Efficient FFT Algorithms for Mobile Devices

Koki Sugawara

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 24.9.2016

Thesis supervisors:

Prof. Risto Wichman

Thesis advisors:

Prof. Mario Di Francesco

M.Sc. Pranvera Kortoçi

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Koki Sugawara

Title: Efficient FFT Algorithms for Mobile Devices

Date: 24.9.2016 Language: English Number of pages: 5+36

Department of Signal Processing and Acoustics

Professorship: S-88 Signal Processing

Supervisor: Prof. Risto Wichman

Advisors: Prof. Mario Di Francesco, M.Sc. Pranvera Kortoçi

Increased traffic on wireless communication infrastructure has exacerbated the limited availability of radio frequency (RF) resources. Spectrum sharing is a possible solution to this problem that requires devices equipped with Cognitive Radio (CR) capabilities. A widely employed technique to enable CR is real-time RF spectrum analysis by applying the Fast Fourier Transform (FFT). Today's mobile devices actually provide enough computing resources to perform not only the FFT but also wireless communication functions and protocols by software according to the software-defined radios paradigm. In addition to that, the pervasive availability of mobile devices make them powerful computing platform for new services. This thesis studies the feasibility of using mobile devices as a novel spectrum sensing platform with focus on FFT-based spectrum sensing algorithms. We benchmark several open-source FFT libraries on an Android smartphone. We relate the efficiency of calculating the FFT to both algorithmic and implementation-related aspects. The benchmark results also show the clear potential of special FFT algorithms that are tailored for sparse spectrum detection.

Keywords: Cognitive radios, spectrum sensing, crowdsourcing, fast Fourier Transform, compressive sensing, sparse FFT, mobile devices, software-defined radio, performance evaluation

Preface

Foremost, I would like to thank Professor Risto Wichman for his general guidance of my thesis work. He saved my degree study after my ten years of absence from university, and helped me to find this interesting topic. My greatest gratitude goes to Professor Mario Di Francesco for giving me this opportunity and detailed instructions. Without his guidance I could not complete this work. I am grateful to Pranvera Kortoçi for navigating my work and inspiring technical discussions. She was the guiding light when I was lost in darkness at the very beginning of this project. I also thank Gopika Premsankar for her informative resources on installation of the sFFT evaluation program, and Hieu Nguyen for his hands-on help in generating scientific graphs.

Finally, I am thankful to my family for their patience and always being the source of my motivation.

Otaniemi, 24.9.2016

Koki Sugawara

Contents

Abbreviations

3GPP	3rd Generation Partnerchip Project
ADC	Analog-to-Digital Converter
AVX	Advanced Vector Extensions
CPU	Central Processing Unit
CR	Cognitive Radio
CS	Compressed Sensing or Compressive Sensing
DAC	Digital-to-Analog Converter
DC	Direct Current
DFT	Discrete Fourier Transform
DSP	Digital Signal Processor
DTFT	Discrete-Time Fourier Transform
DVB-T	Digital Video Broadcasting, Terrestrial
FCC	Federal Communications Commission
FDD	Frequency Division Duplex
FPGA	Field-Programmable Gate Array
FFT	Fast Fourier Transform
FFTS	Fastest Fourier Transform in the South
FFTW	Fastest Fourier Transform in the West
FT	Fourier Transform
GNU	GNU's Not Unix
GPS	Global Positioning System
GPU	Graphics Processing Unit
GSM	Groupe Spécial Mobile
IEEE	Institute of Electrical and Electronics Engineers
IFFT	Inverse FFT
IQ	In-phase and Quadrature-phase
JNI	Java Native Interface
LF	Low Frequency
LSB	Least Significant Bit
LTE	Long-Term Evolution
MF	Medium Frequency
MIT	Massachusetts Institute of Technology
MSB	Most Significant Bit
Msps	Million Samples per Second
NDK	Native Development Kit
OS	Operating System
PC	Personal Computer
RAM	Random Access Memory
RF	Radio Frequency
SDR	Software-Defined Radio
sFFT	Sparse FFT
SIMD	Single-Instruction, Multiple-Data
SISD	Single-Instruction, Single-Data
SRC	Sample Rate Converter
SSE	Streaming SIMD Extensions
TDD	Time Division Duplex
USB	Universal Serial Bus
USRP	Universal Software Radio Peripheral
WLAN	Wireless Local Area Network
XML	Extensible Markup Language

1 Introduction

1.1 Research scope and goals

The widespread popularity of both mobile devices and Internet services with rich multimedia content have significantly increased the traffic on the wireless communication infrastructure, thereby exacerbating the limited availability of radio frequency (RF) resources. Spectrum sharing is a possible solution to this problem which allows unlicensed users to access to licensed RF spectrum as long as they do not produce harmful interference. This scheme requires mobile devices equipped with Cognitive Radio (CR) capabilities to adapt radio-system parameters according to the radio communication environment in real-time.

A widely employed technique to enable CR is real-time RF spectrum analysis by applying the Discrete Fourier Transform (DFT) on RF signals. The DFT decomposes time-domain RF signals into distribution of coefficients in the frequency domain. These coefficients represent the signal power at the corresponding frequency, and therefore can be interpreted as occupied frequency bands. The idea behind the Fast Fourier Transform (FFT) was first published by Cooley and Tukey [9] in 1965 for efficient calculation of the DFT. Since then, the FFT has been applied to a number of research and engineering fields. Nowadays several FFT algorithms and implementations are available as free or open-source libraries.

Even though most of the libraries target PC (personal computer) applications, today's mobile devices integrate powerful processors and sufficient resources to perform the FFT by software. If coupled with software-defined radio (SDR) hardware, these computing capabilities can be employed to execute wireless communication functions and protocols. Researchers regard the pervasive availability of mobile devices with suitable capability as a novel spectrum sensing platform.

This thesis studies the feasibility of using mobile phones in this context, with focus on FFT-based algorithms for spectrum sensing. We benchmark several open-source FFT libraries on an Android smartphone and evaluate their performance in terms of average execution time. We relate the efficiency of calculating the FFT to both algorithmic and implementation-related aspects.

1.2 Contributions

In this thesis we review the benefits of CR to address the scarcity of RF spectrum for wireless communications. We indicate that spectrum sensing is the essential enabling function of CR. In addition to that, we also discuss leveraging widespread mobile devices as a pervasive computing platform, and provide examples of applying such a platform to spectrum sensing.

As one of the core techniques for spectrum sensing, we discuss the FFT algorithm in detail and review solutions to improve FFT performance. Specifically, we study special FFT algorithms that are tailored for sparse spectrum, namely, compressed sensing (CS) and the sparse FFT. We review well-reputed open-source FFT libraries, and describe the benchmarking software we developed to evaluate the FFT libraries

on an Android device. Based on the evaluation results, we discuss the impact of implementation languages and support of SIMD (single-instruction, multiple-data) instruction sets on the FFT performance. Finally, we suggest further research directions in this topic.

1.3 Thesis structure

The rest of the thesis is organized as follows. Chapter 2 introduces the background on CR, SDR, and FFT. Chapter 3 provides the basis of FFT, a few derivations of the FFT algorithm, and open-source FFT libraries. Chapter 4 details our benchmarking software and the device used for evaluation purposes. Chapter 5 presents benchmarking results of several FFT libraries with estimation of impact of the transform decomposition technique. Finally, Chapter 6 concludes the thesis.

2 Background

The goal of this thesis is to evaluate efficient algorithms for computing the Fast Fourier Transform (FFT) on mobile devices. These algorithms allow turning smart phones and other portable devices into software-defined radio platforms. In the rest of this chapter, we first describe the cognitive radio and software-defined radio paradigms, then introduce the FFT as well as compressed sensing, which is a powerful algorithm to solve sparse problems. We conclude by a review of research on using mobile phones as software-defined radio and spectrum sensing platforms.

2.1 Cognitive radios and software-defined radios

Radio frequency spectrum is a shared and limited physical resource. Until now, regulatory bodies have managed the available spectrum by dividing frequency bands in fragments, and by assigning them to licensed users for exclusive operations.

At the user side, development in electronics and radio communication technologies has allowed millions of mobile devices to access several different radio networks constantly. As multimedia content and services become more and more widespread, mobile traffic increases dramatically.

These trends have led to overloaded cellular bands and to the lack of wireless spectrum. To overcome these problems, regulatory bodies in the world have been considering to implement shared spectrum access policy in licensed bands by unlicensed users. In this policy unlicensed users must comply two conditions to protect the licensees' spectrum usage. Unlicensed users can access the spectrum on opportunistic basis, or only when free spectrum is detected, and as far as they do not cause interference to priority users.

Cognitive Radio (CR) is a paradigm to achieve the above-mentioned functions. It is defined by FCC as "a radio that can change its transmitter parameters based on interaction with the environment in which it operates" [10]. CR systems monitor the environment or the context of a user equipment – such as the available spectrum, surrounding noise, moving speed, remaining battery – and adapt its radio parameters dynamically to such a context. Spectrum-sensing CR denotes a subset of full CR that detects and utilizes the best available radio spectrum given the current context.

Software-Defined Radio (SDR) is an advanced enabler technology of CR. While the traditional radio system achieves its system function by switching the data path among a collection of rigid functional blocks (Fig. 1), an SDR system leverages generic functional blocks that are configurable with parameter settings or software (Fig. 2). The ideal SDR would configure all the digital front-end functions of a radio system – such as filters, sample-rate converters, and protocol operations (including modulation, demodulation, error correction, encryption, network protocol) – totally by software on general-purpose digital-signal processors (DSPs) (Fig. 3). Such a software-centric radio system is capable of adapting its system functions to the given context more flexibly than a traditional radio system with fixed functional components.

Implementation of the SDR architecture started in 1980's in the field of digital

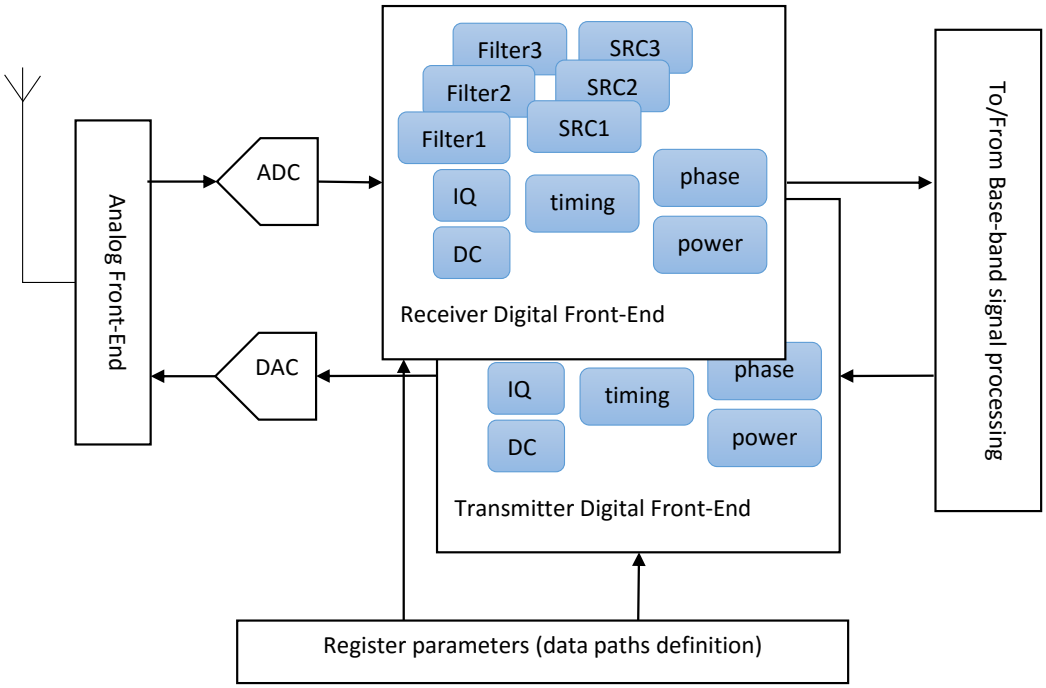

Figure 1: Traditional radio system.

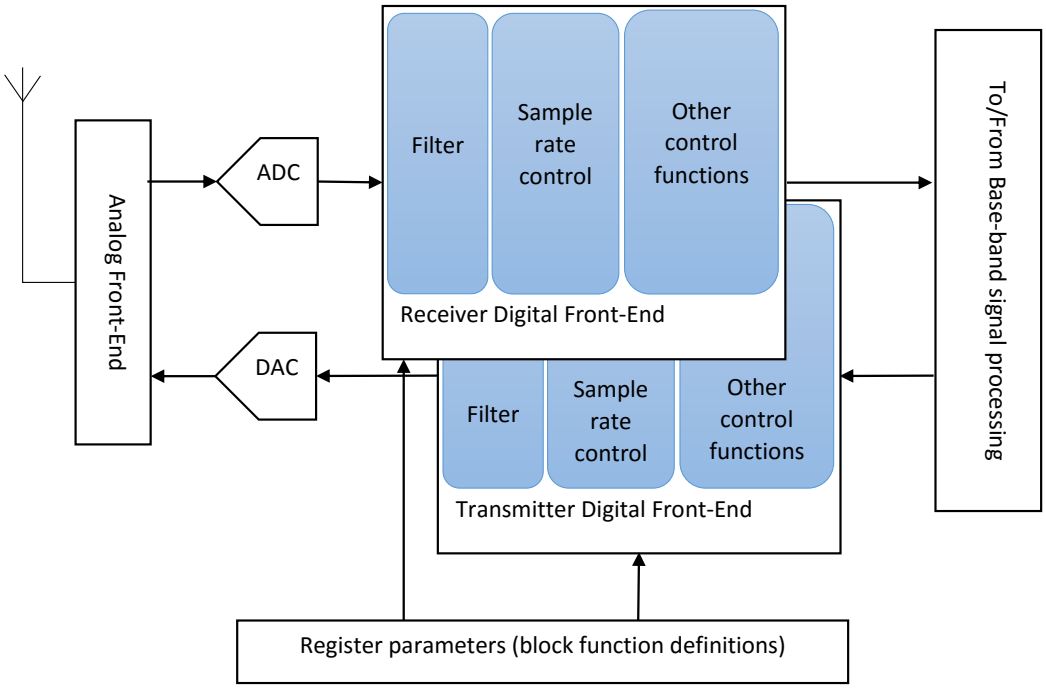

Figure 2: Software-defined radio with generic functional blocks.

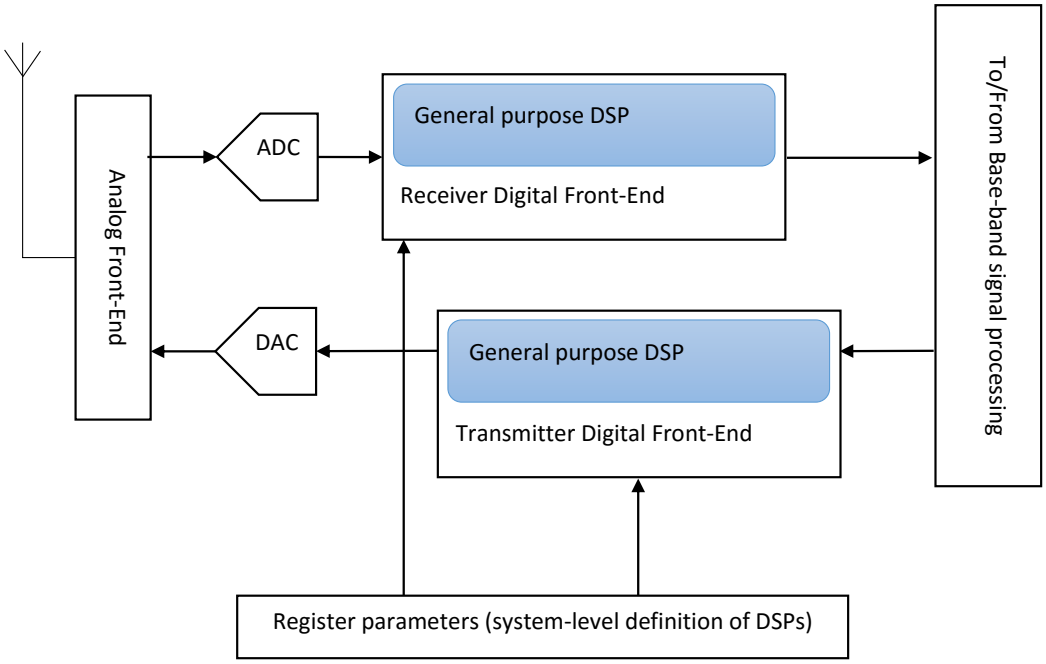

Figure 3: The ideal software-defined raido with general purpose DSPs.

receivers, and in 1991 a GSM base station was built as the first software-based transceiver [23]. A present SDR platform consists of the following components in general; an ADC, a DSP implemented in FPGA, clock and synchronization, a DAC when it supports transmitter, and digital interface, such as Ethernet or USB, to a host computer. Nowadays tens of SDR platforms are available to consumers at affordable price for various purposes such as research, TV/FM/GPS receivers, transceivers, and radars. To name a few examples, Universal Software Radio Peripheral (USRP) is provided as "open source hardware" and used with GNU Radio, an open source software to implement SDR and signal processing. Another module RTL-SDR(RTL2832U) is originally an USB dongle DVB-T TV tuner at $20, but it works as an SDR platform with a software driver [1].

2.2 The Fast Fourier Transform

When implementing spectrum-sensing CR, spectrum analysis is the first step to observe real-time usage of frequency bands. Since the status of spectrum utilization fluctuates over time, such an analysis needs to complete quickly. Due to the limited energy resources in mobile devices, the analysis needs to accomplish with low power consumption, too.

The Fourier Transform (FT) is a mathematic technique to convert a continuous time-domain function into a collection of periodical function components or frequency elements. The transformed result is a complex-valued function of frequency. The absolute value of the transform represents the amplitude of the frequency element, while the argument of the complex value represents the phase offset of the frequency

element. Joseph Fourier discovered this valuable transform in 1822 [11].

The Discrete Fourier Transform (DFT) is a FT with certain assumptions driven by real-world numerical computing. These assumptions include: the data are sampled at a constant rate; the number of data, or data length, is limited and known; and the whole data set forms one long period of data that defines the frequency resolution in the transform. The DFT converts a finite length of sampled signal into complex coefficients of frequency elements.

The Fast Fourier Transform (FFT) is an algorithm that optimizes the number of operations in the DFT by fully utilizing its regularity. The FFT algorithm was first officially published by J.F. Cooley and J.F. Tukey in 1965 [9], though its core idea is found already in a document of the early 19th century by C.F. Gauss [17]. Section 3 describes the FFT algorithm in detail.

2.3 Compressed sensing

The Nyquist-Shannon sampling theorem defines the lowest sampling frequency that allows perfect recovery of the original signal. When the highest frequency component of the original time-continuous signal is limited to B, the sample rate F_s must be at least twice the highest frequency, i.e.,

$$F_s \geq 2B \tag{1}$$

The threshold, $F_s = 2B$, is called the Nyquist rate [8].

The Nyquist-Shannon theorem indicates that the higher the frequency of interest is, the higher the sampling rate must be, by increasing the number of sample data in unit time. The input data length N of the FFT defines the order of FFT operation as $O(N \log N)$. Even though the FFT calculates the DFT efficiently, the data length still dominates the operation time, and it takes more time when the target frequency increases. Thus, more efficient methods are sought.

Compressed sensing (or compressive sensing, CS) is a technique to solve an underdetermined set of linear equations [7] [15]. A target set of linear equations can be generally described in the following form:

$$\mathbf{A}\mathbf{x} = \mathbf{y} \tag{2}$$

where $\mathbf{y} \in \mathbb{R}^m$ is a measurement vector, $\mathbf{A} \in \mathbb{R}^{m \times n}$ is a measurement matrix, and $\mathbf{x} \in \mathbb{R}^n$ is the unknown vector in the linear system. Here, both $\mathbf{y}$ and $\mathbf{A}$ are exactly known. In case of $m < n$, where the number of measurement is less than that of unknowns, the equation is underdetermined and there are multiple solution candidates. To choose one solution from the candidates, we need to provide a criterion of choice. An approach is minimizing the norm[1]of the candidates. A commonly used criterion is minimizing the ℓ_2-norm or the Euclidean length of the vector. Fig. 4 presents the form of the solution space depending on the specified criterion in the two-dimensional space. The ℓ_2-norm criterion provides a circular solution space to seek for a suitable solution and the Euclidean distance from the origin of the solution space is minimized as the $p = 2$ case in Fig. 4.

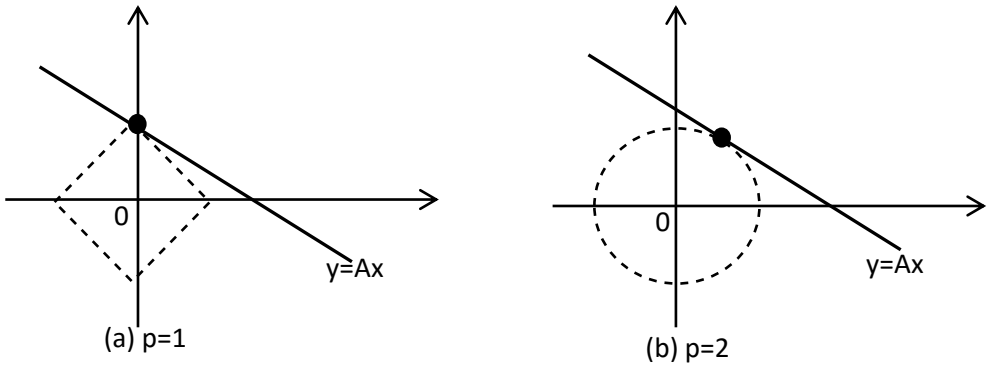

Figure 4: Impact of the ℓ_p norm criterion on the solution space of an identical linear equation for: (a) $p = 1$; (b) $p = 2$. [15]

As Fig. 4 shows, a criterion by ℓ_1-norm forms a diamond-shaped solution space in two dimensions. The figure illustrates that a solution is most likely to be found at a vertex of the solution space. Since the vertices reside on the axes, the solution vector obtained with an ℓ_1-norm criterion is sparser than the solution with an ℓ_2-norm criterion. CS thus obtains a sparse solution by applying the ℓ_1-norm criterion with an underdetermined equation as follows:

$$\min \|\mathbf{x}\|_1 \text{ subject to } \mathbf{Ax} = \mathbf{y} \tag{3}$$

The authors in [15] list applications of CS such as wireless channel estimation, data aggregation in a wireless sensor network, network tomography, and compressed spectrum sensing for cognitive radios.

2.4 Mobile devices as spectrum sensing platforms

Most of mobile devices available today – such as smartphones, tablets, navigators, and note PCs – are capable of accessing the Internet through multiple radio access technologies, such as cellular networks and WLAN standards.

With regard to their computing performance, latest mobile devices are equipped with modern multi-core processors that support SIMD (Single Instruction, Multiple Data) instruction set. A SIMD instruction executes one operation on multiple sets

[1] A norm is a function in a vector space that leads to a scalar value of a vector. This value represents the length or size of the vector. For a $p \geq 1$, p-norm is defined as follows.

$$\|x\|_p \equiv \left(\sum_{i=1}^{n} |x_i|^p \right)^{\frac{1}{p}}$$

When $p = 2$, the norm provides the length of the vector. This is called Euclidean norm, ℓ_2 norm, or just 2-norm. When $p = 1$, the norm is the sum of absolute value of each vector element, or the Manhattan distance of the vector. This is called taxicab norm, ℓ_1 norm, or just 1-norm.

of data in parallel to boost performance especially on multimedia data. As of this writing, ARM is the most popular processor architecture in mobile devices, and their ARMv7 or later processor architecture support their own SIMD instruction set called NEON [2].

With increased performance, mobile devices have become capable enough to be employed as SDR platforms instead of field-programmable gate arrays (FPGAs) or personal computers, the first and second generation SDR platforms, respectively. Park et al. [28] prove that a smartphone with a 1.2 GHz processor can already execute a low-speed radio protocol such as ZigBee (IEEE 802.15.4), and estimate that development of processors will enable a smartphone to run WiFi (IEEE 802.11a/b) protocols as its applications within six years.

To understand the requirements for practical spectrum sensing on mobile devices, let us consider access to 4G cellular bands. Among the LTE-FDD bands, Band 3 has the widest uplink and downlink bandwidths of 75 MHz each [3]. Since duplex operation is defined in the standard, we need to sense only one of either the uplink or the downlink spectrum, i.e. 75 MHz, to detect spectrum usage. The number of 15 kHz sub-carriers in a 75 MHz band is 5,000 Usually the size of FFT is set to a power-of-two value for efficiency reasons. In this case the FFT size should be $N = 2^{13} = 8{,}192$ to cover 5,000 points. The required sampling rate to achieve this FFT is $8{,}192 \cdot 15kHz = 122.88$ Msps. For the case of LTE-TDD, the widest bandwidth of 200 MHz (e.g. Band 43) contains 13,333 of 15 kHz channels. The FFT size should be $N = 2^{14}$, and the required sampling rate is 245.76 Msps. Commercial discrete ADC components for LTE devices support at least 250 Msps per channel [19] [32].

In industrialized countries, most people always bring one or more mobile devices every day wherever they go. Such a new physical circumstance with widespread high-performance wireless mobile devices as a pervasive computing infrastructure enables large-scale spectrum sensing through crowdsourcing.

Nika et al. [25] implement low-cost spectrum monitors by applying RTL-SDR as the RF receiver and mobile devices (including a smartphone) for calculating FFT. They summarize that limited sensitivity and bandwidth of their low-cost spectrum monitors can be compensated with increased data collected by crowdsourcing. One critical point in their spectrum monitor is the decreased RF sensitivity of RTL-SDR when powered by mobile devices via microUSB connector.

Zhang et al. [36] utilize the WiFi chip of a smartphone to measure spectrum of TV channels. They achieve this by introducing an RF frequency translator that consists of a frequency synthesizer and a mixer. The original TV channel signal is frequency shifted to WiFi frequency, and is then received with the smartphone. The smartphone can detect TV channels with reasonable accuracy.

Achtzehn et al. [4] monitor network load on 2G cellular operators by crowdsourcing. Their open-source mobile phones collect not only RF spectrum but also protocol control signals to estimate weekly and daily variations of network traffic. Their cloud backend system schedules policy-based data collection by all participating mobile devices.

Shi et al. [29] adapt parameters of WiFi access points by utilizing smartphones to collect the WiFi signal quality. Their unique crowdsourcing architecture avoids

interrupting an actively communicating device (e.g. a laptop computer) and utilizes an idle device in its proximity (e.g. a smartphone) for the WiFi signal sensing, if available. Since not many WiFi chipsets in smartphones allow detailed measurements because of the limitations in hardware, firmware, or in the driver, they implemented a special hardware for their prototype.

Some of above authors [36] [4] [29] also discuss the benefits of incentive mechanisms, such as a spectrum data market, to attract smartphone users into the signal measurement campaign.

3 The Fast Fourier Transform

This chapter is dedicated to the Fast Fourier Transform (FFT). We first introduce the relation between Fourier Transform (FT) and FFT by focusing on their operations. Then we present a few algorithms to improve the FFT performance, namely, FFT pruning, transform decomposition, and the sparse FFT. In the last section, we review open-source FFT software that realizes the previously-mentioned techniques.

3.1 The original FFT algorithm

Before discussing the FFT, we start by introducing the FT and the Discrete Fourier Transform (DFT) [8] [20] [16].

3.1.1 Fourier Transform

The Fourier series defines a way to represent a general time-continuous periodic function $x(t)$ with period $T_0 = 1/f_0$ as a sum of basic periodic functions. By applying sin and cos functions as the basis functions, $x(t)$ is expressed as follows:

$$x(t) = \frac{a_0}{2} + \sum_{k=1}^{\infty} a_k \cos(2\pi f_0 k t) + \sum_{k=1}^{\infty} b_k \sin(2\pi f_0 k t) \tag{4}$$

where

$$a_k = \frac{2}{T_0} \int_{T_0} x(t) \cos(2\pi f_0 k t)\, dt \tag{5}$$

$$b_k = \frac{2}{T_0} \int_{T_0} x(t) \sin(2\pi f_0 k t)\, dt \tag{6}$$

The coefficients a_k and b_k express the correlation between $x(t)$ and the basis functions.

By using Euler's formula with j as the imaginary unit,

$$e^{j\theta} = \cos\theta + j \sin\theta \tag{7}$$

the Fourier series can be expressed in complex form:

$$x(t) = \sum_{k=-\infty}^{\infty} c_k e^{j2\pi f_0 k t} \tag{8}$$

Here, the rotation vector $e^{j2\pi f_0 k t}$ represents a combination of basis functions sin and cos with frequency $k f_0$. The complex coefficient c_k again represents the correlation between $x(t)$ and basis functions of frequency $k f_0$. The coefficients are defined as follows:

$$c_k = \frac{1}{T_0} \int_{T_0} x(t) e^{-j2\pi f_0 k t}\, dt \tag{9}$$

The Fourier transform solves this coefficient across the continuous frequency space, and decomposes the frequency elements of the original signal $x(t)$ as a continuous function of frequency as follows:

$$X(f) = \int_{-\infty}^{\infty} x(t) e^{-j2\pi f t}\, dt \tag{10}$$

3.1.2 Discrete Fourier Transform

Let us introduce practical conditions of real-world computing in the FT. The first consideration is the form of the time-domain signal. In order to handle practical signals, the original signal $x(t)$ must be sampled at a regular interval to obtain the data sequence $x[n]$, i.e.,

$$x[n] = x(nT_s) \tag{11}$$

where T_s is the sampling interval, or $f_s = 1/T_s$ is the sampling rate. The expression of the corresponding time-continuous signal $x(t)$ with the sample values $x[n]$ is:

$$x(t) = \sum_{n=-\infty}^{\infty} x[n]\delta(t - nT_s) \tag{12}$$

By substituting $x(t)$ in the definition of Fourier Transform Eq.(10) with Eq.(12), and by taking into account that $\int_{-\infty}^{\infty} \delta(t - t_0)f(t)dt = f(t_0)$, we reach the definition of discrete-time Fourier transform (DTFT):

$$\begin{aligned} X(f) &= \int_{-\infty}^{\infty} \left\{ \sum_{n=-\infty}^{\infty} x[n]\delta(t - nT_s) \right\} e^{-j2\pi ft}\, dt \\ &= \sum_{n=-\infty}^{\infty} x[n] \int_{-\infty}^{\infty} \delta(t - nT_s)\, e^{-j2\pi ft}\, dt \\ &= \sum_{n=-\infty}^{\infty} x[n]\, e^{-j2\pi nf/f_s} \end{aligned} \tag{13}$$

The second consideration is the finite data length. Indeed the data length must be finite, so that the calculations will complete in finite time. When the data length is N, the index range of the summation becomes $n = [0..N-1]$, thus

$$X(f) = \sum_{n=0}^{N-1} x[n]\, e^{-j2\pi nf/f_s} \tag{14}$$

When the sample signal of length N contains precisely one period, the signal phase shifts $2\pi/N$ at each sampling interval. This is the smallest representable angular frequency or resolution with the N-length sample. The DFT calculates the frequency coefficients with this resolution from 0 to $\frac{2\pi(N-1)}{N}$, or at $\frac{2\pi k}{N}$ where $k = [0..N-1]$. The frequency coefficients are, again, the correlation between $x[n]$ and the rotation vectors of frequency sets defined by $\frac{2\pi k}{N}$. Since the vector phase increases at this rate, the n-th data is multiplied with the vector at phase $\frac{2\pi kn}{N}$, leading to:

$$X[k] = \sum_{n=0}^{N-1} x[n]\, e^{-j\frac{2\pi}{N}kn} \tag{15}$$

By replacing the indices with suffixes for simplicity, we have

$$X_k = \sum_{n=0}^{N-1} x_n\, e^{-j\frac{2\pi}{N}kn} \tag{16}$$

3.1.3 Fast Fourier Transform

The FFT utilizes regularities in DFT equations to calculate them in a fast manner.

First, let us define the N-th root of unity (1) as

$$W \equiv e^{-j\frac{2\pi}{N}} \tag{17}$$

and simplify the DFT in Eq.(16) as follows:

$$X_k = \sum_{n=0}^{N-1} x_n W^{kn} \tag{18}$$

To observe the regularity, we show the complete matrix equation for the case of $N = 8$ below:

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \end{bmatrix} = \begin{bmatrix} W^{0\cdot0} & W^{0\cdot1} & W^{0\cdot2} & W^{0\cdot3} & W^{0\cdot4} & W^{0\cdot5} & W^{0\cdot6} & W^{0\cdot7} \\ W^{1\cdot0} & W^{1\cdot1} & W^{1\cdot2} & W^{1\cdot3} & W^{1\cdot4} & W^{1\cdot5} & W^{1\cdot6} & W^{1\cdot7} \\ W^{2\cdot0} & W^{2\cdot1} & W^{2\cdot2} & W^{2\cdot3} & W^{2\cdot4} & W^{2\cdot5} & W^{2\cdot6} & W^{2\cdot7} \\ W^{3\cdot0} & W^{3\cdot1} & W^{3\cdot2} & W^{3\cdot3} & W^{3\cdot4} & W^{3\cdot5} & W^{3\cdot6} & W^{3\cdot7} \\ W^{4\cdot0} & W^{4\cdot1} & W^{4\cdot2} & W^{4\cdot3} & W^{4\cdot4} & W^{4\cdot5} & W^{4\cdot6} & W^{4\cdot7} \\ W^{5\cdot0} & W^{5\cdot1} & W^{5\cdot2} & W^{5\cdot3} & W^{5\cdot4} & W^{5\cdot5} & W^{5\cdot6} & W^{5\cdot7} \\ W^{6\cdot0} & W^{6\cdot1} & W^{6\cdot2} & W^{6\cdot3} & W^{6\cdot4} & W^{6\cdot5} & W^{6\cdot6} & W^{6\cdot7} \\ W^{7\cdot0} & W^{7\cdot1} & W^{7\cdot2} & W^{7\cdot3} & W^{7\cdot4} & W^{7\cdot5} & W^{7\cdot6} & W^{7\cdot7} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} \tag{19}$$

The matrix in Eq.(19) clearly shows the relation of the rotation vector phase (W^{kn}) to the target frequency index k of X_k and to the data index n of x_n. Another fact in the matrix is that the data length N limits the range of k in $[0..N-1]$ to maintain the unique rotation-vector matrix. Otherwise, all the column vectors return to their original values after N elements because of aliasing ($W^{(k+N)n} = W^{kn}$).

The matrix in Eq.(19) can be simplified by first applying $W^{(p \mod N)} = W^p$. Even though $W^0 = e^{-j\cdot0} = 1$, we keep the form of W^p at this point to observe the regularity in the matrix:

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \end{bmatrix} = \begin{bmatrix} W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 \\ W^0 & W^1 & W^2 & W^3 & W^4 & W^5 & W^6 & W^7 \\ W^0 & W^2 & W^4 & W^6 & W^0 & W^2 & W^4 & W^6 \\ W^0 & W^3 & W^6 & W^1 & W^4 & W^7 & W^2 & W^5 \\ W^0 & W^4 & W^0 & W^4 & W^0 & W^4 & W^0 & W^4 \\ W^0 & W^5 & W^2 & W^7 & W^4 & W^1 & W^6 & W^3 \\ W^0 & W^6 & W^4 & W^2 & W^0 & W^6 & W^4 & W^2 \\ W^0 & W^7 & W^6 & W^5 & W^4 & W^3 & W^2 & W^1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} \tag{20}$$

In the matrix in Eq.(20) we can observe a periodc property in each row vector with an even index. The right half of each even-index row vector is an exact copy of the left half. This allows to utilize only the left half of the matrix by combining the operands that apply identical coefficients. X_k with even k indices can be simplified as follows:

$$\begin{bmatrix} X_0 \\ X_2 \\ X_4 \\ X_6 \end{bmatrix} = \begin{bmatrix} W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 \\ W^0 & W^2 & W^4 & W^6 & W^0 & W^2 & W^4 & W^6 \\ W^0 & W^4 & W^0 & W^4 & W^0 & W^4 & W^0 & W^4 \\ W^0 & W^6 & W^4 & W^2 & W^0 & W^6 & W^4 & W^2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix}$$
$$= \begin{bmatrix} W^0 & W^0 & W^0 & W^0 \\ W^0 & W^2 & W^4 & W^6 \\ W^0 & W^4 & W^0 & W^4 \\ W^0 & W^6 & W^4 & W^2 \end{bmatrix} \begin{bmatrix} x_0 + x_4 \\ x_1 + x_5 \\ x_2 + x_6 \\ x_3 + x_7 \end{bmatrix} \tag{21}$$

The same periodic property becomes visible in Eq.(20) for odd k indices by factorizing the right half of the matrix by W^4.

$$\begin{bmatrix} X_1 \\ X_3 \\ X_5 \\ X_7 \end{bmatrix} = \begin{bmatrix} W^0 & W^1 & W^2 & W^3 & W^4 & W^5 & W^6 & W^7 \\ W^0 & W^3 & W^6 & W^1 & W^4 & W^7 & W^2 & W^5 \\ W^0 & W^5 & W^2 & W^7 & W^4 & W^1 & W^6 & W^3 \\ W^0 & W^7 & W^6 & W^5 & W^4 & W^3 & W^2 & W^1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix}$$
$$= \begin{bmatrix} W^0 & W^1 & W^2 & W^3 & W^{4+0} & W^{4+1} & W^{4+2} & W^{4+3} \\ W^0 & W^3 & W^6 & W^1 & W^{4+0} & W^{4+3} & W^{4+6} & W^{4+1} \\ W^0 & W^5 & W^2 & W^7 & W^{4+0} & W^{4+5} & W^{4+2} & W^{4+7} \\ W^0 & W^7 & W^6 & W^5 & W^{4+0} & W^{4+7} & W^{4+6} & W^{4+5} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} \tag{22}$$
$$= \begin{bmatrix} W^0 & W^1 & W^2 & W^3 & W^0 & W^1 & W^2 & W^3 \\ W^0 & W^3 & W^6 & W^1 & W^0 & W^3 & W^6 & W^1 \\ W^0 & W^5 & W^2 & W^7 & W^0 & W^5 & W^2 & W^7 \\ W^0 & W^7 & W^6 & W^5 & W^0 & W^7 & W^6 & W^5 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 W^4 \\ x_5 W^4 \\ x_6 W^4 \\ x_7 W^4 \end{bmatrix}$$
$$= \begin{bmatrix} W^0 & W^1 & W^2 & W^3 \\ W^0 & W^3 & W^6 & W^1 \\ W^0 & W^5 & W^2 & W^7 \\ W^0 & W^7 & W^6 & W^5 \end{bmatrix} \begin{bmatrix} x_0 - x_4 \\ x_1 - x_5 \\ x_2 - x_6 \\ x_3 - x_7 \end{bmatrix}$$

The final derivation utilized the property of $W^{\frac{N}{2}} = -1$, where $N = 8$ in the considered example.

We can further simplify DFT calculations by finding matrix periodicity in Eq.(21) and Eq.(22). From Eq.(21),

$$\begin{aligned}\begin{bmatrix}X_0\\X_4\end{bmatrix} &= \begin{bmatrix}W^0 & W^0 & W^0 & W^0\\W^0 & W^4 & W^0 & W^4\end{bmatrix}\begin{bmatrix}x_0+x_4\\x_1+x_5\\x_2+x_6\\x_3+x_7\end{bmatrix}\\ &= \begin{bmatrix}W^0 & W^0\\W^0 & W^4\end{bmatrix}\begin{bmatrix}(x_0+x_4)+(x_2+x_6)\\(x_1+x_5)+(x_3+x_7)\end{bmatrix}\\ &= \begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}\begin{bmatrix}(x_0+x_4)+(x_2+x_6)\\(x_1+x_5)+(x_3+x_7)\end{bmatrix}\end{aligned} \tag{23}$$

$$\begin{aligned}\begin{bmatrix}X_2\\X_6\end{bmatrix} &= \begin{bmatrix}W^0 & W^2 & W^4 & W^6\\W^0 & W^6 & W^4 & W^2\end{bmatrix}\begin{bmatrix}x_0+x_4\\x_1+x_5\\x_2+x_6\\x_3+x_7\end{bmatrix}\\ &= \begin{bmatrix}W^0 & W^2\\W^0 & W^6\end{bmatrix}\begin{bmatrix}(x_0+x_4)+(x_2+x_6)W^4\\(x_1+x_5)+(x_3+x_7)W^4\end{bmatrix}\\ &= \begin{bmatrix}1 & W^2\\1 & -W^2\end{bmatrix}\begin{bmatrix}(x_0+x_4)-(x_2+x_6)\\(x_1+x_5)-(x_3+x_7)\end{bmatrix}\\ &= \begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}\begin{bmatrix}1 & 0\\0 & W^2\end{bmatrix}\begin{bmatrix}(x_0+x_4)-(x_2+x_6)\\(x_1+x_5)-(x_3+x_7)\end{bmatrix}\end{aligned} \tag{24}$$

Moreover, from Eq.(22),

$$\begin{aligned}\begin{bmatrix}X_1\\X_5\end{bmatrix} &= \begin{bmatrix}W^0 & W^1 & W^2 & W^3\\W^0 & W^5 & W^2 & W^7\end{bmatrix}\begin{bmatrix}x_0-x_4\\x_1-x_5\\x_2-x_6\\x_3-x_7\end{bmatrix}\\ &= \begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}\begin{bmatrix}1 & 0\\0 & W^1\end{bmatrix}\begin{bmatrix}(x_0-x_4)+(x_2-x_6)W^2\\(x_1-x_5)+(x_3-x_7)W^2\end{bmatrix}\end{aligned} \tag{25}$$

$$\begin{aligned}\begin{bmatrix}X_3\\X_7\end{bmatrix} &= \begin{bmatrix}W^0 & W^3 & W^6 & W^1\\W^0 & W^7 & W^6 & W^5\end{bmatrix}\begin{bmatrix}x_0-x_4\\x_1-x_5\\x_2-x_6\\x_3-x_7\end{bmatrix}\\ &= \begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}\begin{bmatrix}1 & 0\\0 & W^3\end{bmatrix}\begin{bmatrix}(x_0-x_4)-(x_2-x_6)W^2\\(x_1-x_5)-(x_3-x_7)W^2\end{bmatrix}\end{aligned} \tag{26}$$

The FFT calculation is composed of a regular procedure called a *butterfly*. Fig. 5 shows the complete calculation procedure of the eight-point FFT as a connected butterfly components.

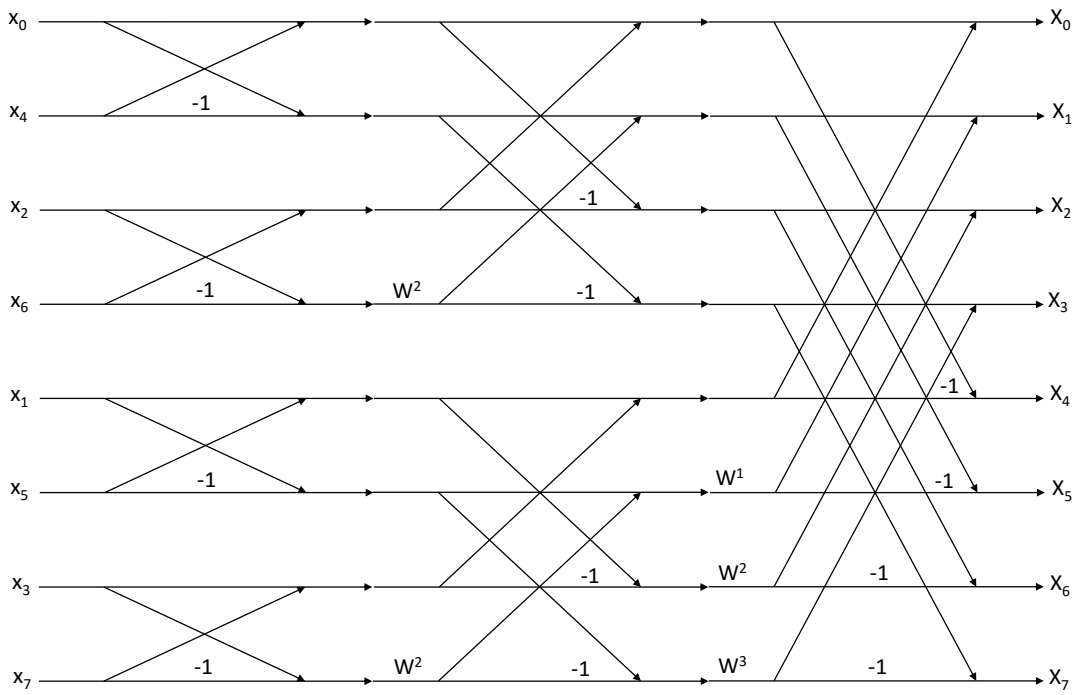

Figure 5: Calculation procedure of the eight-point FFT with butterfly components.

Fig. 5 presents two interesting properties in the FFT operation. The first property is *bit reverse*. The figure is drawn so that the FFT output appears in order of frequency $\{X_0, X_1, X_2, ..., X_7\}$. This is achieved by permuting the input elements in *bit-reverse* order as is shown in the figure, i.e. $\{x_0, x_4, x_2, x_6, x_1, x_5, x_3, x_7\}$. The reason of this name "bit reverse" becomes visible when you write down the indices of the input vector $\{0, 4, 2, 6, 1, 5, 3, 7\}$ in binary representation, which is $\{'000', '100', '010', '110', '001', '101', '011', '111'\}$. By reading each number in reverse direction, i.e. from LSB to MSB, the corresponding decimal number results in an array $\{0, 1, 2, 3, 4, 5, 6, 7\}$.

Another property is that one set of data is processed by only one butterfly to generate a new set of data. For example in Fig. 5, x_0 and x_4 are processed at the top-left butterfly to generate two values $(x_0 + x_4)$ and $(x_0 - x_4)$. And from this point on there is no reference to the original two values (x_0 and x_4) anymore, but only the new set of values are used in the following stage. The implication of this property in computing is that the memory space of the original input data can be reused or overwritten until the end of FFT calculation without allocating a separate memory space for intermediate or final results. This type of memory efficient FFT implementation is called *in-place* implementation, while use of separate memory space is called *out-of-place* implementation.

Each W^p in the matrix (20) corresponds to a complex multiplication operation. The matrix can be simplified by replacing W^0 by 1 and W^4 by -1. The simplified matrix still requires 32 complex multiplications. In the Fig. 5, each W^p coefficient attached to a link corresponds to a complex multiplication. Consequently, the total

number of multiplication operations is reduced to five from 32 in the original equation.

In this example the periodic property of matrix is found by separating the original matrix equation into even and odd indices of X_k, or by collecting every other index. Cooley and Tukey generalizes that the periodic property appears by collecting every p-th index, where p, a *twiddle factor*, is a factor of the current vector length N [9]. In the above example, the original vector length is $N = 2^3$, therefore every matrix decomposition step takes $p = 2$.

3.2 Pruning and transform decomposition

The ordinary FFT described in the previous section executes all the calculation paths in Fig. 5 to obtain N output values from N input values. However, there are cases where not all these calculations need to be done.

One example is when only a subset of input data holds non-zero values. This happens when the input data length is K, but the number of output points or the output resolution N needs to be larger (i.e. $K < N$). In this case the original input data is extended or padded by $(N - K)$ zeros so that the FFT of length N can process the input data.

Another example is when only a subset of the output contains values of interest. This is exactly the case of spectrum-sensing CR, since the radio frequency we are interested in is limited to a specific range of the radio communication bands, but not to low frequency (LF) or medium frequency (MF) bands.

FFT pruning was proposed to facilitate such cases [22] [24]. The number of operations in a single FFT task can be reduced by simply removing or pruning the unneeded calculation paths in Fig. 5 depending on the padded sequence of zeros in the input or the output points of interest. Unfortunately, the impact of FFT pruning is only modest. As an example, the time savings to execute 2^{10}-point FFT with 2^8 non-zero input data (that is, only the first quarter of the input data holds non-zero values) is less than ten percent with pruning [22].

The core algorithm of the *transform decomposition* is identical to the original FFT algorithm of Cooley and Tukey in the sense that it splits a whole FFT task into a set of smaller FFTs and recombine the partial results into the final result [30]. The specialty of this solution is that the transform decomposition maintains the smaller FFTs and recombination as separate processes, while the original FFT flattens out all the calculations as shown in Fig. 5. By separating the final recombination process from the internal smaller FFT tasks, only the output points of interest can be selectively calculated without computing unneeded output points. This method is faster than FFT pruning, and also more flexible in that the output points of interest do not have to be sequential as they do with FFT pruning.

3.3 Sparse FFT

Sparse FFT (sFFT) is the FFT algorithm that analyzes sparse spectrum efficiently. The algorithm was developed at the MIT [13], and its efficiency advantages were proven with a GHz-wide spectrum sensing without very high-speed ADC [14]. The

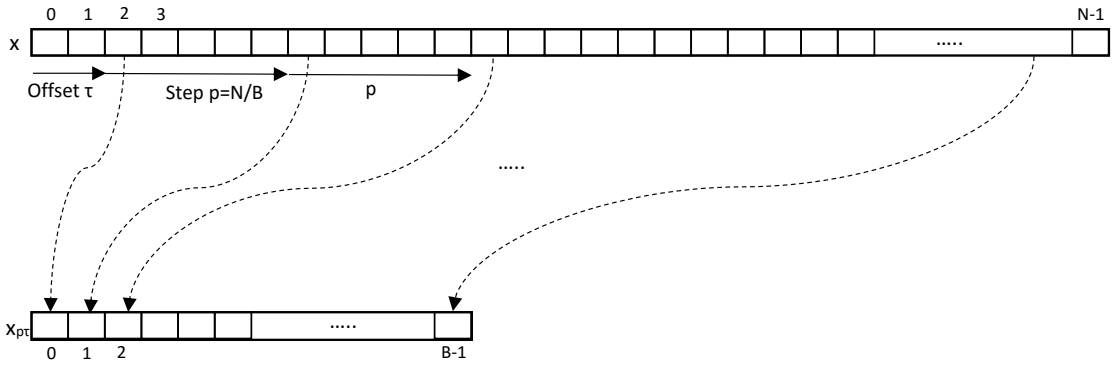

Figure 6: Time-decimation of signals including delay or offset.

sFFT combines basic properties of the FFT under the assumption of k-sparse spectrum.

The first step of the algorithm is the time-decimation of the samples. From the original sample data $\mathbf{x}$ of length N, a subsample data $\mathbf{x_p}$ is built by selecting every p-th element of $\mathbf{x}$ (Fig. 6). Here p is chosen from the factors of N ($p = N/B$). The length of the subsample $\mathbf{x_p}$ is B.

$$x_p[n] = x[np] \text{ , where } n = [0..B-1] \tag{27}$$

Then the FFT is executed on the subsample to obtain spectrum $\mathbf{X_p} = FFT(\mathbf{x_p})$. Notice that all the frequency components in the original signal $\mathbf{x}$ are still conserved in the subsample $\mathbf{x_p}$. But due to the length of subsample, $\mathbf{X_p}$ holds only B frequency elements, while the FFT on the original sample $\mathbf{X} = FFT(\mathbf{x})$ would generate N frequency elements. The original N frequency elements in $\mathbf{X}$ is divided into p sections of size B, and accumulated (or aliased) into B frequency elements of $\mathbf{X_p}$ (Fig. 7). When multiple non-zero frequency elements fall into one frequency element by aliasing, we call it *collision*. By choosing a proper size of $B(= N/p)$ to the sparsity k, the number of collisions is limited under the assumption of k-sparse spectrum $\mathbf{X}$.

To detect a collision, another set of subsamples and its FFT are needed, namely,

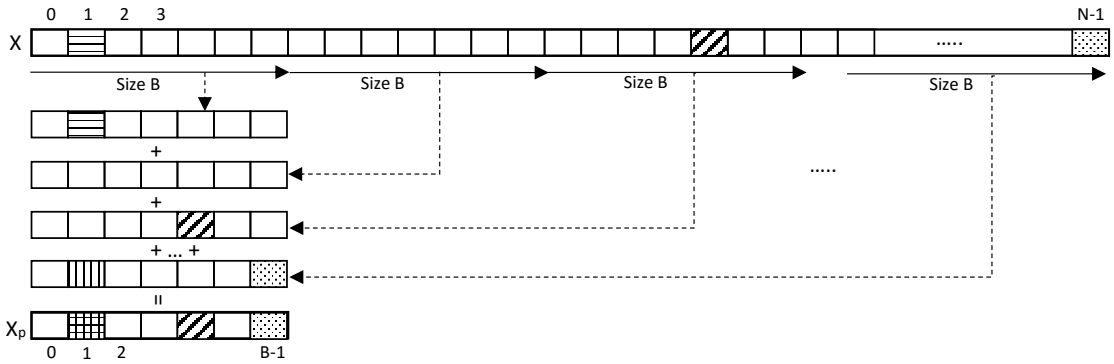

Figure 7: Aliasing of spectrum due to time-decimation. A collision occurred at X_{p1}.

$$x_{p\tau}[n] = x[np + \tau] \text{ , where } n = [0..B-1] \tag{28}$$

where τ is a constant time offset or delay from the sampling point of $\mathbf{x_p}$. Now we have another set of aliased spectrum $\mathbf{X_{p\tau}} = FFT(\mathbf{x_{p\tau}})$. In general, the spectrum of a time-delayed signal has a frequency-dependent phase shift against the original spectrum:

$$\Delta\phi = \frac{2\pi f\tau}{N} \tag{29}$$

When there is no collision at frequency f, the following relationship is observed between $X_p[f]$ and $X_{p\tau}[f]$.

$$X_{p\tau}[f] = X_p[f]\, e^{j\Delta\phi} = X_p[f]\, e^{j\frac{2\pi f\tau}{N}} \tag{30}$$

This means that the magnitude of these two values are identical. In this case the frequency, even the aliased one, can be determined or *estimated* from the phase shift as follows.

$$f = \frac{N\,\Delta\phi}{2\pi\tau} \tag{31}$$

Let us assume that there is a collision of two frequencies f_1 and f_2 that both fall into frequency f by aliasing. Then $X_p[f]$ is the sum of these two components:

$$X_p[f] = X_{f_1} + X_{f_2} \tag{32}$$

The phase of these two frequency components are independently shifted in $X_{p\tau}[f]$:

$$X_{p\tau}[f] = X_{f_1}\, e^{j\frac{2\pi f_1\tau}{N}} + X_{f_2}\, e^{j\frac{2\pi f_2\tau}{N}} \tag{33}$$

Consequently, we observe the collision as the difference in magnitude between $X_p[f]$ and $X_{p\tau}[f]$.

Up to this point of the sFFT process, we have been able to identify all the frequency components without collision in $\mathbf{X_p}$, and estimate the correct frequencies back in $\mathbf{X}$. To resolve the collisions in $\mathbf{X_p}$, sFFT executes the same analysis with another subsampling step q that is also one of the factors of N. With the new subsample $\mathbf{x_q}$, we obtain a new aliased spectrum $\mathbf{X_q} = FFT(\mathbf{x_q})$. Since we have identified some spectrum through the first analysis, those known spectrum can be subtracted from $\mathbf{X_q}$. This operation resolves collisions between the known and an unknown spectrum, and the remaining stand-alone spectrum can be solved again with the above procedure.

The most problematic case is that the pair of frequencies that collided in $\mathbf{X_p}$ collide again in $\mathbf{X_q}$. This situation can be avoided by choosing p and q as co-prime [14]. Further subsampling analysis is required in case of collisions among unknown frequency elements. The number of operations for the sFFT to calculate k-sparse spectrum from N number of data is in order of $O(k \log N)$ [13].

3.4 Efficient FFT for mobile devices

This section summarizes the features of selected FFT software. The foremost criterion of program selection is open-source. Other criteria are the number of references in academic papers and reputation in evaluations. We also chose libraries that were developed under a clear concept, such as pedagogical code and those optimized for mobile applications.

The FFT programs are classified into two categories. The baseline FFTs implement the traditional Cooley and Tukey FFT algorithm and are applicable to general input data. The sparse FFT is only applicable to the input data that is known to result in sparse spectrum.

Baseline FFT libraries

- **The basic FFT implementation of Princeton University** [35] is a pedagogical code focusing on the hierarchical vector decomposition nature of the FFT. It separates the input vector into an even-index vector and an odd-index vector, runs independent FFT on each vector, and then combine the two FFT results into one vector. The input vector is decomposed recursively until each vector contains one element. This is an out-of-place implementation.

- **In-place FFT implementation of Princeton University** [34] includes a few techniques for efficient computing, such as bit-reverse reordering of input data that does not require recursive operations, and in-place memory usage.

- **The FFT program of Columbia University** [33] is also an in-place implementation with bit-reverse reordering. It pre-calculates an array of rotation vectors (W^p) from that a rotation vector is looked up during FFT calculations. The above two implementations of Princeton University calculate rotation vectors on the fly.

- **The FFTW (Fastest Fourier Transform in the West)** from MIT [12] is one of the highly referenced FFT libraries for its speed and robustness. The program internally generates optimized micro FFT codes for various twiddle factors. It supports SIMD instruction sets such as SSE, SSE2, Altivec, AVX, and NEON.

- **Libav** is a community-driven effort that provides portable libraries for multimedia applications [21]. The FFT program supports NEON instruction set, and is distributed as a part of libavcodec library.

- **Cricket FFT** is designed for Android and iOS applications. It is optimized for ARM devices and supports NEON instructions [31].

- **The FFTS (Fastest Fourier Transform in the South)** is also designed for Android and iOS applications, and supports NEON. It dynamically generates code at runtime as FFTW does [5].

- **Superpowered** publishes a multi-platform version of the well-reputed FFT library *vDSP*, which is made by Apple and available only for iOS, with NEON support [18].
- **Kiss FFT** is a compact implementation with the concept of "Keep It Simple and Stupid" [6] without NEON support.

Sparse FFT

- Source code of **the Sparse FFT (sFFT)** is published by MIT [26] as a benchmark program between the sFFT and the FFTW. The FFTW is also utilized within the sFFT to execute FFT on the subsamples. Since its algorithm assumes the FFT result to be a sparse spectrum, the program does not accept a general input signal. The special operational mode of the sFFT is detailed in Section 4.2.4

4 Benchmarking FFT Libraries on Android

This chapter describes the benchmarking software we built to evaluate FFT libraries on Android. After describing the specifications of the Android device used, we detail the developed software starting from its architecture and internal input file generation. Then, we list all the FFT variations supported by the benchmark software and the integration method we used. We continue with the special treatment of the sparse FFT evaluation, and conclude the chapter with the logging function of the benchmarking program.

4.1 The test device

Table 1 summarizes the specification of the test device for the evaluation work, Samsung Galaxy S5 SM-G900F 16GB.

Table 1: The specification of Android test device

Item	Description
Chipset	Qualcomm MSM8974AC Snapdragon 801
CPU	Quad-core 2.5 GHz Krait 400 (ARMv7 compatible architecture)
GPU	Adreno 330
RAM	2 GB
Storage	16 GB
OS	Android 5.0 (Lollipop)

ARMv7 or later processor architecture supports NEON instruction sets.

4.2 Benchmarking software

This section details the benchmarking software from the view points of architecture, input and output control, FFT library integration, and their performance measurement.

4.2.1 Architecture

Fig. 8 shows the software architecture and the data flow of our benchmarking program. The overall structure (including the white boxes and arrows in the figure) was developed in the Android programing environment with Java and XML resource files. The FFT libraries in C/C++ were compiled to ARM executables (the shadowed boxes) and were called from the main program via the Java Native Interface (JNI) framework [27].

The user provides three parameters to the program: the FFT data length, the FFT library to run, and the number of FFT iterations. The program first generates

the FFT input data. Then it calls the selected FFT library and iterates it for the specified number of times. The program measures the execution time of the whole FFT iterations, and then calculates the average execution time for one FFT cycle. It finally records all the input parameters, the first input and the last output of the FFT iterations, and execution time information into a log file.

The developers of the Sparse FFT (sFFT) publish a dedicated performamce comparison program between sFFT and FFTW [26]. This program was compiled and integrated in the main program for sFFT evaluation.

The following sections describe the major functional blocks – Input vector generation, FFT iterations, sFFT vs FFTW comparison, and Logging – in detail.

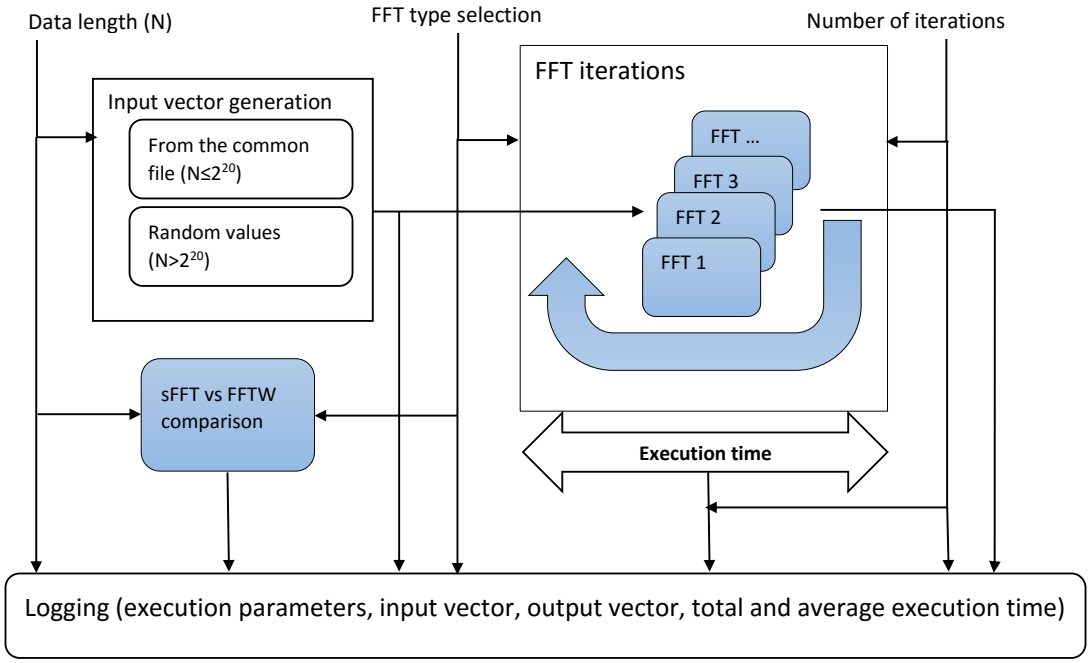

Figure 8: The architecture of the FFT benchmarking program.

4.2.2 Input vector generation

The user provides the input data length as an integer exponent of two (2), such as 10 for a data length of $N = 2^{10}$. Since some FFT libraries require input data lengths to be powers of two, this is a practical way of defining data lengths. The program generates an array of complex values of the specified length as the FFT input vector. There are two operation modes in generating input vector depending on the specified data length. The block reads the shared data file stored on the device and builds an input vector up to data length of $N = 2^{20}$, which is the longest data length the file can provide. This ensures that all the FFT libraries start their operations on identical input vectors. For longer data lengths the block generates input vectors

with random values. The input vector generation function is implemented as Java code.

4.2.3 FFT iterations

The FFT iterations block in the Fig. 8 executes the specified FFT library for the specified number of iterations. At the same time it measures execution time of the whole iterations. When FFT iterations have completed, the program calculates the average FFT execution time per iteration.

Table 2 lists all the FFT libraries built in the benchmarking program. There are two aspects to emphasize here. First, most of the libraries are distributed as C/C++ code. Among the list the libraries from Columbia University and Princeton University are originally implemented in Java. For these libraries we translated the algorithms described in Java into C, compiled them, and integrated them into the benchmarking program in addition to the original Java implementations for the purpose of performance comparison.

Second, FFTW and Libav FFT support the NEON instruction set as a compile-time option. For these libraries two versions of executable are built with and without activating NEON instruction sets for comparison purposes. For the other libraries that do not provide any option with regard to NEON, only one executable was built for each library.

Table 2: List of FFT libraries built into the benchmarking program.

Library name	Java	C/C++	
		NEON	
		off	on
Columbia Univ. FFT [33]	✓	✓	
Princeton Univ. Basic FFT [35]	✓	✓	
Princeton Univ. In-place FFT [34]	✓	✓	
FFTW [12]		✓	✓
Libav FFT [21]		✓	✓
Cricket FFT [31]			✓
FFTS [5]			✓
Superpowered FFT [18]			✓
Kiss FFT [6]		✓	
sFFT [13]		✓	

In this project we developed an iteration control part for each FFT library as a wrapper C code. On top of the loop control function, the wrapper handles data feedback from FFT output to input with scaling. This scaling maintains the FFT output signal power to be identical to the input signal power[2]. Without the scaling the amplitude of signal increases at each FFT iteration, and eventually overflows the data range. Handling data feedback within the binary code reduces the switching overhead

between binary and Java programs. The wrapped FFT libraries are compiled into ARM executables and packed into the Android application. We employed the Native Development Kit (NDK), which Google provides to developers, to cross-compile architecture-specific binary codes and packing them into the Android application. The shadowed blocks in the FFT iterations box in the Fig. 8 correspond to the parts developed with the NDK.

We also developed an interface Java code for each native ARM executable according to the JNI specification. The main Android application passes FFT parameters and input/output vectors through this interface.

4.2.4 Comparison between sparse FFT and FFTW

The evaluation of the sFFT is not exactly the same as the other FFT libraries because of its peculiar algorithm and operation mode. In fact, the sFFT algorithm is targeted to a signal that contains sparse spectrum. Because of this property, the sFFT program requires a sparsity parameter that is the maximal number of significant spectrum values in the result. When the program detects that the number of significant spectrum values exceeds the specified sparsity, it stops with an error message. This means that a tailored input vector is required with a knowledge of its spectrum so that a target sparsity can be at least specified. Under this restriction, the shared input vector is not applicable to sFFT since it does not necessarily fulfill the sparsity constraints.

The applied program is not merely the sFFT library but actually a program for performance comparison between sFFT and FFTW. According to the given sparsity parameter, such program first generates a random sparse spectrum, which is the expected spectrum FFT programs should obtain. Then the program executes the inverse FFT (IFFT) to generate the corresponding input vector. After that, the program executes the sFFT and FFTW algorithms with this tailor-made input vector. The execution time is measured for one FFT run of each algorithm without iterations.

This program was also integrated as a compiled binary with the NDK, and so it is depicted as a shadowed box in the Fig. 8 as the other compiled binaries.

4.2.5 Logging

When the program executes an FFT library, it creates a log file with the file-name format of `<library_name>-<data_length>-<number_of_iterations>.log`, such as `FFTWf_NEON-2^10-10^3.log`.

In addition to the information used to compose the log file name, the total time for the FFT iterations and the average time per iteration are recorded in the log file.

[2] The FFT algorithm in Section 3.1.3 results in the output vector ($\mathbf{X}$) to have signal power of the input vector ($\mathbf{x}$) scaled by N. Usually an FFT library does not normalize the output signal power to maintain the input signal power. The reason behind this is that the interest of FFT applications is usually the relative *form* of the FFT result but not its absolute value. To FFT applications such as spectrum analysis and convolution, the *location* of the peak value is the main issue, and its absolute value is less important.

When the data length is in range $N \leq 2^{20}$, the initial input vector and the final FFT output vector are also recorded in the file.

To prevent the logging process from affecting FFT execution time, logging functions are completely separated from time measurement. First, all the input parameters are recorded at the point when the input vector is created. Then, time measurement is started just before beginning FFT iterations. At the completion of the FFT iterations, time measurement is stopped immediately. Finally, the average FFT execution time is calculated and logged with the other parameters.

5 Evaluation Results

This chapter presents the results obtained by our benchmarking software. In the first section, we observe the impact of implementation options with the same algorithm, namely, from the aspects of implementation languages and support for SIMD instruction sets. Moreover, we compare the performance of ordinary FFT libraries and then carry out a special comparison between FFTW and sFFT. We also estimate the impact of the transform decomposition for sensing spectrum of an LTE band. Finally, we summarize the obtained results.

5.1 Impact of implementation languages and instruction sets

5.1.1 Comparison between C and Java

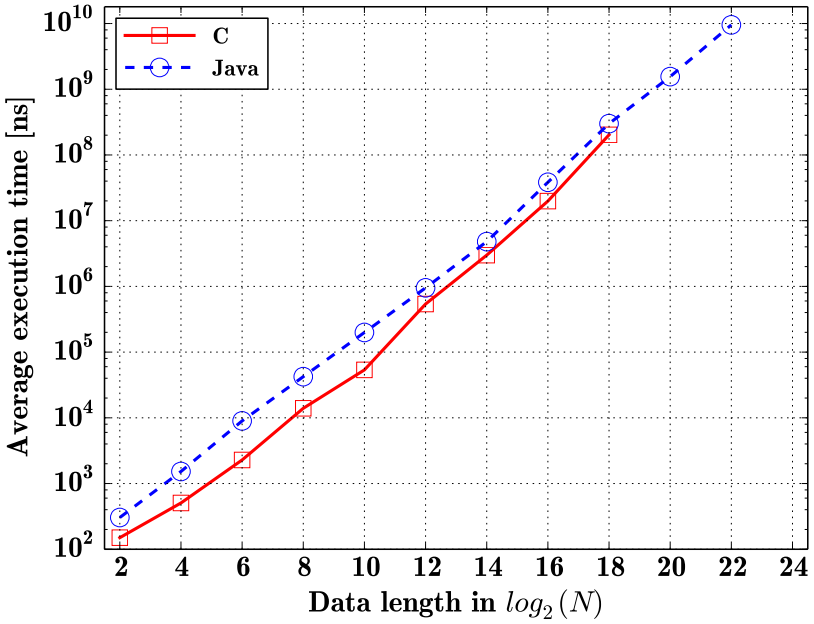

Figure 9: Performance comparison between C and Java implementations of the FFT algorithm from Columbia University. The C implementation is two to four times faster than the Java implementation.

Fig. 9 and Fig. 10 show the average execution times of FFT algorithms implemented in C and Java languages. The C implementation is faster than the original Java implementation in all the three cases tested in this work.

Specifically, Fig. 9 shows the performance of the FFT algorithm from Columbia University. The C version could complete the FFT calculation for data length up to $N = 2^{18}$. Up to this point the C implementation is two to four times faster than the Java implementation. This algorithm creates a pre-calculated rotation-vector array whose size is proportional to the input data length. The memory space occupied by

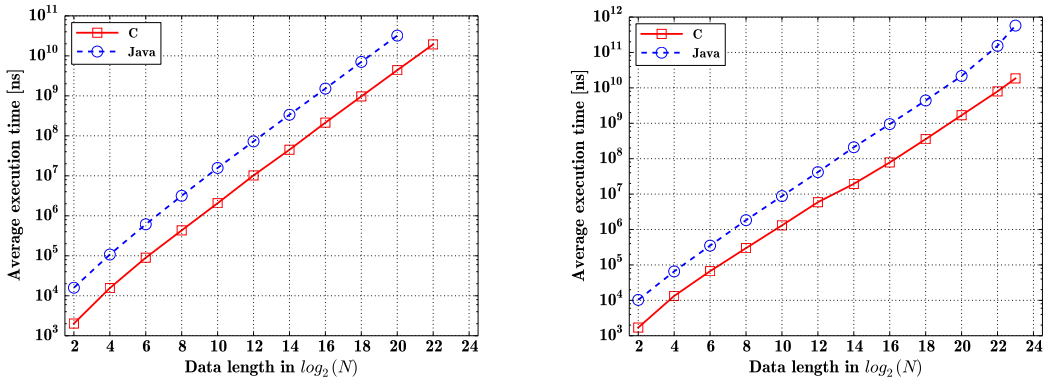

Figure 10: Performance comparison between C and Java implementations of (a) the basic FFT algorithm and (b) the in-place FFT algorithm from Princeton University.

this special array might have caused the reduced performance improvement for data length of $N \geq 2^{12}$, and eventually led to the incomplete tasks.

Fig. 10 presents the performance of the FFT algorithms from Princeton University. Compared to the original Java library, the corresponding C implementation is seven times faster for the basic FFT algorithm (Fig. 10a), and ten to 30 times faster for the in-place FFT algorithm (Fig. 10b).

5.1.2 NEON instruction set

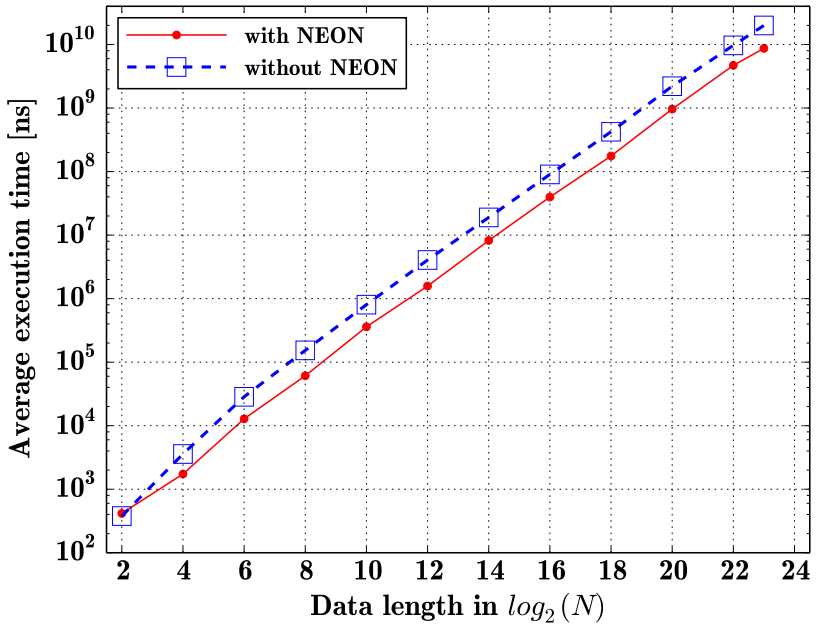

Figure 11: FFTW performance comparison with and without NEON instruction set.

Fig. 11 shows the average execution time of the FFTW library written in C and also supporting the NEON instruction set. The results show that FFTW with the NEON instruction sets is two times faster than the implementation of the same algorithm without the special optimization. Since the NEON instruction set executes one operation on two data sets in parallel, the total number of clock cycles is halved comparing to the corresponding single-instruction single-data (SISD) program. Therefore, this two-fold performance improvement is as expected.

5.2 Comparison among ordinary FFT libraries

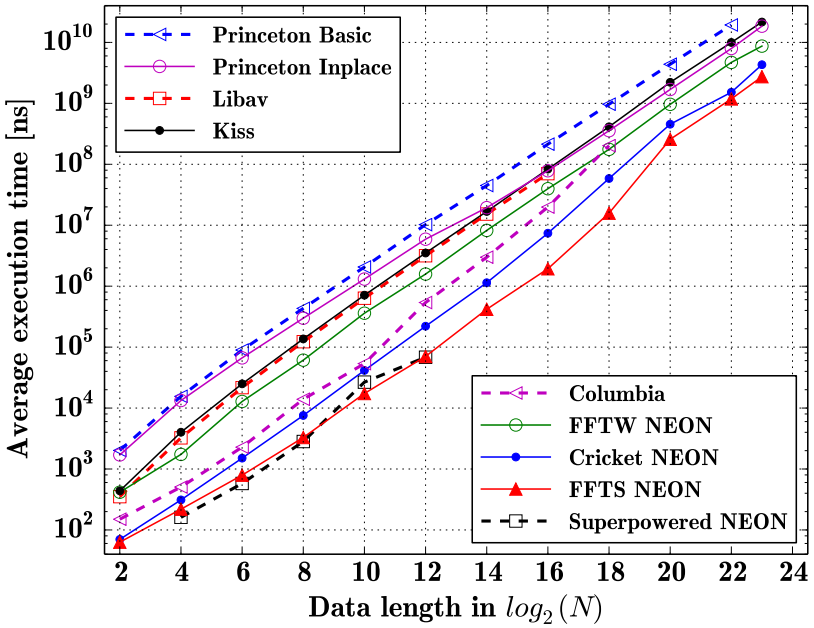

Figure 12: Performance comparison of ordinary FFT libraries implemented in C.

Fig. 12 shows the average execution time of all the ordinary (i.e., non-sparse) FFT libraries implemented in C language. For the libraries that support NEON instruction sets, only the results of NEON-enabled versions are included. The most important observation from the result is that all the performances of ordinary FFTs show a linear dependency on the data length.

We now focus on the performance of the libraries that we translated from Java to C: the three libraries from Princeton University and Columbia University.

The Basic FFT library from Princeton University implements the recursive matrix decomposition process described in Section 3.1.3. Since this is a pedagogical library, the clarity of the algorithm takes more emphasis on its efficiency.

The In-place FFT library from Princeton University introduces bit-reversal permutation on the input vector to utilize memory space efficiently. With the permuted input vector the program can overwrite the memory array of the input vector at each calculation step, and finally the memory array holds the FFT result.

This strategy is called “in-place operation” since it reuses the memory array for the input vector without allocating another memory array for the output vector.

The FFT library from Columbia University also implements the in-place operation. In addition to that, it reduces the number of complex operations by calculating and storing the rotation vector arrays (W^p) before the actual FFT calculations. While it refers to the rotation vector array during the FFT calculations, the libraries from Princeton University calculate the rotation vector on the fly.

Among the ordinary FFT libraries, “FFTS with NEON” is the fastest for almost the whole range of considered data lengths.

5.3 Comparison between FFTW and sFFT

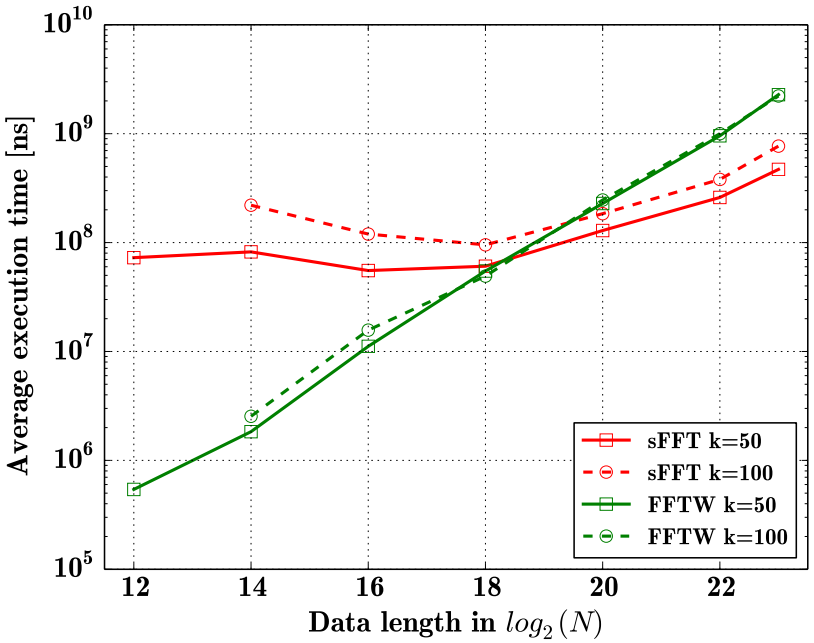

Figure 13: Performance comparison between sFFT and FFTW with sparsity parameter $k = 50, 100$. Note that the data-length range is different from the previous figures.

Fig. 13 shows the execution times obtained with sFFT and FFTW (without NEON instruction sets) for two cases of sparsity, i.e., $k = [50, 100]$. The graph shows differences of properties in the two algorithms.

The performance of sFFT does not show the linear dependency to the input data length as the ordinary FFT libraries do. On the other hand, it is dependent on the sparsity parameter k: it performs faster for a lower k, i.e. when the FFT result consists of fewer significant spectrum.

The performance of FFTW is independent on the sparsity of the resulting spectrum. This also applies to the other ordinary FFT libraries because they do not require the sparsity parameter for execution.

Because of difference in their impact factor on performance, performance of sFFT exceeds that of FFTW for long input data where $N \geq 2^{20}$.

5.4 Impact of the transform decomposition

Even though the transform decomposition could be effective to spectrum-sensing CR, no open-source library was found for this evaluation work. Instead of measuring actual execution time on a device, we try to estimate its performance over the typical FFT program.

To give an example of the advantages in using the transform decomposition, let us consider sensing an operational LTE band in Europe. The most challenging condition for the transform decomposition happens when the frequency range of interest is wide relative to the frequency band. According to the 3GPP standard [3], this happens at Band 3 in that the uplink and downlink bands are 1,710 MHz-1,785 MHz and 1,805 MHz-1,880 MHz, respectively. This frequency range of interest occupies $(1{,}880 - 1{,}710)/1{,}880 = 9.04\%$ of the frequency up to 1,880 MHz. In a 512-point DFT, this range corresponds to $512 \cdot 9.04\% = 47$ points. Referring to Fig. 12 *Performance of different methods for computing a subset of output points of a length 512 DFT* in [30], the transform decomposition reduces the total number of operations by about 16% to calculate 47 points out of 512 comparing to the typical split-radix FFT.

5.5 Summary of results

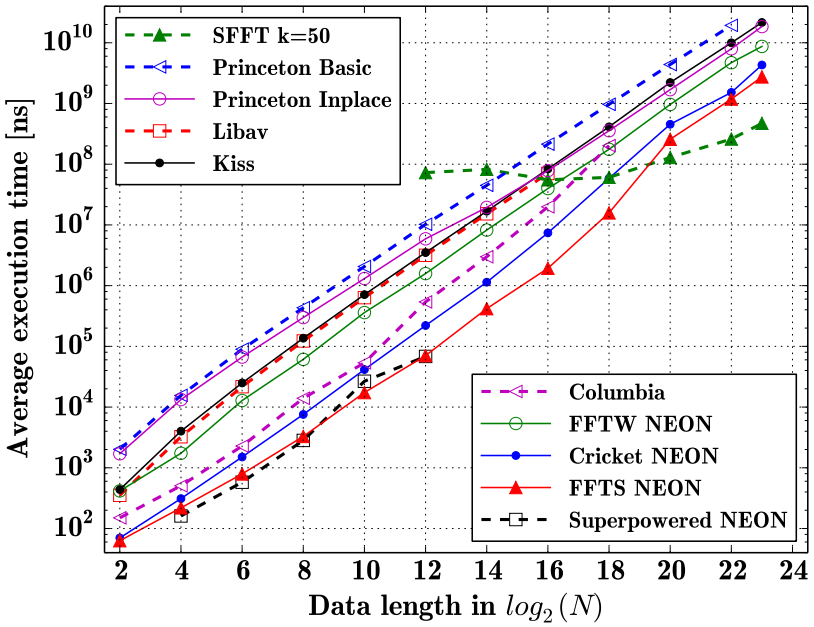

Figure 14: Performance comparison of all FFT libraries.

Fig. 14 shows all the obtained results in a single plot. The operation mode of

sFFT is clearly different from the other FFT libraries. Besides, its performance is significantly better than that of other libraries when the data length $N \geq 2^{20}$ under the condition of sparsity $k = 50$.

6 Conclusion

In this thesis we first discussed cognitive radio (CR) with software-defined radio (SDR) technology as a paradigm to achieve effective utilization of radio frequency spectrum. Effective spectrum sensing techniques are needed to realize CR. Since the Fast Fourier Transform (FFT) plays the major role in spectrum analysis, then we focused the theoretical view from Fourier Transform (FT) through FFT with standard techniques to improve its performance such as FFT pruning and transform decomposition. We also looked at newer techniques such as compressive sensing (CS) and sparse FFT that enable faster FFT operation in practical settings.

Considering that CR will be most beneficial to mobile devices, we benchmarked open-source FFT programs on Android, which is the major mobile platform at present. We integrated a number of FFT executables into an Android benchmark program and compared their average FFT execution time.

From the benchmark results of ordinary FFT programs, we observed the following facts.

- As the type of FFT implementation, the binary executables are two to 30 times faster than the corresponding Java implementations.
- The data length parameter is generally leveraged to improve the performance, such as through the bit-reversal permutation of the input vector and the pre-calculated rotation vector array.
- SIMD (single-instruction multiple-data) instruction sets, such as NEON, increase FFT performance because of the data-level parallelism. This typically results in halving the execution time.
- "FFTS with NEON" performs the best among the ordinary FFT programs.

We also discussed more sophisticated approaches to sparse spectrum sensing, such as CS and the Sparse FFT. The benchmark program actually proved that sFFT executes faster than the considered FFTW library with long input data ($N \geq 2^{20}$) with sparse spectrum cases. However, the constraint in applying sFFT is that it requires the spectrum sparsity as an execution parameter. Because of this requirement, sFFT can not be smoothly applied to general data instead of ordinary FFT programs.

To overcome this difficulty, further study could focus on two aspects. One is a survey of practical CR contexts, such as actual sparsity and the target frequency range, to unravel whether a general parameter for a practical context exists or not. The other one is an algorithmic approach. If there is a way to estimate the spectrum sparsity starting from a rough value to a finer value without using much execution time, the algorithm will become autonomous and robust for general usage.

References

[1] About RTL-SDR. "http://www.rtl-sdr.com/about-rtl-sdr/". Accessed: 2016-07-01.

[2] NEON. "http://www.arm.com/products/processors/technologies/neon.php". Accessed: 2016-02-28.

[3] *ETSI TS 136 101 V12.9.0*, chapter Operating bands and channel arrangement. 3GPP, Oct. 2015.

[4] Andreas Achtzehn, Janne Riihihjärvi, Irving Antonio Barriía Castillo, Marina Petrova, and Petri Mähönen. Crowdrem: Harnessing the power of the mobile crowd for flexible wireless network monitoring. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, pages 63–68. ACM, 2015.

[5] Anthony M Blake, Ian H Witten, and Michael J Cree. The fastest fourier transform in the south. *Signal Processing, IEEE Transactions on*, 61(19):4707–4716, 2013.

[6] Mark Borgerding. Kiss FFT. "https://sourceforge.net/projects/kissff/". Accessed: 2016-02-20.

[7] Emmanuel J Candè and Michael B Wakin. An introduction to compressive sampling. *Signal Processing Magazine, IEEE*, 25(2):21–30, 2008.

[8] A.B. Carlson. *Communication Systems: An Introduction to Signals and Noise in Electrical Communication*. McGraw-Hill Series in Electrical and Computer Engineering. McGraw-Hill, 1986.

[9] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[10] FCC. Notice of proposed rule making and order. "https://apps.fcc.gov/edocs_public/attachmatch/FCC-03-322A1.pdf", December 2003.

[11] Joseph Fourier. *Theorie analytique de la chaleur, par M. Fourier*. Chez Firmin Didot, père et fils, 1822.

[12] Matteo Frigo and Steven G Johnson. Fftw: An adaptive software architecture for the fft. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381–1384. IEEE, 1998.

[13] Haitham Hassanieh, Piotr Indyk, Dina Katabi, and Eric Price. Nearly optimal sparse fourier transform. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 563–578. ACM, 2012.

[14] Haitham Hassanieh, Lixin Shi, Omid Abari, Ezzeldine Hamed, and Dina Katabi. Bigband: Ghz-wide sensing and decoding on commodity radios. 2013.

[15] Kazunori Hayashi, Masaaki Nagahara, and Toshiyuki Tanaka. A user's guide to compressed sensing for communications systems. *IEICE transactions on communications*, 96(3):685–712, 2013.

[16] Monson H Hayes. *Schaum's outline of digital signal processing*. McGraw-Hill, Inc., 1998.

[17] M Heidemann, D Johnson, and CS Burrus. Gauss and the history of the fft. *IEEE Mag*, 1, 1984.

[18] Supowerpowered Inc. iOS and Android FFT & iOS and Android Polar FFT Library. "http://superpowered.com/fft-and-polar-fft". Accessed: 2016-02-29.

[19] Texas Instruments. ADS54J54 Quad Channel 14-Bit 500 MSPS ADC. "http://www.ti.com/product/ADS54J54/datasheet". Accessed: 2016-09-14.

[20] R. Kuc. *Introduction to Digital Signal Processing*. McGraw-Hill series in electrical and computer engineering. McGraw-Hill, 1988.

[21] Libav. FFT functions. "https://libav.org/documentation/doxygen/master/group__lavc__fft.html". Accessed: 2016-02-22.

[22] John D Markel. Fft pruning. *Audio and Electroacoustics, IEEE Transactions on*, 19(4):305–311, 1971.

[23] Joseph Mitola. Software radios: Survey, critical evaluation and future directions. *IEEE Aerospace and Electronic Systems Magazine*, 8(4):25–36, 1993.

[24] Keinosuke Nagai. Pruning the decimation-in-time fft algorithm with frequency shift. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 34(4):1008–1010, 1986.

[25] Ana Nika, Zengbin Zhang, Xia Zhou, Ben Y Zhao, and Haitao Zheng. Towards commoditized real-time spectrum monitoring. In *Proceedings of the 1st ACM Workshop on Hot Topics in Wireless*, pages 25–30. ACM, 2014.

[26] Massachusetts Institute of Technology. SFFT sparse fast fourier transform. "http://groups.csail.mit.edu/netmit/sFFT". Accessed: 2015-10-12.

[27] Oracle. Java Native Interface Specificatio. "http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html". Accessed: 2015-09-29.

[28] Yongtae Park, Jiung Yu, JeongGil Ko, and Hyogon Kim. Software radio on smartphones: feasible? In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, page 17. ACM, 2014.

[29] Jinghao Shi, Zhangyu Guan, Chunming Qiao, Tommaso Melodia, Dimitrios Koutsonikolas, and Geoffrey Challen. Crowdsourcing access network spectrum allocation using smartphones. In *Proceedings of the 13th ACM workshop on hot topics in networks*, page 17. ACM, 2014.

[30] Henrik V Sorensen and C Sidney Burrus. Efficient computation of the dft with only a subset of input or output points. *Signal Processing, IEEE Transactions on*, 41(3):1184–1200, 1993.

[31] Cricket Technology. Cricket FFT. "http://www.crickettechnology.com/ckfft". Accessed: 2016-02-20.

[32] Linear Technology. LTC2153-12 - 12-Bit 310Msps ADC. "http://www.linear.com/product/LTC2153-12". Accessed: 2016-09-14.

[33] Columbia University. FFT.java. "http://www.ee.columbia.edu/~ronw/code/MEAPsoft/doc/html/FFT_8java-source.html", February 2007. Accessed: 2015-06-10.

[34] Princeton University. InplaceFFT.java. "http://introcs.cs.princeton.edu/java/97data/InplaceFFT.java". Accessed: 2015-06-30.

[35] Princeton University. FFT.java. "http://introcs.cs.princeton.edu/java/97data/FFT.java.html", 2011. Accessed: 2015-06-10.

[36] Tan Zhang, Ashish Patro, Ning Leng, and Suman Banerjee. A wireless spectrum analyzer in your pocket. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, pages 69–74. ACM, 2015.