FFT on a particular platform.

It would be interesting to explore other load-balancing strategies (but not excluding decimation, if enough cores are available). Pease's algorithm ([15] ) is a candidate, with the perfect shuffles running parallel on a separate core, as well as parts of the butterfly operation. Cores could progress in parallel performing asymmetrical computation. This allows for the simplest and most efficient synchronisation mechanism i.e. spin-lock.

## 1.4 Goals of This Research

### 1.4.1 Approach and Methodology

It is only natural to explore computing of the FFT efficiently, in the following sense:

- the overall work complexity is asymptotically optimal i.e. $O(N \log N)$ (research exists that argues in favour of Horner's rule, $O(N^2)$, for practical low $N$);
- the arithmetic complexity is at least as good as that of the original Cooley-Tukey algorithm, preferably the more efficient Split-radix (no lower bound is known);
- the I/O complexity in cache misses is close to optimal (no lower bound is known), asymptotically; assume the size fits in memory but not cache (not hard up to $2^{32}$);
- the algorithm is cache-oblivious;
- SIMD, MAC and other pipelining features of modern CPUs are used effectively;
- the work is parallelisable and reasonably load-balanced on available CMP multi-core, for the customary numbers of cores – up to the 9 present in IBM's CellBE.

At the current stage of the research area it is unclear whether the shopping list above is achievable in full, and under what assumptions – even if a dedicated chip is designed. Actually Pease ([15] ) originally suggested his algorithm for the design of dedicated hardware. The top two bullets are easiest to achieve – a variety of decimation strategies are known, for highly composite sizes or powers of two, as well as for prime sizes and co-prime factors of the size. The next three ones have been researched mostly separately.

The last bullet looks deceptively easy, but is not trivial if the rest are kept in mind.

A sizable proportion of the recent papers is dedicated to planners that find an optimal solution, for a given problem size, on a particular platform, from a space of possible ones. For this to be practical, the application is assumed to be of known size, and once

optimised will be run multiple times on the same platform. Although these assumptions are fair, other applications are also conceivable in signal processing and measurement.

We take an integrated approach: consider all adjacent areas as well as the expected timing of events around the FFT computation. We also suggest asymmetrical execution of the parts e.g. shuffles, butterflies and twiddle-factor multiplication, on separate cores. Thus each core will run a simpler algorithm, which is a prerequisite for better pipelining and compiler optimisations.

### 1.4.2 Specific Solution for Computation of Twiddle-factors

We address the problem of finding even more efficient ways to compute the twiddle factors than the ones already known, specifically for power-of-two sizes.

We show a high-accuracy method (data structure and algorithm) to improve on the adjacent area of computing the twiddle factors; our new method also perfectly agrees with incrementality.

### 1.4.3 Specific Solutions for Incrementality

In FFT every input value of the algorithm affects every output value. Can we still save time under the assumption that the application is incremental?

Our answer is yes – if we choose to re-order data to improve space locality (and to simplify the algorithm), that (re-ordering) work is already half done at every power-of-two boundary. Then we come up with the notion of input insertion, whereby an input value arriving after a truncated FFT has already been computed, is propagated into the solution, instead of doing the same power-of-two size FFT all over again.

# Chapter 2: Introducing the Fast Fourier Transform

## 2.1 Basic definitions

### 2.1.1 Polynomials

A polynomial in the variable $x$ over an algebraic field $F$ is $A(x) = \sum_{j=0}^{n-1} a_j x^j$, $a_j \in F$.

The values $a_0, a_1, \ldots, a_{n-1}$ are called the coefficients of the polynomial, typically drawn from the field $C$ of the complex numbers. Any integer that is strictly greater than the degree of a polynomial is a degree-bound of that polynomial. The degree of a polynomial of degree-bound $n$ may be any integer between 0 and $n-1$, inclusive.

If $A(x)$ and $B(x)$ are polynomials, their sum is defined as a polynomial $C(x)$ of same degree-bound, such that $C(x) = A(x) + B(x)$, for any $x$. The coefficients of matching degrees are added together – computational complexity is $O(n)$ for degree-bound $n$.

Similarly, if $A(x)$ and $B(x)$ are polynomials of degree-bound $n$, their product $C(x)$ is a polynomial of degree-bound $2n-1$ such that $C(x) = A(x).B(x)$ for any $x$. This means multiplying each term in $A(x)$ by each term in $B(x)$ and adding those with equal powers; this is called the convolution of the input vectors $a$ and $b$, denoted $c = a \otimes b$. The above process is $O(n^2)$, counting all arithmetic operations on the terms' coefficients.

### 2.1.2 Alternative Representations of Polynomials

The representation from the definition is called the coefficient representation. It is convenient for some operations on polynomials, e.g. addition as above. Also, the operation of evaluating the polynomial $A(x)$ at a given point $x_0$ consists of computing the value of $A(x_0)$. Evaluation takes time $\Theta(n)$ using Horner's rule: $A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \cdots + x_0(a_{n-2} + x_0(a_{n-1}))\ldots))$.

A point-value representation of a polynomial $A(x)$ of degree-bound $n$ is a set of $n$ point-value pairs $\{(x_0, y_0), (x_1, y_1),..., (x_{n-1}, y_{n-1})\}$ such that all of the $x_k$ are distinct and $y_k = A(x_k)$, for $k = 0, 1,.., n-1$. A polynomial has many different point-value representations; evaluation means finding one of these, of size at least the degree-bound.

The point-value representation is as convenient for multiplying polynomials, as for adding them. If $C(x) = A(x)\,B(x)$, then $C(x_k) = A(x_k)\,B(x_k)$ for any point $x_k$, and we can point-wise multiply a point-value representation of $A$ by a point-value representation of $B$ to obtain a point-value representation of $C$.

The inverse of evaluation – the determining of the coefficient form of a polynomial from a point-value representation – is called interpolation.

**Theorem** (Uniqueness of interpolating polynomial): For any set $\{(x_0, y_0), (x_1, y_1),..., (x_{n-1}, y_{n-1})\}$ of $n$ point-value pairs such that all the $x_k$ values are distinct, there is a unique polynomial $A(x)$ of degree-bound $n$ such that $y_k = A(x_k)$ for $k = 0, 1,..., n-1$.

The proof is based on the existence of the inverse of the Vandermonde matrix

$$V(x_0, x_1, ..., x_{n-1}) = \begin{pmatrix} 1 & x_0 & x_0^2 ... & x_0^{n-1} \\ 1 & x_1 & x_1^2 & x_1^{n-1} \\ ... & ... & ... & ... \\ 1 & x_{n-1} & x_{n-1}^2 ... & x_{n-1}^{n-1} \end{pmatrix}$$ – non-singular as $x_k$ are distinct by definition.

A fast algorithm for $n$-point interpolation is based on ***Lagrange's formula***: $A(x) = \sum_{k=0}^{n-1} \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)}$. It is possible to compute the coefficients of $A$ using Lagrange's formula in time $\Theta(n^2)$: compute $P = \prod_{j \neq k} (x_k - x_j)$, then the coefficient representation of $\prod_{j \neq k} (x - x_j)$, then divide it by $(x - x_l) \prod_{j \neq k} (x_k - x_j)$ for each $l = 0, 1, ..., n-1$.

Thus, $n$-point evaluation and interpolation are well-defined inverse operations that transform between the coefficient representation of a polynomial and a point-value representation. The methods described for these are $\Theta(n^2)$ classical arithmetic operations, where $n$ is the degree bound of the polynomial.

### 2.1.3 Signals and Frequencies

Electronics mixes signals by adding them or multiplying/dividing them.

Common practical problems in signal processing involve analysis of the spectrum: if frequency $f$ is present, how strong is $2f$, $3f$, etc (called harmonics). Note: a perfect square wave of a digital signal, alternating between some voltage to represent a binary 1, and ~0V to represent a binary 0, contains an infinite series of $2f$, $3f$, $4f$,...!

This means finding the coefficients of terms of a polynomial (interpolation), since $2f$ is represented by the square of a complex number, $3f$ – by the cube, etc. This is based on Euler's identity: $\cos x + i \sin x = e^{ix}$, as the equality $e^{ky} = (e^{y})^{k}$ holds for any complex $y$.

### 2.1.4 Discrete Fourier Transform (DFT)

The inverse of the particular interpolation when $f = 1/n$ is to evaluate the polynomial $A(x) = \sum_{j=0}^{n-1} a_j x^j$ of degree-bound $n$ at the $n$ complex $n^{\text{th}}$ roots of unity: $\omega_n^0, \omega_n^1, ..., \omega_n^{n-1}$, where $\omega_n = e^{2\pi i/n}$. Without loss of generality (WLOG), assume that $n$ is a power of 2, since a given degree can always be raised – high-order zero coefficients can always be added as necessary. Let $A$ be given in coefficient form $a = (a_0, a_1, ..., a_{n-1})$. Define the results $y_k$, for $k = 0, 1, ..., n-1$, by $y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$.

**Definition 1**: The vector $y = (y_0, y_1, ..., y_{n-1})$ is the Discrete Fourier Transform (DFT) of the coefficient vector $a = (a_0, a_1, ..., a_{n-1})$. We also write $y = \text{DFT}_n(a)$.

The DFT is equivalent to the continuous Fourier transform $F(s) = \int_{-\infty}^{\infty} f(t) e^{2\pi i . st} dt$, for periodic $f$, in a band-limited setting – made discrete via sampling.

$|F(s)|$ would yield the "strength" of frequency $s$ in the mix – its amplitude. A different way to express the DFT is $y = V_n a$, where $V_n$ is the Vandermonde matrix with the powers of the $n$-th complex root of unity. To carry out the inverse, e.g. interpolate for frequency analysis, $V_n^{-1}$ is needed. In [6] there is the following theorem:

**Theorem**: $V_n^{-1}{}_{j,k} = \omega_n^{-kj/n}$

Proof idea is, by using the properties of the complex roots of unity, to show that $V_n^{-1} V_n$ is the identity matrix $I_n$. The above shows the inverse to be similar to DFT. DFT has been proven to be its own inverse, reordered and scaled with a factor of $1/n$.

### 2.1.5 Properties of the Complex Roots of Unity

The description of these properties follows the presentation in [6] . From the definition of complex roots of unity directly follows the following lemma:

**Lemma 1**: (Cancellation lemma)

For any integers $n \geq 0$, $k \geq 0$, and $d > 0$, $\omega_{dn}^{dk} = \omega_n^k$.

**Proof**: $\omega_{dn}^{dk} = e^{2\pi i.dk/dn} = e^{2\pi i.k/n} = \omega_n^k$.

**Corollary**

For any integer n > 0, $\omega_{2n}^{n} = \omega_2 = -1$.

**Lemma 2**: (Halving lemma)

If n > 0 is even, then the squares of the $n$ complex $n^{\text{th}}$ roots of unity are the $n/2$ complex $(n/2)^{\text{th}}$ roots of unity (each occurring twice).

**Proof**: $\omega_n^{n/2} = -1$ implies $\omega_n^{k+n/2} = -\omega_n^k$ hence $(\omega_n^{k+n/2})^2 = (\omega_n^k)^2$. However per the Cancellation lemma $(\omega_n^k)^2 = \omega_{n/2}^k$.

**Lemma 3**: (Summation lemma)

For any integer $n \geq 1$ and nonnegative integer $k$ not divisible by $n$, $\sum_{j=0}^{n-1} (\omega_n^k)^j = 0$.

**Proof**: $\sum_{j=0}^{n-1} (\omega_n^k)^j = \frac{1-(\omega_n^k)^n}{1-\omega_n^k} = \frac{1-(\omega_n^n)^k}{1-\omega_n^k} = \frac{1-(1)^k}{1-\omega_n^k} = 0$.

### 2.1.6 Fast Fourier Transform (FFT) and the Cooley-Tukey Algorithm

By taking advantage of the properties of the complex roots of unity, $\mathrm{DFT}_n(a)$ can be computed in time $\Theta(n \log n)$, as opposed to $\Theta(n^2)$ for the definition formula. This method is attributed to Gauss, but was rediscovered as the Cooley-Tukey Algorithm in [5] .

In the two cases below we use $(\omega_n^k)^2 = \omega_{n/2}^k$ (remember WLOG $n$ is a power of 2).

**Idea 1:** Split the even-index from the odd-index coefficients of polynomial $A(x)$. Assuming $n$ even: $A^{[0]}(x) = a_0 + a_2x + a_4x^2 + \ldots + a_{n-2}x^{n/2-2}$; $A^{[1]}(x) = a_1 + a_3x + a_5x^2 + \ldots + a_{n-1}x^{n/2-1}$, and $A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2)$. Thus we need to evaluate two degree-bound $n/2$ polynomials at $(\omega_n^0)^2$, $(\omega_n^1)^2$, …, $(\omega_n^{n-1})^2$ – the complex $(n/2)^{\text{th}}$ roots of unity each occurring twice – then to combine the results. The problem decomposes into two of half its size. This method is referred to in literature as Radix-2 decimation in time (DIT).

The inverse of this odd/even split is an operation known as perfect shuffle (like with two half-decks of cards).

**Idea 2:** Split the low-index half coefficients of $A(x)$ from the high-index half ones:

$$A(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{n/2-1}x^{n/2-1} + x^{n/2}(a_{n/2} + a_{n/2+1}x + a_{n/2+2}x^2 + \ldots + a_{n-1}x^{n/2-1})$$

i.e. $A(x) = \sum_{j=0}^{\frac{n}{2}-1}(a_j + x^{n/2}a_{j+n/2})x^j$ ; thus for $r = 0,1,\ldots,n-1$, $y_r = \sum_{j=0}^{\frac{n}{2}-1}(a_j + \omega_n^{nr/2}a_{j+n/2})\omega_n^{jr}$ .

Consider the even $r=2k$ separately from the odd $r=2l+1$; taking into account $\omega_n^{n/2} = -1$:

$$z_k = y_{2k} = \sum_{j=0}^{\frac{n}{2}-1}(a_j + a_{j+n/2})\omega_n^{2kj} = \sum_{j=0}^{\frac{n}{2}-1}(a_j + a_{j+n/2}).\omega_{n/2}^{kj}, \text{ for } k = 0,1,\ldots,n/2-1, \text{ and}$$

$$t_l = y_{2l+1} = \sum_{j=0}^{\frac{n}{2}-1}(a_j - a_{j+n/2})\omega_n^{2lj}\omega_n^j = \sum_{j=0}^{\frac{n}{2}-1}(a_j - a_{j+n/2})\omega_n^j.\omega_{n/2}^{lj}, \text{ for } l = 0,1,\ldots,n/2-1.$$

Again, the DFT problem decomposes into two of half its original size. This method is referred to as Radix-2 decimation in frequency (DIF).

Radix-2 DIT and DIF are particular cases of the general Cooley-Tukey algorithm, which allows any radix that divides *n*.

### 2.1.7 Recursive Algorithm for Radix-2 DIT FFT

The pseudo-code below follows DIT literally, hence its correctness is inherent: it begins with a check for the end of the recursion; then an inverse perfect shuffle on the input is performed and the result is assigned to new 0-based vectors a0[] and a1[]. Recursive_FFT() then calls itself for these. Finally, a for-loop iterates incrementally calculating the powers of the root of unity at the same time combining a0[] and a1[].

```
Recursive_FFT(a[0:n-1], n) /* a[ ] is a vector, n is power of 2 */
    if n = 1 then
     return a[];
    endif;
    a0[0:n-1] = {a[0],a[2],...,a[n-2]}; /*vector assignment, an*/
    a1[0:n-1] = {a[1],a[3],...,a[n-1]}; /*inverse perfect shuffle*/
    y0[0:n-1] = Recursive_FFT(a0,n/2);
    y1[0:n-1] = Recursive_FFT(a1,n/2 );
    ωn = e^(2πi/n) ; /* primitive complex root of unity – twiddle-factor */
    ω = 1;
    for k = 0 to n/2-1 do
     y[k] = y0[k] + ω*y1[k]; y[k+n/2] = y0[k] - ω*y1[k];
     ω = ω * ωn; /* computation of twiddle-factors */
    endfor;
    return y;
```

In the computational algorithms, the complex roots of unity have become known as Twiddle-factors, since they are being viewed as coefficient corrections, e.g. in $t_l$ above.

The computational complexity of Recursive_FFT() is $O(n \log n)$. Indeed there are $\log n$ recursion levels of $n$ iteration loops each – $2 \times n/2$ then $4 \times n/4$, etc. We show in section 3.2.7 that the computational complexity of DIF versions of Cooley-Tukey is also $O(n \log n)$.

### 2.1.8 Towards Iterative Algorithm for Radix-2 DIT FFT

The code above is recursive, with overheads for the calls/returns and local vectors. Also, the value $\omega$`*yl[k]` is computed twice, when added and when subtracted. It could be assigned to a variable then reused with the sign reversed; this is known as a butterfly operation.

Follow the exact order of the recursive evaluation, indices expressed in binary:

```
  a[0] a[2] a[4] a[6]          a[1] a[3] a[5] a[7]

a[0] a[4]     a[2] a[6]     a[1] a[5]     a[3] a[7]

 000  100      010  110      001  101      011  111
```

The above are a case of bit-reversal permutation: binary numbers sorted by the least-significant (LS) bit first. They are easy to compute in sub $O(n \log n)$, thus would not affect the complexity. It is then straightforward to write an iterative version of the above algorithm, refer to [6] for suggested implementation.

### 2.1.9 Applications of the FFT

Possible application is digital implementation of AM or FM radio. Presented in Figure 2.1 (©Arrow Electronics) is Amplitude Modulation (AM) of carrier 14kHz by 1kHz. The 1kHz amplitude over time can be found (demodulated) via spectrum analysis using FFT.

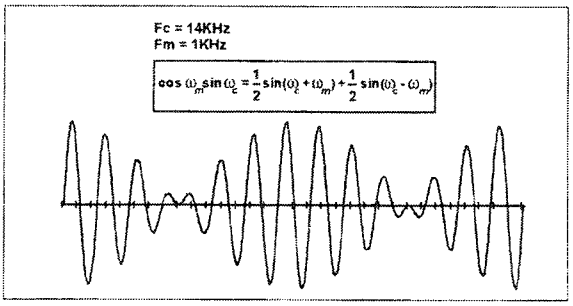



**Figure 2.1. Amplitude Modulation**

At a DSP training course we were presented with a software traffic radar detector too.

Coming back to polynomials, straightforward multiplication (convolution of coefficient vectors) was shown to be $O(n^2)$. Alternatively (per Schönhage & Strassen):

- Evaluate both polynomials at complex roots of unity, using FFT – $O(n \log n)$;
- Point-wise multiplication of the two PVRs – $O(n)$;
- Interpolate the result using inverse FFT – $O(n \log n)$;

Overall running time is therefore $O(n \log n)$. One important application is the efficient multiplication of large prime numbers needed in cryptography: the decimal representation of an integer can be viewed as a polynomial with its digits as the coefficients.

In modern wireless communication systems, both for voice and wideband data transmission, the OFDM (Orthogonal Frequency Division Multiplexing) is used. It also plays an important role in wire-line communication systems. Examples of widely popular standards relying upon it are 820.11a/g, 802.16, DVB, DAB, VDSL, and so on.

In these systems, DFT/inverse DFT, implemented as FFT/inverse FFT both in software and hardware, is the core component in OFDM transmission and reception. Those systems require FFT/IFFT of lengths ranging from 64 to 8192. Thus the FFT needs to be evaluated on the widest possible variety of computing platforms.

### 2.1.10 FFT on Multi-core Platforms

The majority of today's computers are multi-core – with more than one processor/ arithmetical-logical unit (ALU) on a single chip. Many use truly shared memory that can be accessed from any core, some like IBM's CellBE do not – their data needs to be copied by Direct Memory Access (DMA). Some use private cache with shared memory, hence the false sharing problem – data may be present in the wrong core's cache, where it is not needed but takes the space of data that is needed for the computation.

## 2.2 Theoretical Arithmetic Complexity of the FFT

Asymptotically FFT for size $N$ takes $O(N \log N)$ operations. Extensive research has been carried out on the actual operations count, i.e. on the constant factor before $N \log N$ (known to be 5 for Cooley-Tukey), as well as the remaining (non-dominating) terms of the complexity equality. The arithmetic complexity is expressed in the number of floating point operations (FLOPs), additions and multiplications, as a function of the problem size $N$. No tight lower limit is known. Until recently (2007), the best known result had been achieved by Yavne in 1968, his split-radix algorithm running in $4N\lg N - 6N + 8$, where lg means $\log_2$. In [14] Johnson and Frigo give an explanation of that result, and publish an improved count of $34/9N\lg N - 124/27N - 2\lg N - 2/9(-1)^{\lg N}\lg N + 16/27(-1)^{\lg N} + 8$,

also based on the split-radix method: decimation into 3 sub-problems, one of which consists of the even-indexed terms, the other two respectively indexed 1 and 3 modulo 4.

There is also research available that focuses on minimising the floating-point multiplications, but this is achieved with a lot more additions. Such results could be useful on some platforms, particularly dedicated signal-processing FPGAs or processors (including some DSPs) that do not support floating point arithmetic in hardware. For normal processors, especially more recent ones, the arithmetic complexity in the original sense (multiplications and additions together) is more relevant since a multiplication takes the same number of processor cycles as addition. However experiments have shown that performance for the same theoretical FLOP count can differ dramatically depending on data space locality, pipelining, and other features of modern processors. This is the topic of the next section.

The twiddle-factors are mostly assumed (efficiently) pre-computed then reused. Efficient computation of these has focused mainly on avoiding repeated calculations and using some properties of the complex roots of unity like their periodicity: e.g. it is sufficient to calculate the ones in the 1$^{st}$ Cartesian quadrant – the others are obtained via multiplying by $i$, $-1$ or $-i$. These multiplications do not involve FLOPs, as they can be expressed as sign inversions and swapping of the real part with the imaginary part.

It is convenient to calculate the twiddle factors once, store the values and reuse them with new FFTs of the same size. Almost all existing software libraries provide API to populate an array of twiddle-factors given the FFT size, instead of pre-computed tables.

## 2.3 Decimation, Parallelism and Multi-core

Both DIF and DIT allow divide-and-conquer parallelisation, also apply to size factors ≠2, and algorithms exist for prime size (Rader's) or co-prime size factors (Good-Thomas). For large numbers of processors, e.g. on computing arrays or hyper-cubes, strategies have been developed to distribute work equitably (load-balancing). On smaller CMP multi-cores, efforts have been applied mostly to improve the use of caches and pipelines instead – the bit-reversal permute and the stride $a_0$ to $a_{n/2}$ create a problem in the presence of cache: they dramatically decrease speed for FFT sizes that do not fit in it. One solution is the Six-step approach ([9] , [7] ): decimate size $n=pq$ into a $p$x$q$ matrix, each row with coefficients of a size-$q$ FFT, and compute the original FFT in these steps:

1) Transpose – more cache-efficient due to stride smaller than $n/2$;
2) Perform FFT by rows – more cache-efficient due to size much smaller than $n$;
3) Combine results with twiddle-factors – of much lower degree;
4) Transpose;
5) Perform FFT by rows, this combines the parts of the decimation;
6) Transpose.

Most known modern solutions involve techniques to address both parallelism and locality. Some achieve it through a dedicated planner run to find one efficient solution from among a space of many, for a particular problem size and (multi-core) platform.

### 2.3.1 Pipelines and Single Instruction/Multiple Data (SIMD)

Space locality is not the only feature to be considered on modern computing platforms. Many, especially the ones dedicated to efficient computations such as DSPs, have been optimised for certain predictable patterns of instructions and data that occur in time.

For example a common characteristic is the efficient carrying out of a sequence of multiply-and-accumulate (MAC) into a sum $s$: $s \leftarrow s + a_i \,.\, b_i$, for a sequence of values $i$, or the ability to perform with higher efficiency the same operation on a relatively small array/vector of values – vectorization a.k.a. Single Instruction Multiple Data (SIMD).

Another one is instruction pipelining: phase 2 of instruction #$i$ runs in parallel to phase 1 of instruction #$i$+1, so circuitry parts don't wait idle – similar to a conveyor belt.

Sometimes in research the term of time locality is used to denote any of the above.

### 2.3.2 Stockham Autosort and Pease's Algorithm

Certain algorithms have been proven to fit well with space or time locality. One favoured method with SIMD is the Stockham Autosort (1966). Alternatively to the 6-step, it embeds the reverse-bit permute into the butterfly: when computing an FFT decimation stage, the results go into locations of an intermediate working array, determined by transposing (flipping) bit positions in the index's binary representation. Stockham is called auto-sort since no separate re-ordering/transpose step is required.

Pease's algorithm (1968) promises even greater SIMD advantages but requires a separate perfect-shuffle permute stage and $O(N \log N)$ auxiliary storage. In [15] he introduces Kronecker product notation to express any permute via matrix transpositions.

Stockham efficiency deteriorates dramatically when size exceeds $1/3^{rd}$ of the cache; indeed apart from the input array, for each phase it needs two alternating working storage arrays of the same size as the input array, plus more for temporary variables etc.

# Chapter 3: Generic Design

This chapter introduces our approach to the solutions, and describes these in detail.

## 3.1 The Solutions

### 3.1.1 Introducing the Middle-Heap

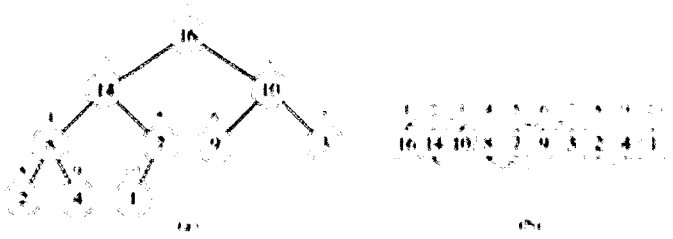
Coming back to twiddle factors, we accept that it is unwise to store them permanently, for all possible problem sizes. We argue however that we can compute them incrementally for the most widely used sizes – powers-of-two. First define array $H_3$: $H_3 = \{-1; i, -i\}$; at step $n>1$, using array of size $2^n$-1, compute the $2^{n+1}$-1 array's entries of indices $2^n$ to $2^{n+1}$ (the odd powers of $\omega_{2^{n+1}}$) – possible to do parallel to sampling. So $H_7$ =

$$=\{-1; i,-i; \omega_8 = \frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}i, \omega_8^{\,3} = -\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}i, \omega_8^{\,5} = -\frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}i, \omega_8^{\,7} = \frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}i\}, \text{ etc.}$$

The above requires a matching data structure: consider a tree that grows by doubling its size with each new level added. It is most appropriate to use a heap (see [6] ) since the underlying storage construct is an array.

The heap is a binary tree completely filled with elements, except possibly at the level of the leaves, stored in an array. The root is stored at index 1 of the array; if a parent has index $i$, its left child has index $2i$, and its right child has index $2i$+1. Each element is, or

has a key value from a partially or totally ordered set. When all parents are larger than their children, the heap is a max-heap. Here is an example from [6] :



A (max-) heap viewed as (a) a binary tree and (b) an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

**Figure 3.1. Max-Heap**

Similarly, a min-heap is the same structure but parents are smaller than their children.

**Definition 2**: An array $H$ of size $N$ in which for any natural $i \leq N$ the following holds: if $2i \leq N$ then $H[i] \leq H[2i]$, and if $2i < N$ then $H[i] \leq H[2i+1]$, is called a min-heap. When either of them exists within the array $H$, $H[2i]$ is defined as the left child of $H[i]$, and $H[2i+1]$ is defined as the right child of $H[i]$ in the tree of min-heap $H$; $H[i]$ is by definition the parent of $H[2i]$ and $H[2i+1]$. $H[1]$ is the tree root of min-heap $H$.

A min-heap $H$ is complete when its size $N$ is of the form $N = 2^n - 1$ for some $n > 1$.

Given the index $i$, the indices of its left child LEFT($i$), right child RIGHT($i$) and when the node is non-root of its parent PARENT($i$), are computed by pseudo-code as follows:

```
PARENT(i)
    return ⌊i/2⌋
LEFT(i)
    return 2i
RIGHT(i)
    return 2i + 1
```

For the complex roots of unity $\omega_n^p$, let partial order be ascending by degree $n$, and total order be by $n$ then (for two roots with equal degrees $n$) – ascending by $p$.

**Definition 3**: A complete min-heap of size $N = 2^n - 1$ containing all the twiddle-factors smaller than or equal to $\omega_{N+1}^N$ is called middle-heap of size $N$, denoted $H_N$.

We use only middle-heaps sorted per the total order defined above, as in Figure 3.2:

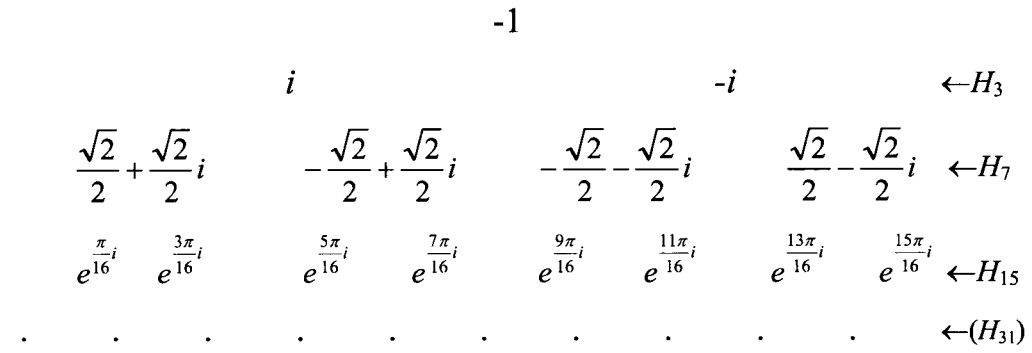



**Figure 3.2. Middle-Heap**

### 3.1.2 Incremental Computation of Twiddle Factors – the Mean-middle Method

When the $H_N$ middle-heap is in-traversed, the twiddle factors are returned in ascending powers of the $N$+1-th primitive root of unity, except the trivial one $\omega_{N+1}^{N+1}=1$. For example in $H_7$ (Figure 3.2), by starting from the left-most leaf $\frac{\sqrt{2}}{2}+\frac{\sqrt{2}}{2}i = e^{\frac{\pi}{8}i}$, then its parent $i = e^{\frac{2\pi}{8}i}$, followed by its sibling $-\frac{\sqrt{2}}{2}+\frac{\sqrt{2}}{2}i = e^{\frac{3\pi}{8}i}$, then up the tree until the root $-1 = e^{\frac{4\pi}{8}i}$ then down left the right sub-tree of the root to the leaf $-\frac{\sqrt{2}}{2}-\frac{\sqrt{2}}{2}i = e^{\frac{5\pi}{8}i}$, etc.

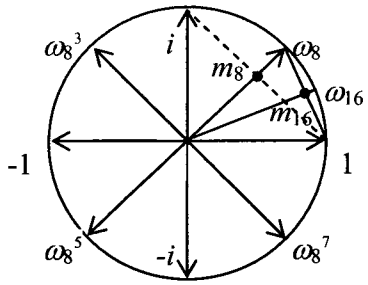
Note that in $H_7$ the leaves have this property: each leaf except the left-most is the mean average of two adjacent traversed values from $H_3$, scaled with coefficient $c_3 = \sqrt{2}$:

$$-\frac{\sqrt{2}}{2}+\frac{\sqrt{2}}{2}i = c_3\frac{i+(-1)}{2}, \quad -\frac{\sqrt{2}}{2}-\frac{\sqrt{2}}{2}i = c_3\frac{(-1)+(-i)}{2}, \quad \frac{\sqrt{2}}{2}-\frac{\sqrt{2}}{2}i = c_3\frac{-i+1}{2}.$$

The left-most leaf is $\frac{\sqrt{2}}{2}+\frac{\sqrt{2}}{2}i = c_3\frac{1+i}{2}$. Thus we can work cyclically from 1 and the first in-traversed value, then drop the 1 and combine the first with the second in-traversed value, then continue in the same way, until in the last step 1 is used again; proof follows.

Consider Figure 3.3; $m_8$ and $m_{16}$ are the midpoints of the respective segments.

**Figure 3.3. Means of Adjacent Twiddle Factors**




The following equalities hold: $\omega_8 = c_3 m_8 = c_3\frac{1+i}{2}$, where $\omega_8 = e^{\frac{\pi}{8}i}$, $c_3 = \sqrt{2}$, $i=\omega_4=e^{\frac{\pi}{4}i}$; $\omega_{16} = e^{\frac{\pi}{16}i} = c_4 m_{16} = c_4\frac{1+\omega_8}{2}$ for $c_4 = \frac{1}{|m_{16}|}$, where $|m_{16}| = \frac{1}{2}\sqrt{\left(1+\cos\frac{\pi}{8}\right)^2+\sin^2\frac{\pi}{8}}$. Similarly, $\omega_{16}^3 = c_4\frac{\omega_8+i}{2}$ (note that $\omega_{16}^2 = \omega_8$), etc.

**Lemma 4**: For any natural $k$ and $n$ $\exists$ constant $c_{n+1}$, such that $\omega_{2^{n+1}}^{2k+1} = c_{n+1}\frac{\omega_{2^n}^k+\omega_{2^n}^{k+1}}{2}$.

**Proof**: Since $|\omega_{2^n}|=1$, $\omega_{2^n}+1$ is collinear with its argument's angle bisector. So is $\omega_{2^{n+1}}$, since per Halving lemma $(\omega_{2^{n+1}})^2 = \omega_{2^n}$ and multiplication by $\omega_{2^{n+1}}$ is rotation by its argument. Denote $c_{n+1} = \frac{2}{|1+\omega_{2^n}|}$. Consider $c_{n+1}\frac{1+\omega_{2^n}}{2} = \frac{2}{|1+\omega_{2^n}|}\frac{1+\omega_{2^n}}{2} = \frac{1+\omega_{2^n}}{|1+\omega_{2^n}|}$:

its module is 1 i.e. equal to that of $\omega_{2^{n+1}}$. Since it is also collinear to $\omega_{2^{n+1}}$, and both are in the first quadrant, they are equal: $\omega_{2^{n+1}} = c_{n+1}\frac{1+\omega_{2^n}}{2}$. Now multiply both sides by $\omega_{2^n}^k$:

$\omega_{2^{n+1}}\omega_{2^n}^k = c_{n+1}\omega_{2^n}^k\frac{1+\omega_{2^n}}{2} = c_{n+1}\frac{\omega_{2^n}^k + \omega_{2^n}^{k+1}}{2}$. Finally, $\omega_{2^{n+1}}\omega_{2^n}^k = \omega_{2^{n+1}}\omega_{2^{n+1}}^{2k} = \omega_{2^{n+1}}^{2k+1}$.

We can compute the leaves for the next Middle-heap degree (the odd powers of the primitive root of unity for the next power of two) by finding mean averages of consecutive traversed values from the middle-heap of the previous degree, and multiplying by a real-value constant $c_n$. The constant can be calculated once for each size.

This allows us to work out each twiddle factor roughly with one complex i.e. two real-valued additions, and two (real-valued) multiplications; add the O(1) time of working out a constant, for each next power-of-two count. We ignore the divide by two in computing the mean average – it is a cheap right shift of the mantissa, or (even better for loss of precision) decrement of the floating point value's (binary) power. The 2 can be hidden in the constant, but below we will note a benefit if $c_n$ is close to 1. Note this calculation can be carried out independent of the Middle-heap data structure; in section 4.1.3 we show that the Middle-heap traversal takes linear time with a small constant.

Call this the Mean-middle method. Its arithmetic complexity is better than that of any other method of computing twiddle factors. This is a major contribution of this work.

We also show that due to its "binary-chop" pattern, the precision of calculating the twiddle factors in this way is also superior to other methods known so far.

### 3.1.3 Load-Balancing for Incrementality

Our approach is that cores specialise in parts of the FFT computation. This is intuitively expected to improve time locality/SIMD, overall. There is work for four cores at least:

C1. Sampling and Data Conversion;

C2. Twiddle Factor Computation;

C3. Shuffle, or Matrix Transpose (for *pxq* decimation, if six-step used – section 2.3);

C4. Butterfly Operation and any other FLOPs;

Under the premise that the computation is incremental, and progresses in parallel with the sampling process, from among the variants of the classical Cooley-Tukey algorithm, Decimation in Frequency (DIF) is the better candidate than Decimation in Time (DIT): the former allows for the computation to proceed with the lower "half" of samples, while the upper "half" is not yet available.

All four cores progress in parallel and any core except the first may have to wait for its previous one to complete a part of its work, in order to make available some data needed for its computation. It appears wise for C3 to wait on C2 before signalling the release of C4, rather than C4 – for both C2 and C3; cascading synchronisation is simpler.

The synchronisation mechanism suggested herein is spin-lock: the overhead is minimal, due to no context-switching. That said, platform features are welcome, like the register files for fast switching of two threads in the PPE core of the CellBE processor.

Note that the work of C1 is not trivial: samples are normally taken in fixed-point arithmetic from Analog-to-Digital Converters (ADCs) and each needs to be converted to floating-point according to some function, in most cases but not always a linear one.

For platforms with more than four cores, each of the threads assigned to Cj above, for j ϵ {1, 2, 3, 4}, can be forked further to C1*k*, C2*l*, C3*m* and C4*n*, where $k \geq 1$, $l \geq 1$, $m \geq 1$ and $n \geq 1$.

C2 work can be parallelised by splitting the set of Middle-heap leaves into 2, $2^2$, ... subsets to assign to C2*l* for $l = 1, 2$, etc. This improves space locality for each core, because only ½, ¼ etc. of the middle-heap is accessed, with false sharing only possible at the very low levels of the heap. There is no false data sharing with others since all the twiddle factors do need to reside in memory shared with C4 (and possibly C4*n*). Because the twiddle-factor computation is efficient, parallelising C2 work should be done sparingly and with care not to compromise load-balancing by taking cores away from the more computationally-intense C4 work, or from the more memory-operation intense (load/store/move/copy) C3 work. It is envisaged that in most cases not more than two cores will be required for twiddle-factor computation.

The C3 shuffle (or transpose) work is also easy to parallelise into C3*m*: Bit_Rev() in section 3.2.4 is divide(into 2)-and-conquer; any transpose is carried out over small sub-matrices.

To parallelise C3 and C4 work into C3*m* and C4*n*, basic results from load-balancing research on processor arrays and hyper-cubes might come handy. However due to the minimal number of overall available cores (up to 8-9), the benefit from research that relies on such a large scale would be limited – simpler strategies are needed.

### 3.1.4 Core-Monotonic Strategy

It is intuitively clear that the less the spin-locks stay active, the more efficient the computation. We shall refer to the following as the Core-monotonic strategy: The lower the core index C*j*, the more its work should be split for parallelisation. This need not be confused with a higher number of parallel cores for lower core index: similarly to C2 above, the C1*k* split will likely never go beyond C1*2*, however even the slightest chance that the C2 spin-lock fires should cause a split (if there are >4 cores).

## 3.2 The Algorithms

### 3.2.1 Middle-heap Algorithms

From here on, middle-heap is in a 0-based array with H[0] = Complex(1,0) = $1+0i = e^{0i}$. Efficient implementation of its in-traversal needs to be considered, as well as its space locality (estimating cache misses). The algorithm below is optimal in both; the proof is in section 4.3.1.

From a leaf, including initially the leftmost, it goes up to the first yet un-traversed parent; from there takes the right sub-tree searching left, depth-first, until a leaf is found.

```
Next_MiddleHeap_Index(n,i) /* Returns the next in-traversal index for i */
    m = 2^(n-1); /* middle of array, leftmost leaf of full heap */
    r = REM(i,2m); /* put within bounds by finding remainder modulo 2^n */
    if r = 0 /* start of traversal */
      then return m; /* return leftmost leaf – the primitive root of unity */
    elseif r = 2^n-1 /* heap-size() from the def., roll over: */
      then return 0; /* the index of complex (1.,0.) */
    endif;
    if r >= m then /* is leaf */
        while ODD(r) /* is right sub, i.e. parent has been traversed */
          do r = PARENT(r); /* backtrack till even */
        endwhile;
        return PARENT(r); /* go to parent of left sub */
      else /* is not leaf */
        r = RIGHT(r); /* take right sub-tree */
        while r < m /* is not leaf */
          do r = LEFT(r); /* go to left child */
        endwhile;
        return r;
    endif;
```

The functions used: `REM()` and `ODD()` are straightforward; `ODD()` can be implemented very efficiently as a bit-AND with the least-significant bit; `PARENT()`, `RIGHT()` and `LEFT()` are those from the heap definition above: they comprise division or multiplication by 2 – can be implemented as a single shift of a register if desired.

The computation of the twiddle factors, as described in section 3.1.2, uses the traversing of the Middle-heap. The base algorithm, presented here for clarity, can be

improved using symmetries: compute first quadrant, then swap/negate Re/Im parts per section 2.2 second last paragraph.

```
Next_MiddleHeap_Leaves(H[0:2^n],n)/*compute H_{2N+1}'s leaves from H_N*/
    ω1 = H[0]; /* 1 cast as complex constant COMPLEX(1.,0.) */
    ω2 = H[Next_MiddleHeap_Index(n,0)];/*N+1th prim. root of unity*/
    m = MOD((ω1+ω2)/2.); /* module √(Re²+Im²) of mean average */
    c = 1./m; /* the factor (real) value to normalise modules to unit circle */
    t1 = 0; N = 1<<n;/* shift, no arithmetic complexity */
    for i = N+1 to 2N+1 do
      t2 = Next_MiddleHeap_Index(n,t1);
      ω2 = H[t2];
      H[i] = CDIV2(ω1+ω2);/*2 FLOPs & efficient (complex) divide by 2*/
      H[i] = H[i]*c; /* multiply complex by real-valued factor – 2 FLOPs */
      ω1 = ω2; t1 = t2;
    endfor;
    return H[];/*of size 2N+1*/
```

The function receives $H_N$ in array of size $2N+1$, and returns $H_{2N+1}$ in it. The functions `MOD()` and `COMPLEX()` are straightforward. `CDIV2()` is divide by 2 of the real and imaginary part, implemented as suggested in section 3.1.2 – by shift of the mantissa or decrement of the power. Either operation is cheaper than a FLOP on most platforms.

Also, it is redundant to store both a complex number and its conjugate/opposite. Alternatively, fill the right sub-tree of the middle-heap root with the values from its left sub-tree ordered differently. In-place DIF for naturally ordered output requires exactly the values from the left sub-tree in bit-reversed order (see section 3.2.5 for more details).

### 3.2.2 Threading and Synchronisation

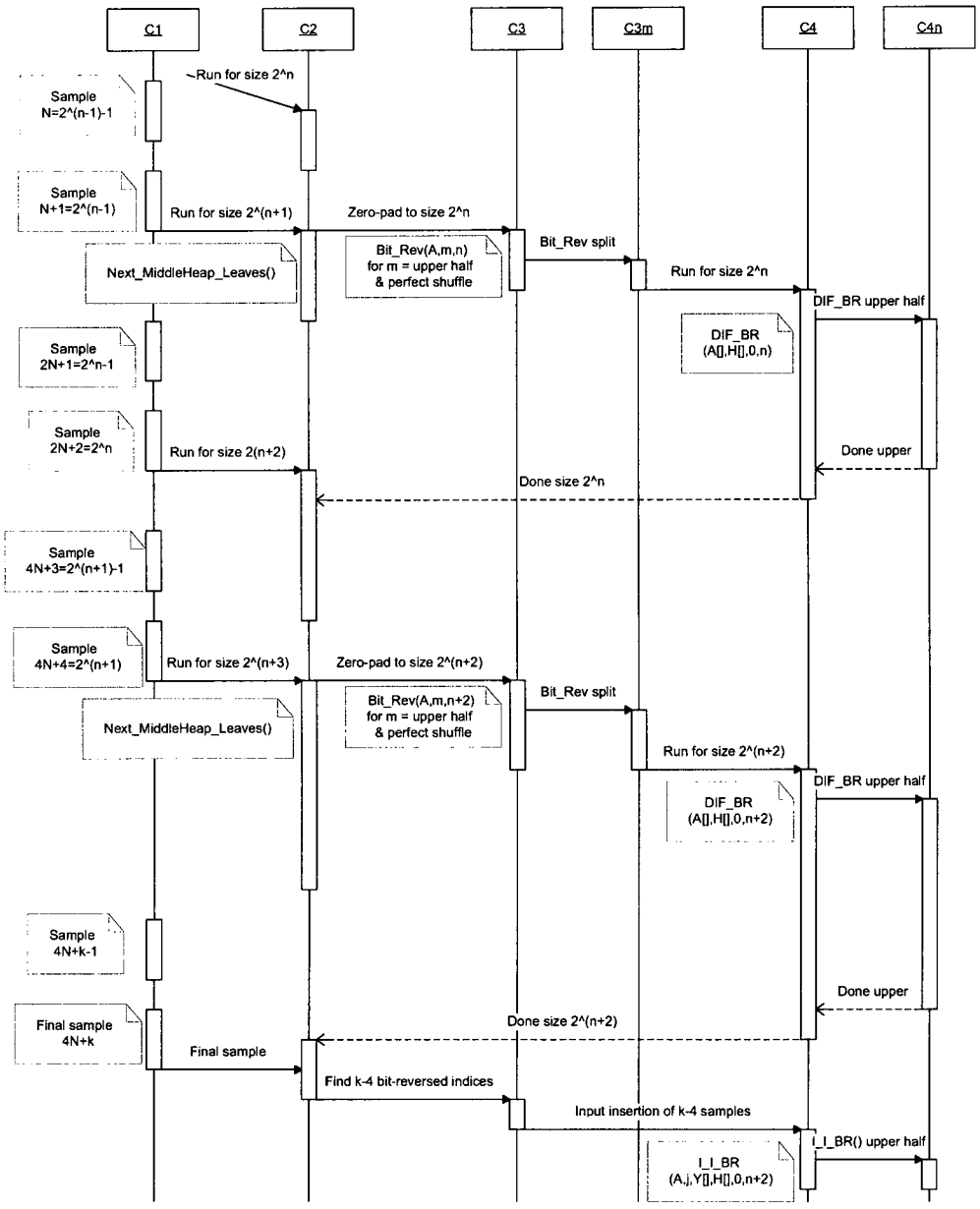
There is normally one thread per core, as described in section 3.1.3. Some processors support in hardware context switching by maintaining two register files – one for each thread. This allows for combining C1 and C3 onto one such core – on the CellBE the PPU core has this feature, and is optimised for the buffer operations of data acquisition, and shuffle/transpose, while the 6-8 SPU cores are optimised for intensive computation, including FLOPs. Spin-locks are used for synchronisation. Per the Core-monotonic Strategy, the lower the core number, the less likely a spin-lock will be locked.

As each sample is taken and converted to usable format (floating-point), C1 checks if the next power of two is surpassed (or a final sentinel is detected), and signals C2. C2 then exits eventual spin-lock within its main loop, and continues by calling `Next_MiddleHeap_Leaves()` accordingly, for the next size. It is wise for C2 to be at least one step ahead of C1, i.e. when C1 goes past $N=2^n$, C2 should start calculating at least $H_{2N+1}$, possibly even $H_{2(2N+1)+1}$, in memory shared with C4.

Within DIF, C3 performs bit-reversal shuffle of the low and high halves of the samples, then signalling C4 to do in-place the basic butterfly operation $a_j+a_{j+n/2}$, $a_j-a_{j+n/2}$. These values are then multiplied by the twiddle factor $\omega_n^j$ (see section 2.1.6). Thus C4 performs a version of DIF, see the next sub-section for details. Once sample $2^{n+1}$ arrives, C3 performs bit-reversal on the upper $2^n$ samples, then perfect shuffle on all $2^{n+1}$. Since each of the halves is bit-reversed, this orders the MS bit to yield bit-reversal of all values.

Consider the following diagram. For names and descriptions of the routines see the following sections. The spin locks are hit at the end of the shown activations.

**Figure 3.4. Sequence of Multi-core Processing**



### 3.2.3 Design of the FFT Plan

For time locality, the FFT plan is DIF, not split-radix. It divides into at least two threads to improve space and time locality – vectorization and SIMD: the C3 thread continuously performs perfect shuffles as the sample values become available. C4 performs butterflies and twiddle-factor multiplication, for values that are adjacent as a result of the shuffles.

The following is a perfect shuffle utility function that will be used further:

```
Shuffle(A[],m,n,B[]) /*Perfect shuffle of 2^n values from A[m+1] into B[]*/
    for i = 1 to 2^(n-1) do
      B[2i-1] = A[m+i];
      B[2i] = A[m+i+2^(n-1)];
    endfor;
    return B[];
```

Calling Shuffle() aims at initial space locality for C4: C4 will start work with adjacent data; C3 locality and false sharing (lack of) with C4 will be discussed in section 4.3.3.

```
Sign_Butterfly(A[]) /* Replace complex A1 and A2 with A1+A2, A1-A2 */
    sum = A[1]+A[2]; dif = A[1]-A[2];
    A[1] = sum; A[2] = dif;
    return A[];
```

Sign_Butterfly() could be run by a different core, if one is available and allocated per the Core-monotonic Strategy, since it involves FLOPs rather than shuffling data.

Our FFT plan is based on the recursive in-place algorithm of Radix-2 DIF (see section 2.1.6). What follows is just description of the logic – the algorithm below has to be adapted to shuffling of the data by a different core, as well as to incrementality of the application - array A[] will get populated upwards, as samples become available from C1:

```
DIF(A[],H[0:2^n],BasE,l);/*In A[2^n], 2^l coeff start at BasE; H[] middle-heap*/
    if l = 0 then
      return A;
    endif;
    M = 2^(l-1); /* middle */
    BasO = BasE + M; /* split "odd" block (1-based even) into 2nd half */
    t = 0;
    for k = 1 to M do
      e = A[BasE+k]+A[BasO+k];/* with extra core, call Sign_Butterfly() */
      o = (A[BasE+k]-A[BasO+k])*H[t]; /* here too – then multiply */
      A[BasE+k] = e;  A[BasO+k] = o;
      t = Next_MiddleHeap_Index(l,t);
    endfor;
    DIF(A,H,BasE, l-1); /* even sub-transform of half length */
    DIF(A,H,BasO, l-1); /* odd sub-transform of half length */
    return A;/* in place, in bit-reversed order! */
```

Based on DIF formulae in section 2.1.6, DIF(A,H,0,n) returns DFT(A), output bit-reversed.

**Proof**: The algorithm maintains these two invariants at each recursive step: - the for-loop tallies the right coefficients to be used recursively by even/odd half-sizes; - all 1s in the LS bit go into the high half of the respective array part (half, 1/4, 1/8, etc.); so the decimation is correct, and each step puts the next bit in the right bit-reversed order.

C3 (C3*m* if more than one core allocated) would need to carry out the inverse bit-reversal. We prefer that input be supplied bit-reversed, for a natural output. Unless recursion uses large amounts of local data, it is good for space locality, due to the stack dynamic with cache.

Note the stride between BasE and BasO is the major issue with space locality when FFT is implemented on a real processor. Our implementation will take care of this by separating the bit-reversal shuffle onto a different core, and in this way will also improve the time locality of the implementation – the loop body becomes simpler for pipelining.

### 3.2.4 Shuffle for Bit-reversed Input

Consider the following algorithm.

```
Bit_Rev(A[],m,n)  /* Bit-rev. permute of 2^n elements starting from A[m]*/
  if n<=1 then return A[] endif; /*For size ≤2 there is nothing to do*/
  A[m]=Bit_Rev(A[],m,n-1); A[m+2^(n-1)]=Bit_Rev(A[],m+2^(n-1),n-1);
  A[m]=Shuffle(A[],m,n,B[]);/*B is a large enough work area*/
return A[];
```

As in DIF(), Bit_Rev() maintains the invariant that each recursive call puts the next bit in the right order for bit-reversed permute, without affecting previous ones. It can also work incrementally (section 3.2.2): omit first Bit_Rev() call if lower half already bit-reversed.

### 3.2.5 DIF Algorithm for Ordered Output

DIF() needs to be modified to take into account bit-reversal input, and provide naturally ordered output. Note that the bit-reversed permutation of a bit-reversed permutation is the natural order permutation (the mirror image of the mirror image of a binary string is that same string). Then consider DIF() with the following change: the recursive calls for the half-sizes occur before the butterflies. This ensures that the new DIF_BR() algorithm processes the bit-reversed permutation of the input vector into the natural order result. Consider the two parts of the following figure, © R. Stern of Carnegie-Melon University:

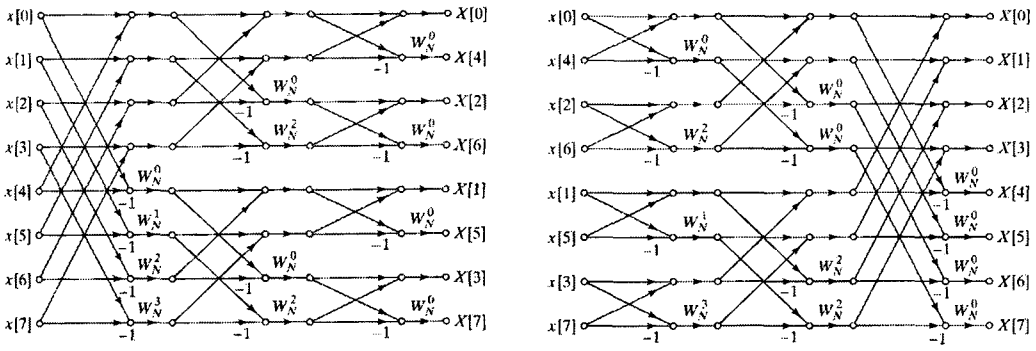


**Figure 3.5. DIF Natural vs. Bit-Reversed Input**

It is not hard to see that they describe the same (non-planar) graph but the right-hand side was obtained from the left by bit-reversing the vertical order of the horizontal edges. The right-hand side diagram shows how the butterflies follow the low-high split in DIF, and at every stage take values recursively obtained from previous stages. In pseudo-code:

```
DIF_BR(A[],H[0:2^n],BasL,l,t);/*In A[], 2^l coeff at BasL in bit-reversed
                                order; t – index of (single) twiddle factor.*/
  if l = 0 then
    return A;
  endif;
  M = 2^(l-1); /* middle */
  BasH = BasL + M; /* split high half into odd block sub-transform */
  DIF_BR(A,H,BasL,l-1,2t); /* 2t is left child in H_N */
  DIF_BR(A,H,BasH, l-1,t>1?2t+1:3); /* right child in H_N's right half*/
  for k = 1 to M do
    e = A[BasL+k]+A[BasH+k];/* with extra core, call Sign_Butterfly() */
    o = (A[BasL+k]-A[BasH+k])*H[t]; /* here too – then multiply */
    A[BasL+k] = e;  A[BasH+k] = o;
  endfor;
  return A;/* in place, in natural order! */
```

The expression `t>1?2t+1:3` evaluates to `2t+1` if `t>1`, and to `3` otherwise. It ensures that only the right-side half of the middle heap is visited – one identical to the left but in bit-reversed order (DIF uses only half the twiddle-factors, above the real axis).

Modify Next_MiddleHeap_Leaves(): only compute the low half of the entries, then copy them to the high half and call Bit_Rev() from section 3.2.4 on that copy. This populates the right-side half with values arranged to be accessed in bit-reversed order, which is needed for the correct execution of DIF_BR():

Each level of a middle-heap can be viewed as leaves of a middle-heap of smaller size. Each new leaf level adds another bit of value 1 to the size consisting of 1s only ($N=2^n-1$), and the left-most leaf doubles the degree of the largest primitive root of unity. Due to the Cancellation lemma, this doubles the powers in all the upper layers for the new primitive root – a bit of value 0 is appended as least-significant to the binary representation of each power. If each level is bit-reversed, the new bit is most significant in the index and does not matter. So the whole sequence H[0],H[1],...,H[$N$] is $\{\omega_{N+1}{}^k\}$, $k$ in bit-reversed order.

The whole middle-heap is shared between C2 and C3, so run Bit_Rev() on C3, in parallel. As usually noted in this work, there exist faster ways to compute indexing of bit-reversal permutations, including implemented directly in hardware and/or microcode.

### 3.2.6 Correctness of the DIF Algorithm for Ordered Output

DIF_BR() is based directly on section 2.1.6 Idea 2 –splitting of the low-half terms from the high. To prove strictly that the algorithm is correct, what remains to be shown is that:

a) The bit-reversed input produces naturally ordered output;

b) The correct twiddle-factors are used to combine the low and high halves.

**Proof**: The algorithm maintains the following two invariants at each recursive step:

a) Each output value is the result of a superposition of butterfly and perfect shuffle $z = bf_k(PS(x_1x_2))$, where $PS$ is a perfect shuffle of the concatenated strings $x_1$, $x_2$, and $bf_k(y_0,y_1, \ldots,y_{2k+1}) = (y_0+y_1,(y_0-y_1)\omega, y_2+y_3,(y_2-y_3)\omega, \ldots, y_{2k}+y_{2k+1},(y_{2k}-y_{2k+1})\omega)$, for some twiddle-factor $\omega$. This in-place Cooley-Tukey computation pattern has either the input, or the output bit-reversed. Our input is bit-reversed, so the output must be in natural order.

b) For each level $l>0$ of the recursion, its sequential invocations use all powers of $\omega_{N+1}$ from 0 to $2^{l-1}$ in bit-reversed order. This holds because in DIF() they were used in natural order, and DIF_BR() runs the same butterflies as DIF() but in bit-reversed order. Use mathematical induction upon $l$. Directly check the recursion root and its two children. Assume that at level $l$ the statement holds. Take an arbitrary recursion instance at level $l+1$. If its parent uses H[$t$], it uses either H[$2t$] when left child, or H[$2t+1$] when right. The parents run in sequence of ascending indices, so the child invocations also use leaves of ascending indices, i.e. with powers in bit-reversed order. This completes the proof.

### 3.2.7 Efficiency of the DIF Algorithms

Similarly to DIT Cooley-Tukey, DIF() and DIF_BR() both run $n=\log_2 N$ levels of recursion. At level $l$, there are $2^{n-l+1}$ copies of the recursion activation, each running a single loop of $2^{l-1}$ iterations of constant complexity denoted here by $c$. Hence the overall complexity $C = n2^{n-l+1}2^{l-1}c = c2^n n = \Theta(N \log N)$, directly from the definition of $\Theta$.

An interesting question is whether DIF() or DIF_BR() achieves better performance. With DIF(), the bit-reversal permute needs to occur sequentially afterwards. This may be efficient on platforms with built-in bit-reversal indexing, like some DSPs – on those the

C3 thread seems unnecessary, however it provides space locality for C4. DIF_BR() is the better candidate, especially if possible to parallelise the input shuffle with the butterflies. DIF/DIF_BR need to be used in a way so that the calculation progresses at least partially incrementally as sampling data becomes available.

### 3.2.8 Introducing Input Insertion

Since each input affects each output, it seems impossible to complete all necessary work on partially available data, with the DIT or DIF decimation techniques, and at $N \log N$ asymptotical computational complexity. With O($N \log N$) work, a single sample could be fully accounted for: consider Figure 3.5; all merges of directed edges are additions, so it is possible to accumulate each new input value into the result vector by following each path from that input value into any position of the result – call this input insertion. [13] shows how to skip part of the calculation for sizes different from powers of two. We suggest implementing it like this: make each butterfly a simple copy when one of the values is 0. Whenever sampling reaches a power of two, start performing FFT for double that size with 0s instead of the unavailable data, unless an overrun occurs i.e. the next sample arrives before the previous FFT has finished, in which case simply continue with the ongoing computation. Once all samples are in, compare arithmetic complexity to see whether it is more efficient to perform input insertion, or the whole FFT from scratch.

Input insertion pseudo-code follows. The program I_I() works similarly to DIF(), thus natural-order j affecting output Y[] that is in bit-reversed order. A very similar routine can of course be written to insert data with a bit-reversed index into natural-order outputs.

```
I_I(A,j,Y[],H[0:2^n],BasE,l);/*Input insertion of A[j] into Y[2^l] from BasE,
                                H[] is middle-heap*/
  if l = 0 then
    Y[BasE] = Y[BasE] + A; /* Add to each array element */
    return Y;
  endif;
  M = 2^(l-1); /* middle */
  BasO = BasE + M; /* split "odd" block (1-based even) into 2nd half */
  t = 0;
  for k = 1 to MOD(j-1,M) do /* find twiddle factor */
    t = Next_MiddleHeap_Index(l,t);
  endfor;
  if j≤M then
      I_I(A,j,Y,H,BasE, l-1); /* even of half length */
      I_I(A*H[t],j,Y,H,BasO, l-1); /* odd of half length */
    else
      I_I(A,j-M,Y,H,BasE, l-1); /* even of half length */
      I_I(-A*H[t],j-M,Y,H,BasO, l-1); /* odd of half length */
  endif;
  return Y;/* is in bit-reversed order! */
```

I_I() first finds the necessary twiddle-factor, then calls itself recursively for the upper half and the lower half of its input range. It copies DIF(), except for the loop, as it deals with a scalar piece of data, not an input vector.

Note finding the twiddle factor can be done much more efficiently: if $j$ is even, take H[$m$+($j$-1)/2], where $m$ is the middle of H, else call Next_MiddleHeap_Index($l$,$m$+$j$/2-1).

Name I_I_BR() the routine that inserts from bit-reversed input into natural output; it can use efficient bit-reversal of a single value, in hardware or the one [6] describes.

### 3.2.9 Optimise DIF Algorithm by Unrolling Recursion Leaves

The recursion stops when the trivial size 1 of the array is reached. This is inefficient, since at least the previous two sizes of 2 and 4 are trivial computationally – the twiddle-factors are 1, $i$, $-1$ and $-i$, so the computation could be performed using no FLOPs, just sign reversal and swapping of real with imaginary part. Known practical implementations may go even further with unrolling the low-degree loops (when iterative) or in stopping the recursion at a higher level. Here we stop at the 8th primitive roots of unity ($H_7$).

$$W = \{\frac{\sqrt{2}}{2}+\frac{\sqrt{2}}{2}i,\ \ i,\ \ -\frac{\sqrt{2}}{2}+\frac{\sqrt{2}}{2}i,\ \ -1,\ \ -\frac{\sqrt{2}}{2}-\frac{\sqrt{2}}{2}i,\ \ -i,\ \ \frac{\sqrt{2}}{2}-\frac{\sqrt{2}}{2}i\}.$$

The multiply by $\omega_8^{2n+1}$ can be optimised using real-valued multiply by $\sqrt{2}/2$, swap and negate. We change the first if-statement to do the calculation using Horner's rule:

```
DIF_BR(A[],H[0:2^n],BasL,l,t);/*In A[], coef at BasL in bit-reversed order*/
 if l = 3 then
  B[1]=A[BasL+2]+A[BasL+6]+A[BasL+4])+A[BasL+8];
  B[1]=B[1]+A[BasL+1]+A[BasL+5]+A[BasL+3]+A[BasL+7];
  B[2]=A[BasL+2]+W[1]*(A[BasL+6]+W[1]*(A[BasL+4]+W[1]*A[BasL+8]));
  B[2]=A[BasL+1]+W[1]*(A[BasL+5]+W[1]*(A[BasL+3]+W[1]*(A[BasL+7]+W[1]*B[2])));
  B[3]=A[BasL+2]+ SWN(A[BasL+6]+ SWN(A[BasL+4]+ SWN(A[BasL+8])));
  B[3]=A[BasL+1]+SWN(A[BasL+5]+ SWN(A[BasL+3]+ SWN(A[BasL+7]+ SWN(B[3]))));
  B[4]=A[BasL+2]+W[3]*(A[BasL+6]+W[3]*(A[BasL+4]+W[3]*A[BasL+8]));
  B[4]=A[BasL+1]+W[3]*(A[BasL+5]+W[3]*(A[BasL+3]+W[3]*(A[BasL+7]+W[3]*B[4])));
  B[5]=A[BasL+2]-A[BasL+6]+A[BasL+4]-A[BasL+8];
  B[5]=A[BasL+1]-A[BasL+5])+A[BasL+3])-A[BasL+7]+B[5];
  ... /* continue similarly for 3rd and 4th quadrant */
  return B;
 endif;
 ...
```

`SWN()` swaps the Re and Im part and negates the new real part – no FLOPs involved. The DIF() version would be similar, with indices for in-order input, bit-reversed output.

# Chapter 4: Analysis of the Generic Design

This chapter evaluates properties of our design, especially the computational complexity.

## 4.1 Properties of the Middle-heap

### 4.1.1 Properties Derived from Min-Heap

**Lemma 5:** All even array indices in a middle-heap are left children of their parents; all odd indices (except the root 1) are right children of their parents.

**Proof**: This is straightforward using the fact that at each heap level except the root, and possibly except the leaves, there is an even number of elements.

### 4.1.2 Properties Derived from Complex Roots of Unity

**Lemma 6:** All odd powers of $\omega_{N+1}$ are exactly the leaves of the middle-heap $H_N$.

**Proof:** Due to the Halving lemma, any even power ${\omega_{N+1}}^{2k} = {\omega_{(N+1)/2}}^{k}$ must already occur higher up, with degree $(N+1)/2$ (note $N+1$ is a power of two).

Therefore all the non-leaves are even powers of the primitive complex root of unity – all the non-zero ones. They are returned in ascending order when the heap is in-traversed. By definition, since we sort by degree then by power, in the sorted heap they are in ascending order: $\omega_{N+1}$ is followed by ${\omega_{N+1}}^{3}$, etc.

The middle of the middle-heap array is the $N+1^{\text{th}}$ primitive complex root of unity, where $N$ is the heap size. This is so since the middle of the array is the left-most leaf of the tree, i.e. the lowest odd power (1) of the primitive root.

### 4.1.3 Computational Complexity of Traversal

Locating the next power of the primitive root of unity per section 3.2.1 is O(log $N$) worst case: indeed the most that needs to be traversed is the height of the whole tree, which is log $N$. It is however constant time O(1) average case. To prove it, consider this:

**Lemma 7**: When the whole middle-heap $H_N$ is traversed by using Next_MiddleHeap_Index() consecutively, the number of edges traversed is $E<2N$.

**Proof**: When traversing the whole heap, in half of the $N$ cases namely all the even-indexed leaves (¼) plus their parents (another ¼), only one edge is traversed. The same applies to the higher levels, each one half the size of the preceding, so in ¼ of the cases 2 edges will be traversed, in one eighth – 3 edges, etc. Thus the edges traversed are $E=\sum_{j=1}^{\lfloor \log N \rfloor} \frac{N}{2^j} j = N \sum_{j=1}^{\lfloor \log N \rfloor} \frac{j}{2^j} < N \sum_{j=1}^{\infty} \frac{j}{2^j}$. To simplify this expression note the following:

$\sum_{j=1}^{\infty} \frac{j x^{j-1}}{2^j} = \left( \sum_{j=0}^{\infty} \frac{x^j}{2^j} \right)' = \left( \left( 1 - \frac{x}{2} \right)^{-1} \right)' = \frac{2}{(2-x)^2}$. Substitute $x = 1$ to obtain $E<2N$, i.e. the work to traverse the whole middle-heap in order of increasing powers of the $N$+1$^{th}$ primitive root of unity is O($N$) with constant <2. This completes the proof.

**Corollary**: The work for locating the next degree of the $N$+1$^{th}$ primitive complex root of unity is average O(1). Indeed if it takes overall less than $2N$ steps to traverse the whole structure of size $N$, finding the next power of the primitive complex root of unity takes on average less than 2 steps – simple arithmetic operations at that.

This result is asymptotically optimal for traversing any whole structure of size $N$. It is also quite good arithmetically, since it is less than twice the theoretical lower limit.

To locate random index $j$, we use Next_MiddleHeap_Index($l$,$m$+($j$-1)/2) (see section 3.2.8). So three operations need to be added to the function's – one addition, one decrement and one division. This works out to $\log N+3$ operations, on average <5.

### 4.1.4 Arithmetic Complexity of Computing Twiddle-factors

In the single for-loop of Next_MiddleHeap_Leaves(), there are four FLOPs: one complex addition and two multiplications by c. There is also a call to Next_MiddleHeap_Index(), which is average less than 2 integer divisions or multiplications, and possibly 1 integer addition. Strictly, arithmetic complexity $C$ counts only the 4 FLOPs, but it is fair to take into account that arithmetic (not if implemented inline, as shifts/increments) – with a co-processor they take similar time. The loop runs $N+1$ times (for the $N+1$ odd powers). So the count is $7(N+1)+Sq+7$, where $Sq$ is the work it takes to compute square root in the expression MOD((ω1+ω2)/2.) (replaces calls to sin/cos in other methods). The constant 7 is reached as follows: twice multiply to raise a floating point value to the power of two in the MOD() function, 2x add to find the sum of two complex numbers plus one time in the MOD() function, and finally 2x divide by 2 – once for the Re and once for the Im part of the sum. Sum the above expression over the middle-heap height (note $N=2^n-1$): $C \le$

$$\sum_{j=2}^{\lfloor \log N \rfloor}(7.2^j+Sq+7) = 7\sum_{j=0}^{\lfloor \log N \rfloor}2^j - 21 + (Sq+7)(\lfloor \log N \rfloor - 1) = 7N + (Sq+7)\lfloor \log N \rfloor - Sq - 28.$$

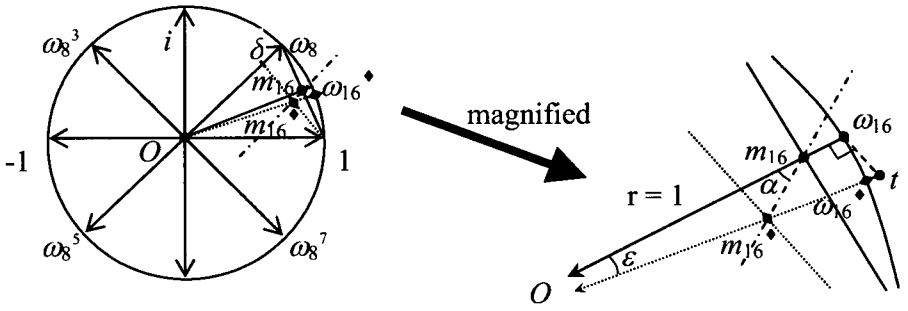
Under the assumption that FLOPs are much more expensive than other computation, only the 4 FLOPs per loop count; if also the two divisions by 2 are implemented efficiently as function CDIV2() using shift/decrement, as described in section 3.1.2: $C \le 4N + (Sq+5)\lfloor \log N \rfloor - Sq - 17$.

The dominating term is the same as in the best theoretical result so far, $4N$ (see [4] ), however, due to finite-arithmetic error propagation with the recursive sine evaluation, that method needs more operations for compensating the accumulating errors.

**Theorem**: The Euclidean-distance error of the Mean-middle method is always below the floating-point precision used.

**Proof**: See Figure 4.1 below. Because $\mathrm{Re}(\omega_8) = \mathrm{Im}(\omega_8)$, its argument (angle) is accurate – exactly $\pi/4$. Any error $\delta$ in its module (length) creates a new (deltoid-shaped) mid-point $m_{16}^{\blacklozenge}$. Due to side-split, line $m_{16}m_{16}^{\blacklozenge}$ is parallel to $O\omega_8$ and $|m_{16}m_{16}^{\blacklozenge}| = \frac{1}{2}\delta$. Let $Om_{16}^{\blacklozenge}$ intersect the unit circle at $\omega_{16}^{\blacklozenge}$; by definition, $\delta$ has no effect on $\omega_{16}^{\blacklozenge}$'s module.

**Figure 4.1. Error Propagation of Mean-middle Method**




Denote $\varepsilon = \angle m_{16}Om_{16}^{\blacklozenge}$, $\alpha = \angle\omega_8O\omega_{16} = \angle Om_{16}m_{16}^{\blacklozenge}$. Apply the sine law to $\Delta m_{16}Om_{16}^{\blacklozenge}$:

$$\frac{2\sin\varepsilon}{\delta} = \frac{\sin(\pi-\varepsilon-\alpha)}{|Om_{16}|} = \frac{\sin(\varepsilon+\alpha)}{\cos\alpha} = \sin\varepsilon + \tan\alpha\cos\varepsilon\text{, hence } \tan\varepsilon = \frac{\delta}{2-\delta}\tan\alpha\,.$$

Because the error from finite floating-point precision $\delta < 2^{-16} << 2$, and $\alpha = \pi/8 < \pi/4$,

$\tan\varepsilon = \dfrac{\delta}{2-\delta}\tan\alpha < \dfrac{\delta}{2}\tan\dfrac{\pi}{4} = \dfrac{\delta}{2}$. Since segment $t\omega_{16}$ is opposite to an obtuse angle in $\Delta\omega_{16}t\omega_{16}^{\blacklozenge}$, $|t\omega_{16}| = \tan\varepsilon > |\omega_{16}\omega_{16}^{\blacklozenge}|$. Thus the worst-case Euclidian-distance error is $< \delta/2$. For large n, $\tan\alpha << 1$ and the error is well below floating-point precision i.e. negligible.

A note on amalgamating the divisor 2 from finding the mean average (section 3.2.1) into the scaling factor $c = 1/m$: this would save the (albeit efficient) divide by 2 in every iteration of the for-loop, but lead to $\lim_{n\to\infty} m_n = 2$, instead of 1 originally. If kept as it is, for large enough $n$, $m_n - 1$ will go below the floating-point precision. If properly detected, this condition may save the need for the complex-to-real-valued multiplication by $c = 1/m$ in the loop. The gain for very large sizes is twofold – dominating term goes from $4N$ to $2N$.

To optimise the calculation finally, it is standard to compute the first quadrant only, then use the symmetries per section 2.2 second last paragraph – swap/negate Re and/or Im parts to obtain the other three quadrants (only first and second are needed for DIF).

## 4.2 Time Locality and Pipelining

### 4.2.1 Twiddle-factor Computation

The expected efficiency of twiddle-factor computation is very high, since the different logic's circuitry of the processor/core is exercised in parallel: with each iteration of the main for-loop, there are 2 integer multiplies/divides or shifts, or one of each, 1 addition, 2 multiply FLOPs and one add FLOP. The FLOP operation follows a standard MAC (multiply-and-accumulate) pattern: the expression is `H[i]=H[i]*c`, where of course H[i] is complex. It doesn't get any better than this in a loop, for pipelining.

### 4.2.2 FFT Plan

The butterfly operation is a pretty standard pattern for processor pipelining. The unrolling of loops/recursion leaves described in section 3.2.9 fits well with SIMD vectorization.

## 4.3 Space Locality

### 4.3.1 Cache Misses with Twiddle-factor Computation

The cache complexity of the Mean-middle method is only slightly different from the computational complexity of traversal – it is very cache-efficient as the middle-heap is traversed sequentially: as was discussed in section 4.1.3, accessing H[($N$+1)/2] is followed by accessing its parent H[($N$+1)/4], then H[($N$+1)/2+1], then shortly thereafter – H[($N$+1)/4+1], etc. Whatever the cache line size (ad-hoc cache-obliviousness!), this consistently takes almost two lines only, until either one line is exhausted or ($N$+1)/4 leaves have been traversed; meanwhile new cache lines are required with exponentially decreasing frequency.

### 4.3.2 Cache Misses with the FFT Plan

The cache efficiency of the FFT plan is hard to establish accurately, almost impossible without knowing the particular platform of execution. The design features recursive routines but of course when they are implemented the algorithms can become iterative instead, depending on whether stack locality takes priority over the improved running time of iteration. Another observation that can be made is that although each phase of the DIF makes strides, in most cases large ones up to half the size of the problem, there is again a pattern of access similar to that of middle-heap traversal – the location alternates between two spots most of the time i.e. two cache lines are consistently reused. The unrolling of loops/recursion leaves described in section 3.2.9 also contributes to cache

efficiency, by avoiding the fragmentation into small-sized arrays to be decimated into two halves – the arrays are sized at least 8 complex numbers (16 floating-point values).

Frigo et al. ([9] ) have proven cache obliviousness of a particular FFT Radix-$\sqrt{N}$ decimation when a specific cache-oblivious algorithm for matrix transpose is used. Our design is totally different, and to expect straightforward cache obliviousness would be unrealistic. However the following observation can be made: the strides of both DIF algorithms are sequential powers of two – increasing or decreasing with each next phase. Assuming that the cache line is larger than 16 floating-point numbers, and its size is a power of two, it is only natural to expect that it will always be crossed, either upwards or downwards, no matter what the size of the problem. Thus the consistency part of cache obliviousness is undoubted. Optimality can be the topic of future research.

### 4.3.3 Avoiding false sharing

The only candidate culprits for false sharing could be C3 and C4: C1 has to share all its data (the samples) with C3, and C2 has to share all its data (the twiddle-factors) with C4. C2 and C3 don't share any data at all. Also as shown in section 3.1.3, C2*l* are only affected by false sharing between themselves at the start of the middle-heap array – when the levels of the tree have half-sizes smaller than the cache line – the effect is negligible.

Because C4's DIF_BR() runs sequentially after C3's Bit_Rev(), no false sharing between them is possible. Our implementations perform the DIF completely in place, and each C4*n* core works on its bit-reversed upper (odd-indexed) or lower (even-indexed) half sub-array, therefore false sharing cannot occur. Each C3*m* also works on its own upper/lower half-array. These are both followed sequentially by combining the two

halves: when C4 performs the final stage butterflies with a stride of half the array, or when work area arrays are used by C3 during the final perfect-shuffle stage.

Both phases are very cache-efficient: due to the nature of the perfect shuffle, locations at the destination and at each source half are accessed sequentially; as mentioned in section 4.3.2, C4 follows a pattern alternating between cache lines from the two halves.

Besides, our bit-reversal algorithm is only presented for completeness – there is an efficient way to compute bit-reversal indices described in [6] , and only shuffle the actual data once per item. Also, as previously noted many platforms designed for intensive computation provide bit-reversal indexing implemented in hardware or microcode.

Thus the problem of false sharing is all but eliminated.

## 4.4 Final Optimisation of Load-balancing

C2 is occupied at the start of the whole process, once for each next power-of-two problem size, and goes idle soon after the maximum problem size is reached.

On asymmetrical platforms like the CellBE, a core sufficiently optimised for FLOPs would have been assigned as C2, since its main task is the computation of twiddle-factors. Once the latter is over, C2 can undertake one part of the butterfly operation's envisaged split into Sign_Butterfly() and multiplication by a twiddle-factor (see section 3.2.3). Imagine a vertical line across the arrows of each $W_N^j$ step of each phase in Figure 3.5: this is where the split occurs. To preserve the natural direction of spin-lock dependencies, C2 will perform Sign_Butterfly() holding the spin-lock for C4; once unlocked, C4 will do the multiplication by the particular twiddle-factor in parallel with the Sign_Butterfly() for the next pair of values.

This is reasonably load-balanced. Two complex additions by C2 (of 2 FLOPs each) work out to 4 FLOPs, and one complex multiplication by C4 takes at least 5 FLOPs – 6 in our case since we apply no algebraic optimisation. Further, unlike DIT only half of the steps in each DIF phase involve multiplication by a twiddle-factor. Thus C4 takes roughly 1.5/2 times, i.e. 75% of the FLOP-related load of C2 in the butterfly, but of course needs logic to separate the butterflies that need it from those that don't, too.

This is the final touch on the asymmetric load-balancing.

# Chapter 5: Conclusions and Future Work

Most of the existing research on actual efficiency of the FFT on modern multi-core computing platforms takes a black-box approach: using heuristics experimentally search through a space of solutions to find the most efficient one, and then benefit from reusing the plan for the same problem size and for the same platform multiple times.

This work employed a white-box approach - under the following natural assumptions: a) the exact size of the DFT problem cannot always be known in advance, b) the data arrives incrementally as a result of sampling and some initial processing, c) the computing platform fits the ideal cache model and the tall cache assumption, d) the processor cores are not many but modern enough to use pipelining/vectorization, and e) some cores may be better than others at performing particular type of computation, we suggest a matching asymmetrical method of parallelisation with a threading strategy, show its benefits and prove optimality in important aspects of the computation. We also invent a more efficient way of computing the twiddle factors, and suggest a data structure to store them that works well under the notion of incrementality of the application. All these mathematically sound ideas need to be tested experimentally.

Suggested topics for future research could focus on exploring the level of granularity of inter-thread synchronisation (spin-locks), so that overhead and delays are optimal; herein the granularity strides between power-of-two sizes, which may not be the best.

# Bibliography

[1] Alok Aggrawal, Jeffrey Scott Vitter;
"The Input/Output Complexity of Sorting and Related Problems".
*Communications of the ACM, Sept 1988, Vol.31 No.9*, p. 1116-1127.

[2] Ayaz Ali, Lennart Johnsson, Jaspal Subhlok;
"Scheduling FFT computation on SMP and Multicore Systems".
*Proc ICS'07, June 18-20, 2007 Seattle WA, USA*, p. 293-301.

[3] Gene Amdahl; "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities".
AFIPS Press, *American Federation of Information Processing Societies Conference Proceedings, (30), 1967,* p. 483-485.

[4] Jen-Chuan Chi, Sau-Gee Chen; "An Efficient FFT Twiddle Factor Generator".
*Proc. XII$^{th}$ EU Signal-Processing Conference,* EUSIPCO *2004, Vienna,* p.1533-1536.

[5] Cooley, J., Tukey, J.; "An Algorithm for the Machine Computation of Complex Fourier Series". *Mathematics of Computation 19 (1965)*, p. 297-301.

[6] T. H. Cormen, C. E. Leiserson, and R. L.Rivest;
"Introduction to Algorithms".
©2006 *McGraw Hill, first edition MIT Press* 1990.

[7] Franz Franchetti, Yevgen Voronenko, Markus Püschel;
"FFT Program Generation for Shared Memory: SMP and Multicore".
*Proc. 2006 ACM/IEEE Conference on Supercomputing SC'06*, p. 51.

[8] Matteo Frigo, Steven G. Johnson; "The Design and Implementation of FFTW3".
*Proceedings of the IEEE, Vol.93 No.2, Feb 2005*, p.216-231.

[9] Matteo Frigo, Charles E. Leiserson, Harold Prokop, Sirdhar Ramachandran; "Cache-oblivious algorithms".
*40th Annual Symposium on Foundations of Computer Science 1999*, p. 285-297.

[10] Jianjun Guo, Mingche Lai, Zhengyuan Pang, Libo Huang, Fangyuan Chen, Kui Dai, and Zhiying Wang; "Memory System Design for a Multi-core Processor".
*Proc. 2008 Int. Conf. Complex, Intelligent and Software Intensive Systems* p.601-606.

[11] J.L. Gustafson; "Reevaluating Amdahl's Law".
*Communications of the ACM, May 1988,* p. 532-533.

[12] Mark D. Hill (University of Wisconsin-Madison), Michael R. Marty (Google); "Amdahl's Law in the Multicore Era".
*Computer, July 2008*, p. 33-38.

[13] Joris van der Hoeven; "Notes on the Truncated Fourier Transform".
2007 (unpublished - http://www.texmacs.org/joris/tft/tft-abs.html)

[14] Steven G. Johnson, Matteo Frigo;
"A Modified Split-Radix FFT with Fewer Arithmetic Operations";
*IEEE Transactions on Signal Processing, 55(1)*, p.111-119.

[15] M.C. Pease; "Adaptation of the Fast Fourier Transform for Parallel Processing".
*Journal of the ACM, Vol.15 No.2, April 1968,* p. 252-264.

[16] Bryan Singer, Manuela Veloso; "Learning to Construct Fast Signal Processing Implementations". *Journal of Machine Learning Research 3 (2002)*, p.887-919.

[17] Guangming Tan, Ninghui Sun, and Guang R. Gao; "Improving Performance of Dynamic Programming via Parallelism and Locality on Multicore Architectures".
IEEE *Trans. on Parallel and Distributed Systems, Vol. 20/2, Feb 2009*, p. 261-274.