# High Performance Sparse Fast Fourier Transform

Master Thesis

Jörn Schumacher

Supervisor:
Prof. Markus Püschel

May 6, 2013

ETH Zurich
Department of Computer Science

**Abstract**

The Sparse Fast Fourier Transform is a recent algorithm developed by Hassanieh et al. at MIT for Discrete Fourier Transforms on signals with a sparse frequency domain. A reference implementation of the algorithm exists and proves that the Sparse Fast Fourier Transform can be faster than modern FFT libraries. However, the reference implementation does not take advantage of modern hardware features like vector instruction sets or multithreading.

In this Master Thesis the reference implementation's performance will be analyzed and evaluated. Several optimizations are proposed and implemented in a high-performance Sparse Fast Fourier Transform library. The optimized code is evaluated for performance and compared to the reference implementation as well as the FFTW library.

The main result is that, depending on the input parameters, the optimized Sparse Fast Fourier Transform library is two to five times faster than the reference implementation.

# Contents

**1 Introduction** **7**
1.1 Motivation . . . 7
1.2 Thesis Objective . . . 9
1.3 Related Work . . . 9
1.4 Contributions and Results . . . 10
1.5 Outline . . . 11

**2 The Sparse Fast Fourier Transform** **13**
2.1 Notation . . . 14
2.2 Basic Principles . . . 14
2.2.1 Random Spectrum Permutation . . . 14
2.2.2 Window Functions . . . 15
2.2.3 Fast Subsampling and DFT . . . 17
2.3 SFFT Version 1 . . . 17
2.4 SFFT Version 2 . . . 23
2.5 SFFT Version 3 . . . 23
2.6 SFFT Version 4 . . . 25

**3 Performance Analysis** **29**
3.1 Asymptotic Runtime . . . 29
3.1.1 Sparse Fast Fourier Transform Version 1 . . . 29
3.1.2 Sparse Fast Fourier Transform Version 3 . . . 32
3.2 Benchmarks . . . 33
3.3 Profiling . . . 36
3.4 Roofline Analysis . . . 37

**4 Performance Optimizations** **41**
4.1 Instruction Reduction . . . 41
4.1.1 FFTW . . . 41
4.1.2 Inlining and explicit complex arithmetic . . . 43
4.1.3 Fixed loop configurations . . . 43
4.1.4 Optimizing Individual Instructions . . . 45

4.2 Cache Usage Optimizations . . . . . . . . . . . . . . . . . . 46
4.2.1 Chunking . . . . . . . . . . . . . . . . . . . . . . . . 46
4.2.2 Data Layout . . . . . . . . . . . . . . . . . . . . . . 48
4.2.3 Stride-2 FFTs . . . . . . . . . . . . . . . . . . . . . 49
4.3 Vectorization . . . . . . . . . . . . . . . . . . . . . . . . . 50
4.3.1 SSE Support and Memory Alignment . . . . . . . . . 50
4.3.2 SSE Implementations of Compute Intensive Functions 51
4.3.3 More Vectorization . . . . . . . . . . . . . . . . . . . 52
4.4 Multithreading . . . . . . . . . . . . . . . . . . . . . . . . 55
4.4.1 Parallelizing Filters using OpenMP . . . . . . . . . . 55
4.4.2 Coarse Multithreading . . . . . . . . . . . . . . . . . 55
4.5 Miscellaneous Optimizations . . . . . . . . . . . . . . . . . 56
4.5.1 Compilers and Compiler Options . . . . . . . . . . . 56
4.5.2 High-Performance Trigonometric Functions and Intel IPP . . . . . . . . . . . . . . . . . . . . . . . . . . 56
4.5.3 Result Storage Data structure . . . . . . . . . . . . . 57

**5 Results 59**
5.1 Runtime Benchmarks . . . . . . . . . . . . . . . . . . . . . 59
5.2 Performance . . . . . . . . . . . . . . . . . . . . . . . . . . 59
5.3 Cold-Cache Benchmarks . . . . . . . . . . . . . . . . . . . 62
5.4 Profiling . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
5.5 Roofline Analysis . . . . . . . . . . . . . . . . . . . . . . . 66
5.6 Multithreading . . . . . . . . . . . . . . . . . . . . . . . . 66

**6 Conclusions 69**
6.1 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
6.2 Outlook . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
6.3 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . 70

**Acknowledgments 71**

**A Manual 73**
A.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . 73
A.1.1 When Should I use the SFFT library? . . . . . . . . . 73
A.1.2 Target Platform . . . . . . . . . . . . . . . . . . . . . 73
A.1.3 Limitations and Known Bugs . . . . . . . . . . . . . 73
A.1.4 Credits . . . . . . . . . . . . . . . . . . . . . . . . . . 73
A.2 Installation . . . . . . . . . . . . . . . . . . . . . . . . . . 74
A.2.1 Prerequisites . . . . . . . . . . . . . . . . . . . . . . . 74
A.2.2 Compiling From Source and Installation . . . . . . . 74
A.2.3 Linking against the SFFT Library . . . . . . . . . . . 75

A.3 Usage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
A.3.1 Computing Sparse DFTs . . . . . . . . . . . . . . . . . . 75
A.3.2 SFFT Versions . . . . . . . . . . . . . . . . . . . . . . . 77
A.4 Development . . . . . . . . . . . . . . . . . . . . . . . . . . 78
A.4.1 Development and Benchmark Tools . . . . . . . . . . . 78
A.4.2 An Overview of the Sourcecode . . . . . . . . . . . . . 79

# Chapter 1

# Introduction

## 1.1 Motivation

The Fourier Transform is an important and well-known mathematical method with a variety of applications in many scientific disciplines. In its discrete (DFT) form it can be formulated as

$$\hat{x} = \mathrm{DFT}_n \cdot x, \tag{1.1}$$

where $x$ and $\hat{x}$ are $n$-dimensional complex input and output vectors and $\mathrm{DFT}_n = (\omega_n^{kl})_{0 \leq k,l < n}$ for an $n$-th primitive root of unity $\omega_n = e^{-2\pi i/n}$. There are many applications for the DFT; for example [RKH10] mentions applications in signal processing, image compression, noise filtering or numerical solution of PDEs, amongst others.

A straightforward evaluation of equation 1.1 involves $\mathcal{O}(n^2)$ operations. Since the DFT is such a useful tool for many applications, there is a need for fast algorithms. The most well-known fast algorithm for DFTs is the *Fast Fourier Transform (FFT)*, originally described by Cooley and Tukey in [CT65][1]. The asymptotic runtime of this FFT is $\mathcal{O}(n \log n)$ and it is therefore much faster than the straightforward algorithm.

Reducing the runtime cost of the transform from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$, the FFT was a revolutionary algorithm. By the *Computing in Science and Engineering* Journal, it was picked as one of the top 10 algorithms of the 20th century in [DS00], describing it as the "most ubiquitous algorithm in use today to analyze and manipulate digital or discrete data".

Though several improvements to the Cooley-Tukey-FFT were proposed, like in-place algorithms or split-radix algorithms (refer to [RKH10] for further information), no algorithm for general DFTs is currently known with

[1]Though previously discovered by Gauss, the method did not get much attention until Cooley's and Tukey's paper

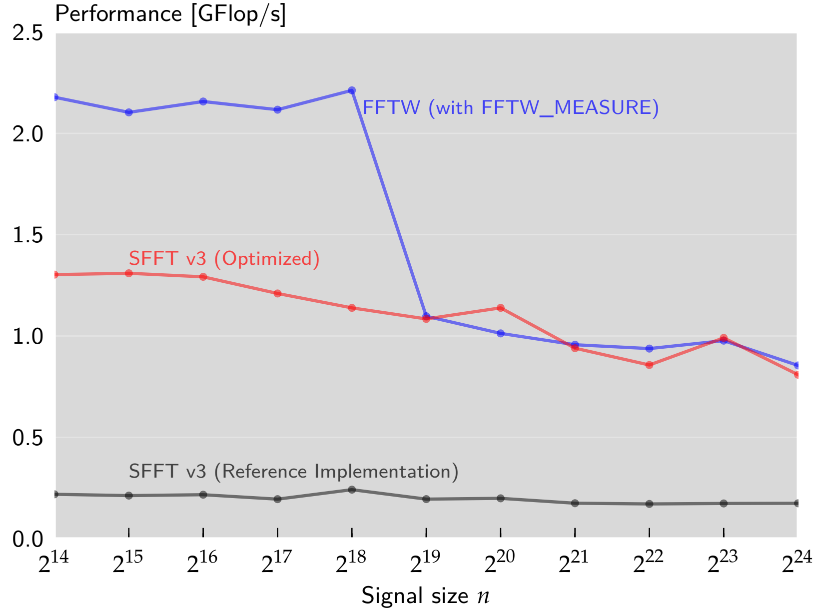



Figure 1.1: Performance of DFTs of signals with $k = 50$ frequency components.

a better asymptotic runtime than $\mathcal{O}(n \log n)$. It is possible to define even better algorithms by adding constraints on the input- and output-vectors $x$ and $\widehat{x}$, however. The *Sparse Fast Fourier Transform (SFFT)*, recently proposed by [HIKP12b], is such an algorithm. The SFFT can be applied to signals $x \in \mathbb{C}^n$ with a sparse frequency domain $\widehat{x}$, i.e. only $k < n$ unknown elements of $\widehat{x}$ are nonzero (while the time domain signal $x$ is still dense).

Besides the algorithmic improvements, new computer architectures are constantly developed and improved. Modern general purpose CPUs feature multi-level caches, instruction level parallelism or vector instruction sets. Additionally, accelerator technologies like GPUs or FPGAs can be used to boost program performance. Parallelism is becoming increasingly important, as modern desktop CPUs typically package multiple cores, or computers can be connected to compute clusters. With this variety of target platforms it is hard for compilers to generate optimal machine code that makes use of all features and runs at high performance. Thus, manually optimized libraries for specific target platforms are being written for all kinds of algorithms.

The original Cooley/Tukey-FFT and similar algorithms have been implemented in such high-performance libraries like *FFTW* (see [FJ]) or *CUFFT* (see [Nvi07]). These implementations make use of modern computer architecture features and are carefully designed to deliver the highest possible

performance. Due to the novelty of the method, no such high-performance implementation existed for the SFFT. In this thesis I aim to address this missing piece by developing a high-performance SFFT library.

Figure 1.1 shows the results of my work. Compared to the reference implementation the SFFT's performance has increased significantly, and it is competitive to FFTW's performance.

## 1.2 Thesis Objective

At the time of writing this thesis, 3 different SFFT algorithm versions were implemented in two different reference implementations. The first reference implementation including SFFT Version 1 and 2 was published on the Sparse Fast Fourier Transform Website [KHPI], the second reference implementation including SFFT Version 3 is still unpublished but was kindly provided by the authors.

The reference implementations are written in standard C++ code, single-threaded and without any hardware-specific modifications. The FFTW library is used for internal DFT computations. On a recent Intel Sandy-Bridge architecture, the measured performance of all algorithms was between 0.2 GFlop/s and 0.5 GFlop/s, which is $4.5\,\% - 11.4\,\%$ of the system's scalar, single-threaded peak performance, or less than $1\,\%$ of the system's vectorized, multi-threaded peak performance (see also Chapter 3).

The goal of this thesis is to take the SFFT reference implementations, unify them into a single codebase, and optimize it to obtain a high-performance library. Optimizations include the usage of vector instructions sets like SSE, memory-hierarchy targeted improvements, and multithreading.

The optimization is guided by a preceding performance analysis of the original, unoptimized code, and the optimization outcome will be evaluated and compared to other state-of-the art DFT implementations. This thesis shall result in an easy-to-use, highly optimized library implementing the various SFFT algorithms.

## 1.3 Related Work

Currently, 4 different SFFT algorithms exist. They are defined in [HIKP12b] (SFFT v1 and v2) and [HIKP12a] (SFFT v3 and v4), where also some benchmark results are shown. SFFT v3 can only be applied to exactly $k$-sparse signals, whereas SFFT v1, SFFT v2, and SFFT v4 can also be applied to noisy signals. The algorithms's asymptotic runtimes are $\mathcal{O}\left(\log n \sqrt{nk \log(n)}\right)$

for SFFT v1, $\mathcal{O}\left(\log n \sqrt[3]{nk^2 \log(n)}\right)$ for SFFT v2, $\mathcal{O}(k \log n)$ for SFFT v3, and $\mathcal{O}(k \log n \log(n/k))$ for SFFT v4. The current implementation of SFFT v1 and v2 only supports a limited set of input sizes, and no implementation at all exists for SFFT v4.

Regarding high performance DFT libraries, the *FFTW* library [FJ] is a well known and very fast FFT library. It uses different techniques like codelet generation and runtime autotuning to achieve a high performance. It will often be used as a reference in benchmarks throughout this thesis. The key concepts of its implementation are discussed in [FJ05]. Currently, no sparse DFT algorithm is implemented in FFTW.

There have been different previous approaches to implement DFTs by exploiting signal sparsity. An overview of the different algorithms is given in Table 1.1.

*Pruning*, as described in [Mar71], is a method that is applicable when a large portion of an FFT's input vector is known to be zero. With the pruning technique, the FFT is reduced to the operations that actually contribute to the result. Output pruning, i.e. parts of the output vector are known to be zero, can also be done and was described in [SR79]. The asymptotic runtime of the Pruned FFT is $\mathcal{O}(n \cdot \log k)$, where $k$ is the number of nonzero inputs or outputs. In [FP09], the pruning algorithm was reformulated in terms of the Kronecker product notation, making it suitable for use with the code generation tool Spiral. The paper reports up to 30 % speedup over competing FFT implementations. A drawback of pruning is that the signal's sparsity pattern has to be known in advance, i.e. it has to be known which signal coefficients are nonzero.

Another algorithm for sparse signals (sparse in the frequency representation) is *FADFT-2*, implemented in the *AAFFT* library [Iwe]. Using random sampling, this algorithm achieves a runtime of $\mathcal{O}(k \cdot \text{polylog}(n))$, where $k$ is the number of Fourier coefficients to be reconstructed and $n$ the signal size. An evaluation of AAFFT is given in [IGS07]. The sampling algorithm used in AAFFT, described in [GST08], is similar to the SFFT approach.

Various applications of the SFFT are thinkable. One particular application is presented in [HAKI12], where it is shown how to apply the Sparse Fast Fourier Transform to a specific algorithm in the GPS system.

In [IM], an SFFT extension to two-dimensional Fourier Transforms on sparse signals is discussed.

## 1.4 Contributions and Results

My implementation is based on the sourcecode provided by Haitham Hassanieh, Piotr Indyk, Dina Katabi, and Eric Price, which was partly

| | *SFFT v1* | *SFFT v2* | *SFFT v3* |
|---|---|---|---|
| Asymptotic Runtime | $\mathcal{O}\left(\log n \sqrt{nk \log(n)}\right)$ | $\mathcal{O}\left(\log n \sqrt[3]{nk^2 \log(n)}\right)$ | $\mathcal{O}\left(k \log n\right)$ |
| Algorithm | Probabilistic | Probabilistic | Probabilistic |
| Constraints | Restricted set of input parameters | Restricted set of input parameters | Only exactly $k$-sparse signals |
| Implementation | [KHPI] | [KHPI] | Unpublished |

| | *SFFT v4* | *Pruning* | *AAFFT* |
|---|---|---|---|
| Asymptotic Runtime | $\mathcal{O}\left(k \log n \, \log(n/k)\right)$ | $\mathcal{O}(n \cdot \log k)$ | $\mathcal{O}(k \cdot \text{polylog}(n))$ |
| Algorithm | Probabilistic | Deterministic | Probabilistic |
| Constraints | — | Sparsity pattern must be known in advance | — |
| Implementation | No Implementation | [FP09] | [Iwe] |

Table 1.1: Different DFT algorithms for sparse signals and their properties.

published on the website [KHPI]. During my thesis work I unified the implementations of all three SFFT versions and merged them into a single codebase. I developed a user-friendly, extensible, yet simple user interface. The code was packaged into an easily installable and documented library.

I performed a detailed analysis of the library's baseline performance and identified hotspots and possible performance blockers. Various analysis techniques were used, including the recently developed roofline analysis method [WWP09].

I optimized the library where possible, leveraging modern CPU features like SSE and multithreading. The implementation was tested on recent Intel hardware.

The final implementation was benchmarked and its performance compared to the reference implementation. Depending on the algorithm version and input parameters, a two- to five-fold speedup was achieved and the optimized algorithm performance is competitive to high-performance libraries like FFTW.

## 1.5 Outline

In chapter 2 the theory behind the Sparse Fast Fourier algorithms is explained in detail. Pseudocode of all relevant versions is given and some

implementation details are discussed.

A detailed performance analysis of the SFFT is given in chapter 3. This includes a derivation of the algorithms's asymptotic runtimes, runtime benchmarks, performance estimations and roofline analysis. Based on these metrics, claims about algorithm properties are made.

In chapter 4 several optimizations and their effects are briefly explained. Some experiments investigating individual optimization's influence are described.

Evaluation of optimizations and the final SFFT implementation is depicted in chapter 5.

A final discussion of the algorithm's performance in chapter 6, relating it to other high-performance libraries, concludes this thesis.

The optimized implementation of the Sparse Fast Fourier Transform algorithms is packaged in a library with a simple user interface. A software manual with a detailed description of how to use this library is given in appendix A.

# Chapter 2

# The Sparse Fast Fourier Transform: An Overview

Given a time-domain signal $x \in \mathbb{C}^n$ with a $k$-sparse frequency domain $\hat{x}$, the *Sparse Fast Fourier Transform (SFFT)* will compute the $k$ nonzero frequency coefficients of $\hat{x}$. The SFFT exists in different versions. For versions 1, 2, and 4, $\hat{x}$ does not need to be exactly $k$-sparse, i.e. the signal may be noisy. Then, the algorithm output $\hat{x}'$ fulfills the following guarantee:

$$\left\| \hat{x} - \hat{x}' \right\|_\infty^2 \leq \epsilon \left\| \hat{x} - y \right\|_2^2 / k + \delta \left\| x \right\|_1^2 ,$$

where $\epsilon$ and $\delta$ are accuracy parameters and $y$ is the $k$-sparse vector minimizing $\| \hat{x} - y \|_2^2$.

SFFT Versions 1 and 2 were first described in [HIKP12b]. Improvements were introduced by the authors in their follow-up paper [HIKP12a], where Version 3 and 4 are defined. The Sparse Fast Fourier Transform is a probabilistic algorithm, i.e. the $k$ significant frequency components of a $k$-sparse signal are reconstructed with a finite probability. All of the 4 versions will be discussed in this chapter. Some common ideas are shared by all versions:

- The input vector $x$ is *permuted* with random parameters.
- *Window functions* are used as filters to extract a subset of the $n$ elements of the signal. This step is crucial to achieve a sub-linear runtime: it allows extracting information out of the input vector without touching all $n$ elements.
- Using subsampling and a low-dimensional FFT, the signal's Fourier coefficients can be *binned* into a small number of bins.

- By repeating the above steps multiple times and combining the results, the $k$ nonzero Fourier coefficients can be found with high probability.

In the following sections the different versions of the SFFT are explained in detail. But first some of the basic principles and methods that are shared by all versions of the Sparse Fast Fourier Transform are explained. The following nomenclature, definitions and reasoning follow very closely the publications [HIKP12b] and [HIKP12a] by the inventors of the SFFT.

## 2.1 Notation

Several conventions and notations are used in this thesis. A time-domain signal is written as $x$, the DFT of the signal is written as $\widehat{x}$. The notation $[n]$ is defined as the set $\{0, 1, \ldots, n-1\}$. All vector indices are implicitly calculated modulo the vector size, e.g. $x_i$ of an $n$-dimensional $x$ is actually $x_{i \bmod n}$. A set of vector elements can be written as a vector subscripted with a set of indices, for example $x_I = \{x_i \mid i \in I\}$.

## 2.2 Basic Principles

In the following sections, I will use the following definition of the DFT without the constant scaling factor:

$$\widehat{x}_i = \sum_{j \in [n]} \omega^{ij} x_j, \qquad i = 0, \ldots, n-1.$$

This makes some proofs easier, but is not relevant in practical implementations.

### 2.2.1 *Random Spectrum Permutation*

The first important tool for the SFFT is spectrum permutation as defined in Definition 1:

**Definition 1.** *Let $\sigma$ be invertible modulo $n$, i.e. gcd($\sigma$, $n$) = 1, and $\tau \in [n]$. Then, $i \mapsto \sigma i + \tau \mod n$ is a permutation on $[n]$. The associated permutation $P_{\sigma,\tau}$ on a vector $x$ is then given by*

$$(P_{\sigma,\tau} x)_i = x_{\sigma i + \tau},$$

*$P_{\sigma,\tau} x$ is a* permutation *of $x$.*

This permutation has an interesting property: when a permutation is applied to a time-domain signal $x$, the signal's frequency domain $\widehat{x}$ is also permuted. This is derived in Lemma 1, and an example of a permutation applied to a signal is shown in Figure 2.1.

**Lemma 1.** *Let $P_{\sigma,\tau}$ be a permutation and $x$ be an $n$-dimensional vector. Then*

$$\left(\widehat{P_{\sigma,\tau}x}\right)_{\sigma i} = \widehat{x}_i \omega^{-\tau i}.$$

*Proof.* Let $i$ be arbitrary chosen from $1, \ldots, n$. Then,

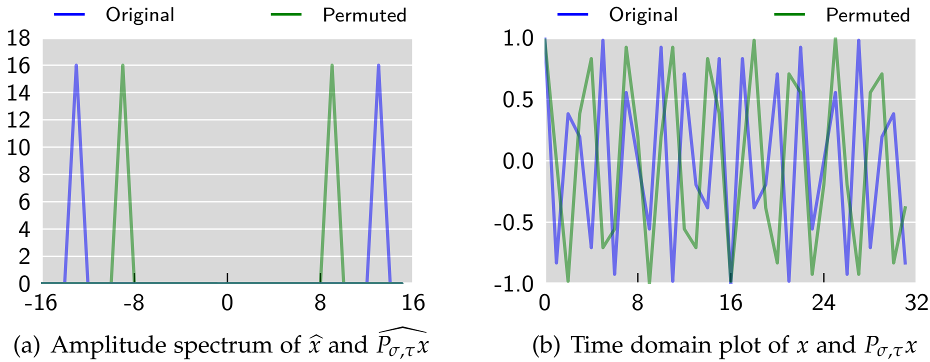
$$\begin{aligned}
\left(\widehat{P_{\sigma,\tau}x}\right)_i &= \sum_{j\in[n]} \omega^{ij} x_{\sigma j+\tau} \\
&= \sum_{a\in[n]} \omega^{i\sigma^{-1}(a-\tau)} x_a && \text{(with } a = \sigma j + \tau\text{)} \\
&= \omega^{-i\sigma^{-1}\tau} \sum_{a\in[n]} \omega^{\sigma^{-1}ia} x_a \\
&= \widehat{x}_{\sigma^{-1}i} \omega^{-\tau\sigma^{-1}i}.
\end{aligned}$$

The Lemma follows by substituting $i = \sigma i$. Note that $\omega^{-\tau i}$ changes the phase, but not the magnitude, of $\widehat{x}_i \omega^{-\tau i}$. □

The purpose of this permutation in the SFFT is to reorder a signal's frequency-domain $\widehat{x}$. This way nearby coefficients can be torn apart. In the SFFT, however, we do not have access to the input signals Fourier spectrum as that would involve performing a DFT. The permutation as defined in Definition 1 allows to permute the signal's Fourier spectrum by *modifying the signal's time-domain $x$.*

### 2.2.2 *Window Functions*

An important feature of the SFFT algorithm is that only a part of an input signal is used for computations. Otherwise a sub-linear runtime could not be achieved. Unfortunately, it is not possible to simply cut individual parts out of a signal.




(a) Amplitude spectrum of $\widehat{x}$ and $\widehat{P_{\sigma,\tau}x}$ (b) Time domain plot of $x$ and $P_{\sigma,\tau}x$

Figure 2.1: A signal $x$ before and after permutation.

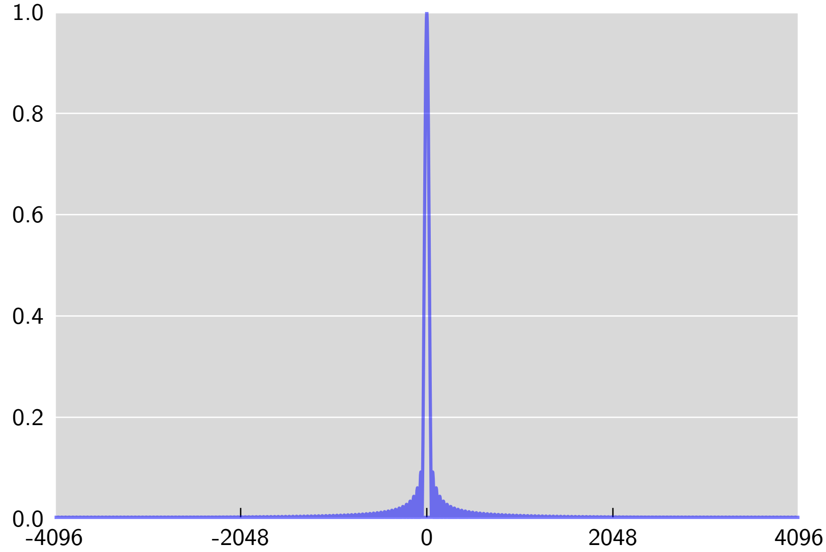



Figure 2.2: Amplitude spectrum of a simple flat window function based on a Gaussian function.

Discrete Fourier Analysis applies to signals that are periodic. If a part is cut out of a periodic signal, discontinuities can appear at the part's boundaries. In the frequency-domain these discontinuities appear as additional frequency components. This effect is called *spectral leakage*.

To extract parts of a signal in a smooth way and avoid spectral leakage, *window functions* are used.

**Definition 2.** *A pair of n-dimensional vectors* $(G, \widehat{G}') = (G_{B,\delta,\alpha}, \widehat{G}'_{B,\delta,\alpha})$ *is a* $(B, \alpha, \delta)$*-parameterized* flat window function *if* $|supp(G)| = \mathcal{O}(\frac{B}{\alpha} \log n/\delta)$ *and*

- $\widehat{G}'_i = 1$ *for* $|i| \leq (1-\alpha)n/(2B)$,
- $\widehat{G}'_i = 0$ *for* $|i| \geq n/(2B)$,
- $\widehat{G}'_i \in [0, 1]$ *for all* $i$,
- $\left\|\widehat{G}' - \widehat{G}\right\|_\infty < \delta$.

These filters have a small pass region where $\widehat{G}$ is 1 and a big region where $\widehat{G}$ is negligible. An example of a flat window function is shown in Figure 2.2; an example where a flat window function is applied to a signal is shown in Figure 2.3.

Using a window function $(G, \widehat{G}')$, a part of size $|\text{supp}(G)|$ can be extracted out of a vector $x$ by multiplying $G$ and $x$ and neglecting the coefficients with value zero. According to the convolution theorem, the multiplication is equivalent to a convolution of $\widehat{G}$ and $\widehat{x}$. Thus, with a smart choice of $\widehat{G}$, one can reduce the spectral leakage effect. For more details, refer to [Har78]. The window functions used in the SFFT algorithms are based on Gaussian functions.

#### 2.2.3 Fast Subsampling and DFT

A frequent step in the SFFT algorithms is computing the DFT of a low-dimensional vector and then subsampling and summing up the result. Lemma 2 shows that this computation can be done very efficiently by reverting these steps.

**Lemma 2.** *Let $B \in \mathbb{N}$ divide $n$, $x$ be an $n$-dimensional vector and $y$ be a $B$-dimensional vector with $y_i = \sum_{j=0}^{n/B-1} x_{i+Bj}$ for $i = 1, \dots, B$. Then, $\widehat{y}_i = \widehat{x}_{i(n/B)}$.*

*Proof.*

$$\begin{aligned}
\widehat{x}_{i(n/B)} &= \sum_{j=0}^{n-1} x_j \omega_n^{ij(n/B)} \\
&= \sum_{a=0}^{B-1} \sum_{j=0}^{\frac{n}{B}-1} x_{Bj+a} \omega_n^{i(Bj+a)n/B} \\
&= \sum_{a=0}^{B-1} \sum_{j=0}^{\frac{n}{B}-1} x_{Bj+a} \omega_n^{ian/B} \\
&= \sum_{a=0}^{B-1} y_a \omega_n^{ian/B}
\end{aligned}$$

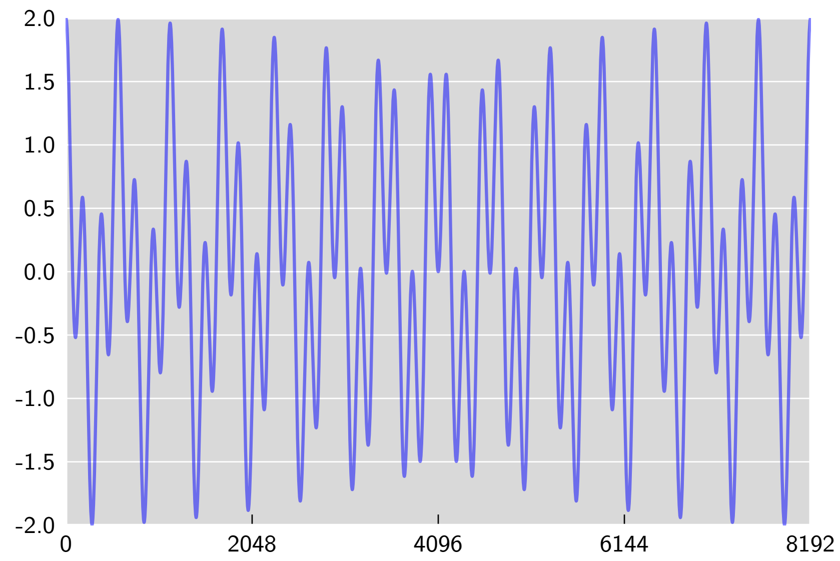
Note that when $\omega_n$ is the $n$-th complex primitive root of unity, $\omega_n^{n/B}$ is the $B$-th complex primitive root of unity $\omega_B$. Thus, it follows $\widehat{x}_{i(n/B)} = \widehat{y}_i$. □

**Corollary 1.** *With the definitions as in Lemma 2, the asymptotic runtime to calculate $\widehat{y}$ is $\mathcal{O}(|\text{supp}(x)| + B \log B)$.*

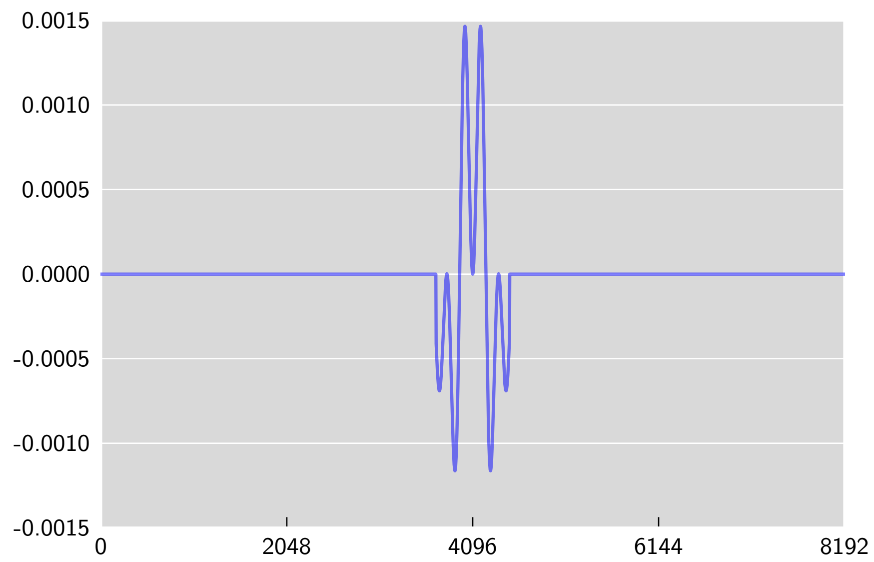
With these definitions and Lemmas it is now possible to define the Sparse Fast Fourier Transform.

## 2.3 SFFT Version 1

Version 1 of the SFFT consists multiple executions of two kinds of loops: *location loops* and *estimation loops*. The purpose of the first kind, location




(a) A signal.




(b) The signal after a window function was applied.

Figure 2.3: Gaussian Standard Window Function applied to a sample signal.

loops, is to generate a list of candidate coordinates $I$. Candidate coordinates $i \in I$ have a certain probability of being indices of one of the $k$ significant, nonzero coefficients in $\widehat{x}$. By running multiple iterations of the location loops, this probability can be increased. The second type, estimation loops, are used to exactly determine the frequency coefficients $\widehat{x}_I$ of a given set of coordinates $I$.

**Algorithm 1** SFFT v1.

Input: $x \in \mathbb{C}^n$, $k < n$, $L \in \mathbb{N}$. Output: A $k$-sparse vector $\widehat{x}$.

1. Run a number of $L$ *location loops*, returning $L$ sets of coordinates $I_1, \ldots, I_L$.
2. *Count* the number $s_i$ of occurrences of each found coordinate $i$, that is: $s_i = |\{r | i \in I_r\}|$.
3. Only keep the coordinates which occurred in at least half of the location loops. $I' = \{i \in I_1 \cup \cdots \cup I_L | s_i > L/2\}$.
4. Run a number of $L$ *estimation loops* on $I'$, returning $L$ sets of frequency coefficients $\widehat{x^r_{I'}}$.
5. *Estimate* each frequency coefficient $\widehat{x}_i$ as $\widehat{x}_i = \text{median}\{x^r_i | r \in \{1, \ldots, L\}\}$. The median is taken in real and imaginary components separately.

---

This is an overview of the steps that are performed in location loops:

1. *Random spectrum permutation.* Randomly choose a $\sigma$ invertible mod$n$ and $\tau \in [n]$. Permute the input vector $x$ with the permutation $P_{\sigma,\tau}$: $(P_{\sigma,\tau}x)_i = x_{\sigma i + \tau}$.
2. *Apply filter.* Using a flat window function $G$, compute the filtered and permuted vector $y = G \cdot (P_{\sigma,\tau}x)$.
3. *Subsampling and FFT.* With $B$ dividing $n$, compute $\widehat{z}$ with $\widehat{z}_i = \widehat{y}_{i(n/B)}$ for $i \in [B]$. Using Lemma 2, $\widehat{z}$ can be computed as the $B$-dimensional DFT of $z$, where $z_i = \sum_{j=0}^{n/B-1} y_{i+Bj}$ for $i \in [B]$.
4. *Cutoff.* Only keep the $d \cdot k$ coordinates of maximum magnitude in $\widehat{z}$. Those are the so-called *bins* where the non-negligible coefficients were hashed to. Call the set of coordinates that are kept $J$. $d$ is a parameter of SFFT v1.
5. *Reverse hash function*. Steps 1 – 3 describe a hash function $h_\sigma : [n] \to [B]$ that maps each of the $n$ coordinates of the input signal to one

of $B$ bins. $h_\sigma$ can be defined by $h_\sigma(i) = \text{round}(\sigma i B/n)$. This hash function has to be reversed for the coordinates in $J$. The output of a location loop is the set of coordinates mapping to one of the bins in $J$: $I = \{i \in [n] \,|\, h_\sigma(i) \in J\}$.

The random permutation of the spectrum is performed to get different results in subsequent location loop runs. This is necessary for two reasons. First, the output of a single location loop is only guaranteed to contain the correct $k$ nonzero frequencies at a constant probability. Multiple location loop executions increase this probability. Second, in a single location loop many coordinates map to the same bin. Using multiple runs, each with a different random spectrum permutation, it is unlikely that a non-significant frequency (where $\widehat{x}_i$ is very small or zero) maps to one of the nonzero bins $J$ each time and is therefore falsely considered as one of the candidate coordinates.

The flat window function in step 2 is used as a filter to extract a certain set of elements of $x$. Note that although the filter consists of $n$ elements, most of these elements are negligible and it is sufficient to multiply with $w$ significant elements.

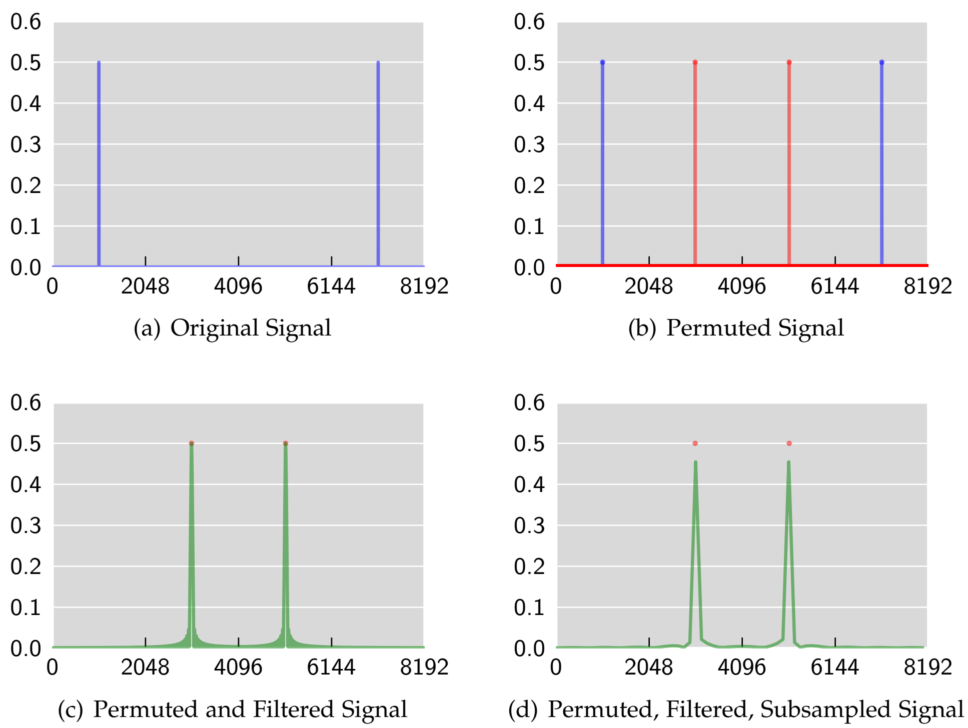
In step 3 the permuted and filtered input $y$ is now hashed to a set of $B$ bins. Therefore $y$ is subsampled, summed up, and a $B$-dimensional FFT is performed. With high probability, each bin contains at most one non-negligible coefficient.

After subsampling, $B$-dimensional FFT and removing non-significant parts of the signal, the original signal coefficients have to be reconstructed. This is done by reverting the hash function $h_\sigma$. This generates more than $k$ outputs, but, after all location loops were executed, only the $k$ coefficients with highest occurrence numbers in all location loops are kept.

The purpose of estimation loops, the second type of loops in SFFT v1, is to reconstruct the exact coefficient values given a set of coordinates $I$. The implementation of estimation loops is similar to location loops: they share the first 3 steps. The fourth and last step in an estimation loop is *"Given a set of coordinates $I$, estimate $\widehat{x}_i$ as $\widehat{x}'_i = \widehat{z}_{h_\sigma(i)} \omega^{\tau i} / \widehat{G}_{\sigma(i)}$"*, which basically removes the phase change due to the permutation and the effect of the filter.

When multiple frequencies hash to the same bin, a hash collision occurs and the estimation fails. To compensate this, the value of $\widehat{x}'_i$ can be set to the median of the corresponding outputs of all estimation loops.

Figure 2.4 shows the various signal manipulations that are performed in the location and estimation loops. Algorithm 1 shows pseudocode for SFFT v1.0 using location- and estimation loops. Figure 2.5 shows a flow diagram of SFFT v1.




(a) Original Signal

(b) Permuted Signal

(c) Permuted and Filtered Signal

(d) Permuted, Filtered, Subsampled Signal

Figure 2.4: Effects of the individual signal manipulation steps in the frequency domain.

Figure 2.5: A simplified flow diagram of SFFT v1.

SFFT v1 depends on several parameters, e.g., the number of location and estimation loops, parameters for the Gaussian filter or the number of bins. Each of these parameters is highly dependent on the algorithm's input, i.e., the signal size $n$ and the number of nonzero coefficients $k$. Unfortunately, this complicates practical implementations. The reference implementation from [HIKP12b] therefore hard-codes parameter settings for a number of input scenarios. Since it is based on this reference implementation, my implementation suffers from the same problems The same problem applies to SFFT v2, but SFFT v3 does not have this restriction.

## 2.4 SFFT Version 2

Version 2 of the SFFT is very similar to version 1. The only difference is that a heuristic is used to find the signal's significant coefficients quickly. The heuristic based on a special filter, a modified version of the algorithm described in [Man95]. Here, it will be referred to as *Mansour filter*.

Mansour filter loops can be implemented as follows. Let $w_M$ be the size of the filter. Then:

1. Choose a random offset $\tau \in [n/w_M]$.
2. Subsample the input vector by computing $z_i = x_{\tau+i \cdot w_M}$ for $i \in [w_M]$.
3. Compute $\widehat{z}$ as the DFT of $z$.
4. Return the coordinates of maximum magnitude in $\widehat{z}$.

Remember that the window functions are used to extract parts of the signal and keep spectral leakage minimal. The Mansour filter has no spectral leakage at all. This is a major advantage since the error is reduced, and thus, the Mansour filter can speed up the execution of the SFFT. But there are also some drawbacks. One problem is that permutations cannot be used to resolve hash collisions, since only the offset is random. However, according to [HIKP12b] this is not an issue in practical implementations.

## 2.5 SFFT Version 3

While the core ideas of version 3 of the SFFT are still similar to version 1 and 2, this version introduces two major improvements.

The first improvement is based on the observation that once a frequency coefficient of the signal was found and estimated, it can be removed from the signal. This fact can be used to reduce the amount of work to be done in subsequent steps. Unfortunately updating the whole signal would

**Algorithm 2** SFFT v2.

Input: $x \in \mathbb{C}^n$, $k < n$, $L \in \mathbb{N}$. Output: A $k$-sparse vector $\widehat{x}$.

1. Run a number of $L_1$ *Mansour loops*, returning $L_1$ sets of coordinates $I_1, \ldots, I_{L_1}$.
2. Run a number of $L_2$ *location loops*, returning $L$ sets of coordinates $I_{L_1+1}, \ldots, I_{L_1+L_2}$. Let $L = L_1 + L_2$.
3. *Count* the number $s_i$ of occurrences of each found coordinate $i$, that is: $s_i = |\{r | i \in I_r\}|$.
4. Only keep the coordinates which occurred in at least half of the location loops. $I' = \{i \in I_1 \cup \cdots \cup I_L | s_i > L/2\}$.
5. Run a number of $L$ *estimation loops* on $I'$, returning $L$ sets of frequency coefficients $\widehat{x^r_{I'}}$.
6. *Estimate* each frequency coefficient $\widehat{x}_i$ as $\widehat{x}_i = \text{median}\{x^r_i | r \in \{1, \ldots, L\}\}$. The median is taken in real and imaginary components separately.

---

require $\mathcal{O}(n)$ operations and is therefore too costly. However, it is not necessary to update the input signal. Instead, it is sufficient to update the $B$-dimensional output of a measurement (that is: application of filter, DFT and subsampling). This way the removal of the effects of already found coefficients can be done in $\mathcal{O}(B)$ time.

The second important addition in SFFT v3 is an improved scheme for finding the signal's significant frequency coordinates using individual measurements. In SFFT v1 and v2, multiple location loops were run and their results combined in order to get correct candidate coordinates at a high probability. [HIKP12a] proves that two distinct measurements (calls to `HashToBins`) are enough.

The idea here is to perform the measurements with similar permutations that only differ in the phase-altering parameter. Permutations, as defined in Definition 1, have two parameters $\tau$ and $\sigma$. As it was mentioned in Lemma 1 the parameter $\tau$ changes the phase of the signal because a term $\omega^{\tau j}$ is implicitly multiplied to each coordinate $j$ of the frequency-domain signal. The two calls to `HashToBins` are performed with the same $\sigma$, but one time with $\tau = 0$ and one time with $\tau = 1$.

When no hash collision occurs only a single nonzero frequency coefficient maps to a bin. Since the phase change in the second measurement

also depends on the bin's coordinate, the coordinate can be reconstructed out of the phase difference of the two measurements.

The drawback of this approach is that it is only applicable to exact $k$-sparse signals, i.e. $k$-sparse signals which are not affected by any noise.

The complete SFFT v3 algorithm, including all improvements, is shown in Algorithm 3. A flow diagram is depicted in Figure 2.6.

## 2.6 SFFT Version 4

Version 4 of the SFFT algorithm uses the same ideas as version 3, but eliminates the restriction that only exact $k$-sparse signals can be used. It does this by using the same scheme for finding finding candidate coordinates as version 3, but allowing again more than two distinct measurements and reconstructing a finite number of bits of the coordinates in each measurement. This approach is similar to a binary search, in each step the region of a frequency is further reduced. The details of this algorithm are very complex, and at the time of this writing no implementation of SFFT v4 exists. Therefore it is not treated in this thesis. The details of the algorithm are described in [HIKP12a].

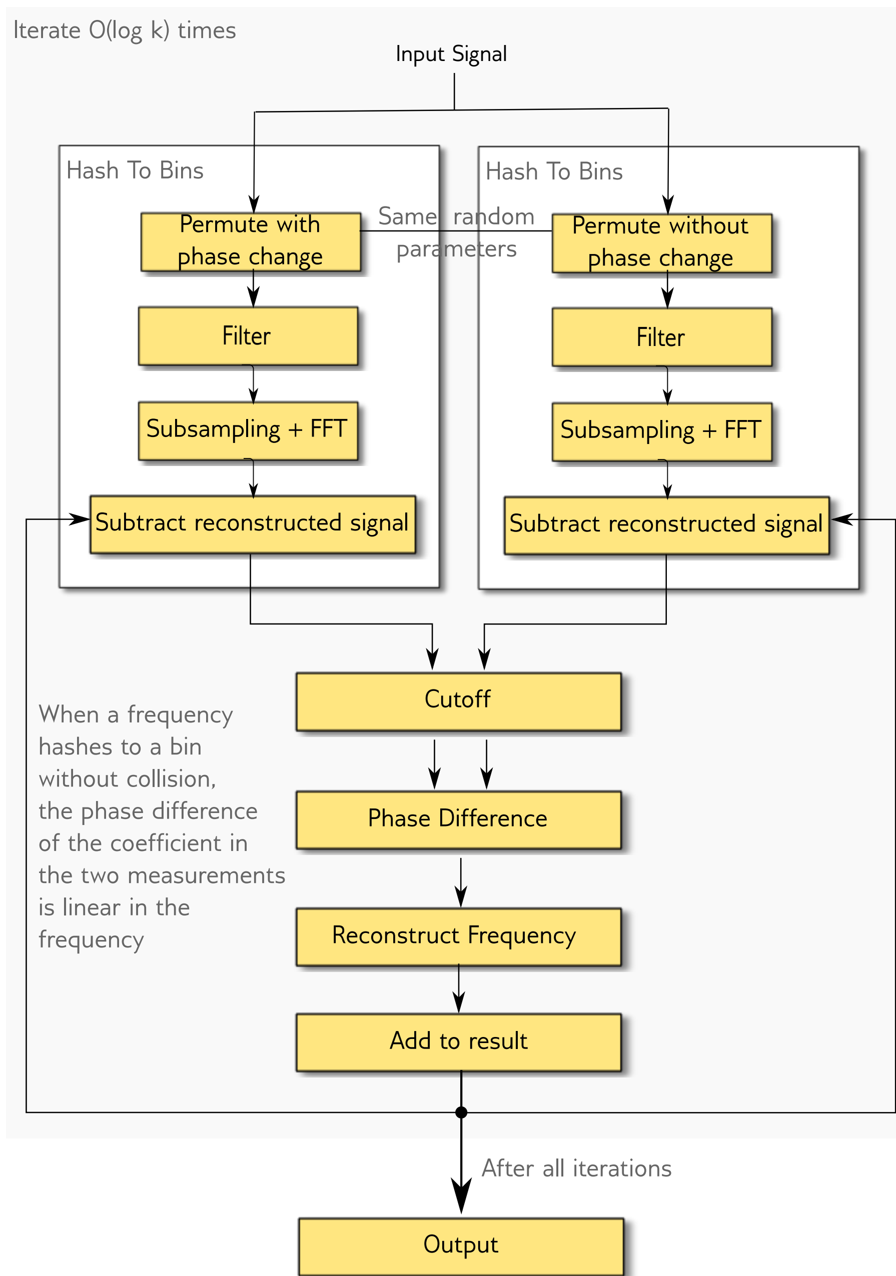



Figure 2.6: A simplified flow diagram of SFFT v3.

**Algorithm 3** SFFT v3.

Input: $x \in \mathbb{C}^n$, $k < n$, parameters $B, \alpha, \beta, \delta$. Output: A $k$-sparse vector $\widehat{x}$.

1. Initialize $z \in \mathbb{C}^n$ to 0.
2. For $t = 1 \ldots \log k$ do
   (a) Randomly choose an odd number $\sigma$ and any number $b$ from $[n]$, and generate the permutations $P_{\sigma,0,b}$ and $P_{\sigma,1,b}$.
   (b) Perform two measurements with the function HashToBins using the two permutations on the input signal. Assume the output of the individual measurements are $\widehat{u}$ and $\widehat{u}'$.
   The function HashToBins works like this:
      i. Let $B = k/(2^t \cdot \beta)$.
      ii. Compute $\widehat{y}_{jn/B}$ for $j \in [B]$, where $y = G_{B,\alpha,\delta} \cdot (P_{\sigma,a,b}x)$.
      iii. Remove the effects of the already found frequencies by computing $\widehat{y}'_{jn/B} = \widehat{y}_{jn/B} - \left(\widehat{G'_{B,\alpha,\delta}} * \widehat{P_{\sigma,a,b}z}\right)_{jn/B}$ for $j \in [B]$.
      iv. Return $\widehat{v}_j = \widehat{y}'_{jn/B}$.
   HashToBins works very similar to some steps of SFFT v1 location loops. Constants $\alpha, \beta, \delta$ have to be set appropriately.
   (c) For every nonzero bin $\widehat{u}_j$:
      i. Set $a = \widehat{u}/\widehat{u}'$.
      ii. Assuming there is no hash collision (i.e. only one frequency coordinate was hashed to bucket $j$), the phase of $a$ is now linear in the frequency coordinate. One can reconstruct it using $i = \sigma^{-1}(\text{round}(\text{phase}(a)\frac{n}{2\pi})) \mod n$.
      iii. $\widehat{u}_j$ can be used as estimate for the magnitude of the coefficient of frequency $i$. Thus, set $\widehat{z}_i = \widehat{z}_i + round(\widehat{u}_j)$.
3. $\widehat{z}$ is an approximation to the DFT of the $k$-sparse signal $\widehat{x}$.

# Chapter 3

# Performance Analysis

## 3.1 Asymptotic Runtime

In this section, asymptotic runtime bounds for the Sparse Fast Fourier Transform algorithms are derived. Table 3.1 shows an overview. The derivations are based on [HIKP12b] and [HIKP12a].

Table 3.1: The different versions of the Sparse Fast Fourier Transform and their asymptotic runtimes.

| *Algorithm Version* | *Cost* |
|---|---|
| SFFT 1 | $\mathcal{O}\left(\log n \sqrt{nk \log(n)}\right)$ |
| SFFT 2 | $\mathcal{O}\left(\log n \sqrt[3]{nk^2 \log(n)}\right)$ |
| SFFT 3 | $\mathcal{O}\left(k \log n\right)$ |
| SFFT 4 | $\mathcal{O}\left(k \log n \ \log(n/k)\right)$ |

### 3.1.1 *Sparse Fast Fourier Transform Version 1*

The Sparse Fast Fourier Transform algorithm in Version 1 consists of multiple location and estimation loops. The runtime cost of these loops is $\mathcal{O}(B \log n)$ as it will be shown in Lemma 4. First, another useful Lemma will be proved.

**Lemma 3.** *Let L be the number of SFFT v1 location loops and an SFFT v1 run. Let $I_r$ be the output of location loop $r$. Let $I'$ be the set of coordinates that occur in at least half of the location loops. Then, $I' \leq 2dkn/B$.*

*Proof.* The coordinates in $I'$ occur in at least $L/2$ location loop outputs. If $I_r$ is the output of location loop $r$, we have

$$\sum_{r=1}^{L} |I_r| \geq \frac{L}{2}|I'|$$

Since $|I_r| = dkn/B$ it is

$$\begin{aligned} L \cdot dk\frac{n}{B} &\geq \frac{L}{2}|I'| \\ \Rightarrow |I'| &\leq 2 \cdot dkn/B. \end{aligned}$$

□

Using this Lemma one can now proof Lemma 4.

**Lemma 4.** *Let $w = \mathcal{O}(B \log \frac{n}{\delta})$ and $\delta = \mathcal{O}(n^{-c})$. Let $B \in \mathbb{N}$ divide $n$. Then, SFFT Version 1 location and estimation loops run in $\mathcal{O}(B \log \frac{n}{\delta})$.*

*Proof.* The first few steps are common in location- and estimation loops. They are:

- Permuting the input vector with a random permutation.
- Multiplying a part of the permuted input vector with the support of the filter $G$.
- Subsampling the result of the multiplication.
- And performing a DFT on the $B$-dimensional result vector.

The input vector does not actually have to be permuted, but the filter multiplication can be implemented so that the vector is accessed in the correct, permuted order. The cost of this step is then $\mathcal{O}(w)$, since $w$ is the size of the support of $G$. The subsampling step takes again time $\mathcal{O}(w)$, since $w$ elements have to be summed up. The resulting vector is $B$-dimensional. A $B$-dimensional FFT takes time $\mathcal{O}(B \log B)$. The overall asymptotic cost for these steps is therefore:

$$\begin{aligned} c_{inner} &= \mathcal{O}(w) + \mathcal{O}(w) + \mathcal{O}(B \log B) \\ &= \mathcal{O}(w + B \log B) \end{aligned}$$

Substituting $w$ into the above equation results in

$$\begin{aligned} c_{inner} &= \mathcal{O}(B \log \frac{n}{\delta} + B \log B) \\ &= \mathcal{O}(B \log \frac{n}{\delta}) \end{aligned}$$

**a) Location Loops**
In location loops two steps are performed additionally to the steps above:

- Given the $B$-dimensional result vector, determine the $d \cdot k$ coordinates of maximum magnitude,
- and compute the elements mapping to these coordinates.

Determining the $d \cdot k$ coordinates is a straightforward computation and requires $\mathcal{O}(B)$ operations. As mentioned before, location loops implement a hash function $h_\sigma : [n] \to [B]$. For each coordinate $i$ of the $d \cdot k$ coordinates, there are $n/B$ coordinates $j_r$ with $h_\sigma(j_r) = i$. The overall running time to compute the location loop output is therefore $\mathcal{O}(dkn/B)$.

For location loops, we can estimate the computational cost as

$$\begin{aligned} c_{loc} &= c_{inner} + \mathcal{O}(dk\frac{n}{B}) \\ &= \mathcal{O}(B \log \frac{n}{\delta} + dk\frac{n}{B}). \end{aligned}$$

**b) Estimation Loops**
The last step in estimation loops is to compute estimates of $\widehat{x}_{I'}$ for a given set of coordinates $I'$. The cost of this operation is $\mathcal{O}(|I'|) = \mathcal{O}(dkn/B)$ (Lemma 3). Thus, the total cost of estimation loops is

$$\begin{aligned} c_{est} &= c_{inner} + \mathcal{O}(dkn/B) \\ &= \mathcal{O}(B \log \frac{n}{\delta} + dkn/B). \end{aligned}$$

□

The asymptotic runtime of SFFT v1 can now be derived, as it is shown in the next theorem:

**Theorem 1.** *Let $w = \mathcal{O}(B \log \frac{n}{\delta})$, $\delta = \mathcal{O}(n^{-c})$, $B = \mathcal{O}(\sqrt{\frac{nk}{\epsilon \log n/\delta}})$, $L = \mathcal{O}(\log n)$ and $d = \mathcal{O}(1/\epsilon)$. The SFFT Version 1 has an asymptotic runtime cost of*

$$c_{SFFTv1} = \mathcal{O}\left(\log(n)\sqrt{nk\log(n)}\right).$$

*Proof.* The cost of a SFFT 1 execution is determined by $L$ inner loops, plus the cost of combining the results of these inner loops:

$$c_{SFFTv1} = L \cdot c_{loc} + L \cdot c_{est} + c_{output}.$$

Computing the output involves computing medians of $L$-element sets for all coordinates in $I'$, so the cost of constructing the output can be estimated as

$$c_{output} = \mathcal{O}(L \cdot |I'|).$$

Combining this with Lemmas 3 and 4, the cost of SFFT v1 can be estimated as

$$\begin{aligned}
c_{SFFTv1} &= \mathcal{O}(L \cdot c_{loc} + L \cdot c_{est} + c_{output}) \\
&= \mathcal{O}(L \cdot (B \log \frac{n}{\delta} + dk\frac{n}{B}) + L \cdot (B \log \frac{n}{\delta} + dk\frac{n}{B}) + L \cdot dk\frac{n}{B})) \\
&= \mathcal{O}(L \cdot B \log \frac{n}{\delta} + L \cdot dk\frac{n}{B}).)
\end{aligned}$$

Substituting the parameters in the equation results in

$$\begin{aligned}
c_{SFFTv1} &= \mathcal{O}\left(\log(n)\sqrt{\frac{nk}{\epsilon \log(\frac{n}{\delta})}} \log(\frac{n}{\delta}) + \log(n)kn\frac{1}{\epsilon}\sqrt{\frac{\epsilon \log(\frac{n}{\delta})}{nk}}\right) \\
&= \mathcal{O}\left(\log(n)\sqrt{\frac{nk \log(\frac{n}{\delta})}{\epsilon}} + \log(n)\sqrt{\frac{nk \log(\frac{n}{\delta})}{\epsilon}}\right) \\
&= \mathcal{O}\left(\log(n)\sqrt{\frac{1}{\epsilon} nk \log(\frac{n}{\delta})}\right) \\
&= \mathcal{O}\left(\log(n)\sqrt{nk \log(n)}\right).
\end{aligned}$$

□

### 3.1.2 *Sparse Fast Fourier Transform Version 3*

SFFT v3 consists of $\log k$ executions of an inner loop. The inner loop of SFFT v3 consists of two executions of the HashToBins function. HashToBins is similar to the inner loop of SFFT Version 1, but it has an additional step where the already reconstructed solution is subtracted from the $B$-dimensional result vector. This requires an additional $\mathcal{O}(B)$ operation. The complexity of HashToBins is still $\mathcal{O}(B \log n)$. This observation can be used to prove the next theorem:

**Theorem 2.** *The asymptotic runtime of SFFT version 3 is*

$$c_{SFFTv3} = \mathcal{O}(k \log n).$$

*Proof.* In the $t$-th iteration of SFFT v3's outer loop, $B$ is chosen as $B = \frac{k}{\beta 2^t}$. Therefore the runtime of the $t$-th iteration of the inner loop can be estimated as

$$c_{HashToBins}(t) = \mathcal{O}\left(\frac{k}{\beta 2^t}\log n\right)$$

Summing up over $\log k$ iterations results in

$$\begin{aligned}
c_{SFFTv3} &= \mathcal{O}\left(\sum_{t=0}^{\log k} c_{HashToBins}(t)\right) \\
&= \mathcal{O}\left(\sum_{t=0}^{\log k} \frac{k}{\beta 2^t}\log n\right) \\
&= \mathcal{O}\Big(\log n \underbrace{\sum_{t=0}^{\log k} \frac{k}{\beta 2^t}}_{\text{geometric series}}\Big) \\
&= \mathcal{O}\left(k \log n.\right)
\end{aligned}$$

□

## 3.2 Benchmarks

In this section some benchmarks of the non-optimized SFFT implementations are presented. All benchmarks are warm-cache measurements performed on an Intel(R) Xeon(R) E5-2660 CPU at 2.20 GHz with the Sandy Bridge micro-architecture, consisting of 8 physical cores or 16 virtual cores (using Hyper-Threading technology). Each core contains a 64 KB L1-cache and a 256 KB L2-cache. An additional 20 MB L3-cache is shared among the cores. Error bars in plots show 95 % confidence intervals.

The first benchmarks are runtime experiments reproducing the results found by [HIKP12b] and [HIKP12a], where the runtime of SFFT v1, v2, v3 is compared to the runtime of an equivalent FFTW call, which does not exploit the signal's sparsity. First, the signal sparsity $k$ was kept constant ($k = 50$) and the signal size $n$ was varied. Results of this experiment are shown in Figure 3.1. Second, the signal sparsity $k$ was varied and the signal size $n$ was kept constant to $2^{22}$. Results of this experiment are shown in Figure 3.2. FFTW's runtime is constant in the second experiment, since FFTW's runtime only depends on the signal size, not on the signal sparsity.

All SFFT algorithms beat FFTW when the sparsity-size quotient $k/n$ is small enough. Especially Version 3 is almost always better, which was expected because of its small asymptotic runtime cost.

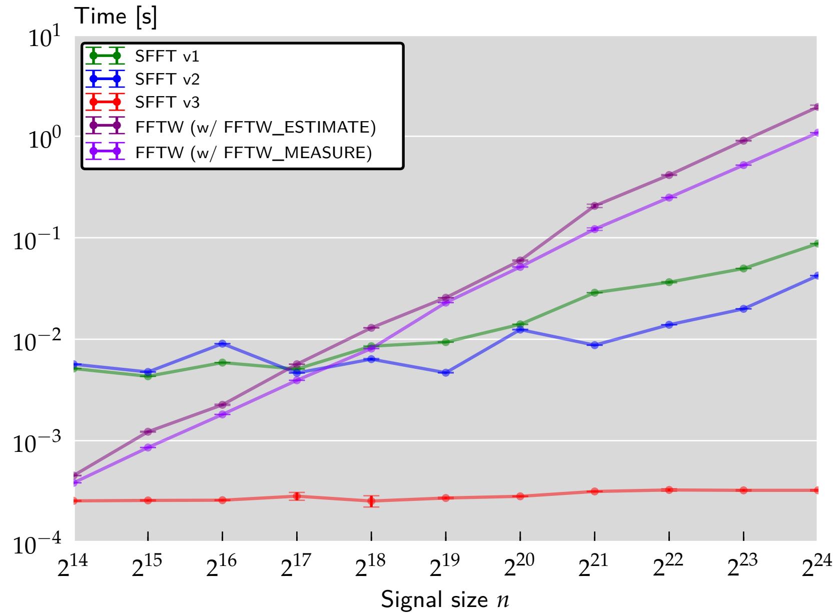



Figure 3.1: Runtime of different non-optimized SFFT versions versus signal size $n$ ($k = 50$).

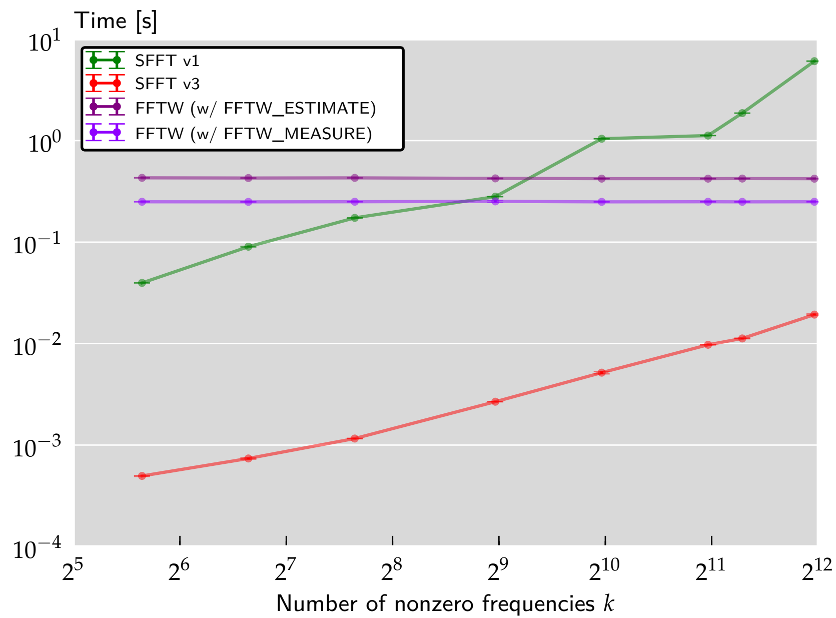



Figure 3.2: Runtime of different non-optimized SFFT versions versus signal sparsity $k$ ($n = 2^{22}$).

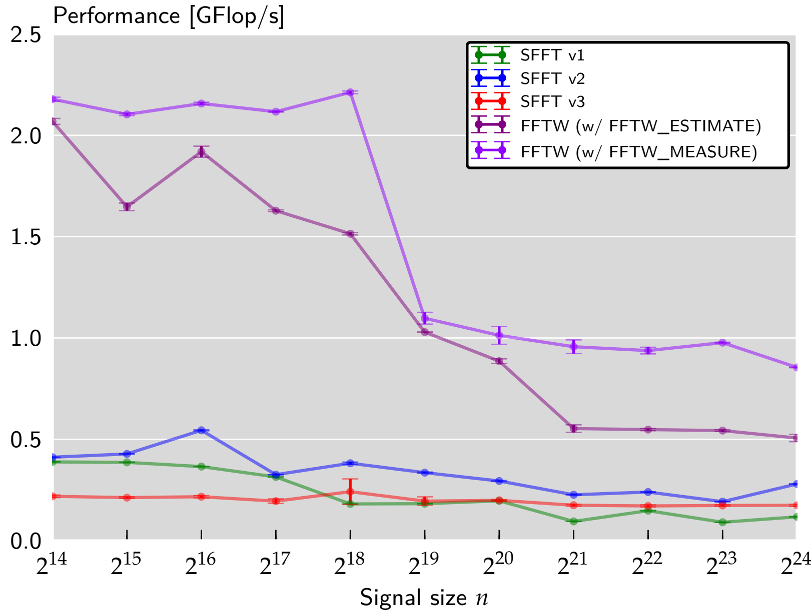



Figure 3.3: Performance of SFFT v1, v2 and v3 (non-optimized, $k = 50$) and FFTW. Note that each algorithm has a different operations count, so the plot does not serve as runtime comparison but only as efficiency comparison.

Performance is a useful metric to evaluate the implementation of an algorithm on a particular system. In this thesis, performance is measured in measured in GFlop/s, since the SFFT mainly consists of floating point operations.

The amount of floating point operations is measured with an instrumentation in the sourcecode implemented with C macros. The instrumentation has to be activated explicitly with a configuration parameter at compile time, so it does not influence timing benchmarks. The instrumentation counts scalar arithmetic floating point operations. Simple operations (e.g. ADD, MULT) are counted as 1 Flop. Complex operations are counted as the amount of equivalent real operations, for example a complex multiplication involves 4 real multiplications, 1 real addition and 1 real subtraction. The operation count of standard library functions like `sin` were measured using performance counters on Intel CPUs using the PCM library. Operation counts for FFTW plans can be computed with FFTW's `fftw_flops` function.

Figure 3.3 shows performance measurements for the SFFT algorithms. The benchmark system's scalar single-core peak performance can be computed as

| *Function* | *Runtime [s]* | *% of Total Time* | *Performance [GFlop/s]* |
|---|---|---|---|
| Inner Loops | 1.18e-02 | 88.51 | 0.23 |
| Estimate Values | 1.53e-03 | 11.49 | 0.01 |

Table 3.2: Profile of an SFFT v1 run (non-optimized).

| *Function* | *Runtime [s]* | *% of Total Time* | *Performance [GFlop/s]* |
|---|---|---|---|
| Mansour Filter | 2.61e-03 | 22.67 | 0.76 |
| Inner Loops | 4.61e-03 | 40.27 | 0.35 |
| Estimate Values | 4.27e-03 | 37.07 | 0.01 |

Table 3.3: Profile of an SFFT v2 run (non-optimized).

$$\begin{aligned}\text{Peak Perf.} &= \underbrace{1\,\text{Flop/Instr.}}_{\text{Scalar Instructions}} \times \underbrace{2\,\text{Instr./Cycle}}_{\text{Instruction Level Parallelism}} \times \underbrace{2.2 \times 10^9\,\text{Cycle/s}}_{\text{CPU Frequency}} \\ &= 4.4\,\text{GFlop/s}.\end{aligned}$$

However, peak performance is hard to reach, for example when algorithms are memory-bound (i.e., the performance is limited by the memory bandwidth). The benchmark shows that there is room for improvement. For comparison, FFTW's performance was added to the benchmark, but note that the higher performance of FFTW does not imply smaller runtime – the instruction count of the algorithms is fundamentally different.

## 3.3 Profiling

Tables 3.2–3.4 show runtime profiles for the different non-optimized SFFT versions. Fixed input parameters $n = 2^{20}$ and $k = 50$ were chosen.

The profiles show that the algorithms' most costly parts are filter applications. In SFFT v1 and v2 filter applications are implemented in the *Inner Loops* part; in SFFT v2 an additional *Mansour Filter* is applied. SFFT v3 implements a Mansour Filter, a Gaussian Filter without spectrum permutation, and a Gaussian Filter with spectrum permutation. In all 3 SFFT versions at least 60 % of the runtime consist of filter applications.

Comparison between SFFT v1 and SFFT v2 proves the Mansour Filter heuristic successful (while maintaining about the same accuracy, refer to the robustness-to-noise experiment in [HIKP12b]). The inner loop runtime and

| *Function* | *Runtime [s]* | *% of Total Time* | *Performance [GFlop/s]* |
|---|---:|---:|---:|
| Mansour Filter | 7.71e-05 | 19.81 | 0.20 |
| Estimate Frequencies | 7.33e-05 | 18.83 | 0.12 |
| Gauss Filter | 1.01e-04 | 26.01 | 0.14 |
| Estimate Frequencies | 2.46e-05 | 6.34 | 0.16 |
| Permuted Gauss Filter | 6.72e-05 | 17.25 | 0.14 |
| Estimate Frequencies | 5.87e-06 | 1.51 | 0.13 |
| Loop Between Filters | 3.99e-05 | 10.25 | 0.15 |

Table 3.4: Profile of an SFFT v3 run (non-optimized).

the overall runtime could be reduced significantly; the estimation phase duration, however, increased.

Using the *callgrind* profiling tool it was also checked how much time was spent in the internal FFTW calls for the same input parameters ($n = 2^{20}$ and $k = 50$). For SFFT v1, about 3.2 % of the runtime was inside FFTW calls; for SFFT v2 it was 13.8 %, and for SFFT v3 about 1.9 % of the runtime was inside FFTW calls. This means that the internal DFT computations are not the main bottlenecks of the implementation.

## 3.4 Roofline Analysis

Roofline Analysis is a novel method for investigating an algorithm's performance on particular hardware (see [WWP09]). The method's core idea is to relate the algorithm's operational intensity, measured in Flop/Byte, to its performance, measured in Flop/s or Flop/Cycle. Operational intensity is defined as follows:

**Definition 3** (Operational Intensity)**.** *The operational intensity $I$ of an algorithm $A_i$ with input $i$ on a particular system is defined as the ratio*

$$I = \frac{W}{M},$$

*where $W$ is the operations count of $A_i$, and $M$ is the memory traffic between the system's last-level cache and the system's main memory.*

Here, the work $W$ is measured as the number of arithmetic floating point operations (additions, multiplications, but also comparisons), as in the SFFT mainly floating point operation are performed. Other applications could define work differently.

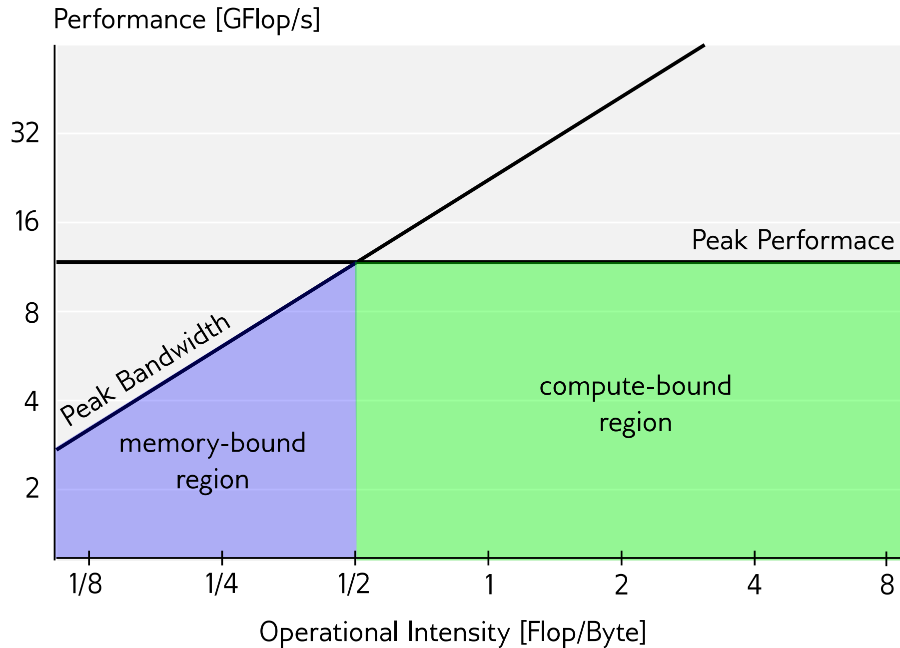



Figure 3.4: An exemplary roofline plot.

In Roofline Analysis, algorithm performance is plotted against operational intensity in a log-log plot. Two additional system-specific performance ceilings are inserted to the plot. The first ceiling is the system's maximum peak performance, i.e. the maximum amount of work that can be performed per unit of time. The second ceiling is a memory bandwidth limit. That is, for an algorithm with operational intensity $I$ Flop/Byte and a system with a maximum bandwidth of $B$ Byte/$s$, the memory induced performance limit is $I \cdot B$ Flop/s. Thus, when $P_{max}$ is the system's peak performance, algorithm performance $P$ is bound by

$$P \leq \max\{I \cdot B, P_{max}\}.$$

Roofline Analysis allows to define exactly when an implementation is memory-bound, and when it is compute-bound. Implementations with an operational intensity below the crossing point of the two ceiling are memory-bound, implementations with a higher operational intensity are compute-bound (see Figure 3.4).

Computing the necessary quantities for Roofline Analysis, memory traffic, work, and runtime, can be easily done for simple algorithms. For complex algorithms like the Sparse Fast Fourier Transform, however, the task is more involved. While work and runtime can be easily determined

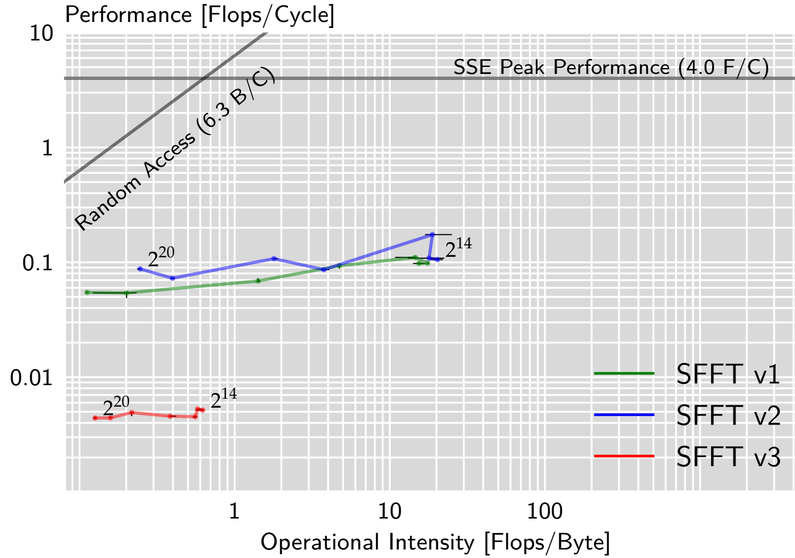



Figure 3.5: Roofline plots of SFFT v1, v2 and v3 (non-optimized, $n = 2^{14} \ldots 2^{20}$, $k = 50$).

(this has already been done for the performance measurements), it is not clear how to gather reliable memory traffic data. The tool *perfplot* [per13] has been developed for exactly this purpose. It accesses performance counters on recent Intel CPUs using Intel's PCM library, giving data for all necessary quantities. The advantage of this approach is that no knowledge about algorithm nor the underlying hardware is necessary; all data are simply measured.

Using *perfplot* I analyzed all SFFT versions. The measurements were performed on an Intel(R) Xeon(R) X5680 CPU clocked at 3.33 GHz with a 12 MB L3-cache. All measurements were cold-cache measurements, i.e., the memory traffic also contains compulsory cache misses.

Figure 3.5 shows a roofline plot of all SFFT versions. The analysis proves that there is room for optimizations.

For input sizes larger than $2^{18}$ (SFFT v1, v2) respectively larger than $2^{15}$ (SFFT v3) all versions are memory-bound, therefore optimizations targeting cache utilization should improve performance.

The performance measured by *perfplot* is slightly lower than the performance that was previously estimated. The reason for this are the different methods to obtain operations counts. The *perfplot* data are gathered from CPU performance counters, while the data from previous performance

measurements were counted within the library using an instrumentation framework.

# Chapter 4

# Performance Optimizations

In this chapter the various optimizations that were applied to the reference implementation are presented. The optimizations are grouped by the specific issue that they address, though sometimes an optimization can belong to more than one category. All optimizations were tested in the implementation, and sometimes micro-benchmarks were developed to compare different strategies.

## 4.1 Instruction Reduction

The first step in optimizing the SFFT algorithms was to establish a clear separation between a planning and an execution phase. This approach is similar to FFTW's approach. In the planning phase, every possible pre-computation is performed that can be reused across different DFT computations of the same input size (and is thus amortized). This involves the allocation of storage, generation of filter vectors and creation of FFTW plans. The execution phase contains every task that cannot be shared among multiple SFFT executions, like generating random numbers and execution of the actual algorithm. Once a plan is created it can be executed repeatedly with different inputs of the same size, so that the planning cost is only a one-time cost.

### *4.1.1 FFTW*

Internally, all SFFT algorithms perform several DFTs, for example in the SFFT v1 location and estimation loops, or in SFFT v3's *HashToBins* routine. In the original, non-optimized implementation FFTW is used for this purpose. FFTW is a high-performance library with a good performance, but by using some features of the library the performance of the FFTs can still be increased.

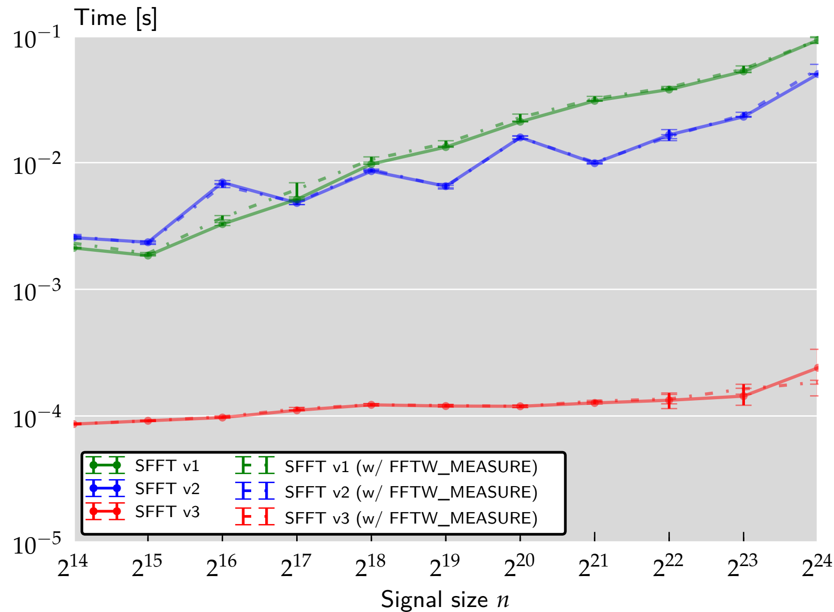



Figure 4.1: A comparison of the different FFTW options for the internal use in the SFFT algorithms.

The first optimization was already mentioned: FFTW plans can be pre-computed in the SFFT planning phase. Even though FFTW does reuse already computed data from previous plans to some extent, there is no need to create the same FFTW plans multiple times.

In the SFFT algorithms, often multiple equally sized FFTs are computed in a row, or the implementation can be adapted to do this. For example, the FFTs of all SFFT v1 location loops can be computed at once. Instead of computing each of the FFTs separately, FFTW supports an interface (`fftw_plan_many`) to compute all of the FFTs within a single call. This way, FFTW has more freedom to schedule operations and optimize the execution; for example, the FFTW twiddle factors only have to be computed once.

FFTW supports multiple optimization levels that can be configured by flags, the most important are `FFTW_ESTIMATE` and `FFTW_MEASURE`. For the SFFT, in most cases it is sufficient to use `FFTW_ESTIMATE`, because the FFTs computed are relatively small and the higher optimization level of `FFTW_MEASURE` only pays off for large input sizes.

Figure 4.1 shows an experiment investigating the impact of the different FFTW parameters in the SFFT. There is no difference at all between the usage of `FFTW_ESTIMATE` and `FFTW_MEASURE`. Thus, it is sufficient to use the faster `FFTW_ESTIMATE` in the SFFT planning phase.

A last optimization to the FFTW calls is to use in-place FFTs. After the DFT step in the filters, the time-domain of the computed vector is not used anymore. So an in-place FFT can be used to improve FFTW's performance and reduce SFFT's memory consumption. FFTW supports this natively by simply passing the same vector as input and output argument.

### 4.1.2 *Inlining and explicit complex arithmetic*

To reduce function call overhead, non-public functions can be inlined or declared static so that the compiler can effectively inline them. For example, a profile of an SFFT run showed many calls to the function `__muldc`, the C standard library's implementation of complex multiplication. The complex vectors were replaced with real vectors of twice the size. Complex numbers are stored now as tuples of real and imaginary part. This is byte-compatible to the C standard library's complex data type. This overhead was removed by replacing this function call with an own implementation of complex multiplication.

Before the optimization, inner location loops in SFFT v1 and v2 looked like this:

```
int index=b;
for(int i = 0; i < filter.sizet; i++)
  {
    x_sampt[i%B] += origx[index] * filter.time[i];
    index = (index+ai) %n;
  }
```

After the optimization, the code changed to:

```
for(int i = 0; i < 2*filter.sizet; i+=2)
  {
    double ac = d_orig_x[index] * d_filter[i];
    double bd = d_orig_x[index+1] * d_filter[i+1];
    double ad = d_orig_x[index] * d_filter[i+1];
    double bc = d_orig_x[index+1] * d_filter[i];
    d_x_sampt[i%(2*B)]    += ac-bd;
    d_x_sampt[i%(2*B)+ 1] += ad+bc;
    index = (index + 2*ai)%(2*n);
  }
```

### 4.1.3 *Fixed loop configurations*

The SFFT algorithms use many parameters controlling accuracy and run-time. This fact required the baseline implementation to be very generic: all possible parameter combinations have to be considered. A very successful

optimization is to fix some of the parameters and specialize the implementation to only support the fixed parameters. The generic implementation can be kept as a fallback.

This was done for the measurement loop counts in SFFT v3. Running two inner loops in each filter (Mansour Filter, Gaussian Filter and Permuted Gaussian Filter) is usually sufficient to reconstruct the signal coefficients with a high probability. The code of SFFT v3 was greatly simplified by specializing for fixed loop counts. Many functions became much simpler, like expensive median calculations, which could be removed entirely (the median of 2 values is trivial to compute).

For example, this is the routine for applying the Gaussian filter in SFFT v3 with a variable loop count:

```
int Gauss_Filt(sfft_v3_data* data, complex_t *origx, int n,
               complex_t *filter, int w, int B,
               complex_t *x_gauss, int init_G_offset)
{
  // [...]

  for(int j = 0; j < 2; j++)
  {
    for(int i = 0; i < 2*w; i+=2)
    {
      index = (2*init_G_offset+2*j+i) % (2*n);

      double ac = d_orig_x[index]   * d_filter[i];
      double bd = d_orig_x[index+1] * d_filter[i+1];
      double ad = d_orig_x[index]   * d_filter[i+1];
      double bc = d_orig_x[index+1] * d_filter[i];

      d_x_sampt[2*B*j + i%(2*B)]     += ac-bd;
      d_x_sampt[2*B*j + i%(2*B) + 1] += ad+bc;
    }
  }

  fftw_execute(tl_data->fftw_plan_gauss);

  return 0;
}
```

The loop over `j` was unrolled and removed. Then, it was easy to adjust the iteration scheme so that the filter is only traversed once; note that in each inner loop only a single filter value is loaded:

```
int Gauss_Filt_loops2(sfft_v3_data* data, complex_t *origx, int n,
               complex_t *filter, int w, int B,
               complex_t *x_gauss, int init_G_offset)
{
  // [...]
```

```
  double a1 = d_orig_x[2*init_G_offset];
  double b1 = d_orig_x[2*init_G_offset + 1];

  for(int i = 0; i < 2*w; i+=2)
  {
    double a0 = a1;
    double b0 = b1;

    double a1 = d_orig_x[(2*init_G_offset + i) % (2*n)];
    double b1 = d_orig_x[(2*init_G_offset + i + 1) % (2*n)];

    double c = d_filter[i];
    double d = d_filter[i + 1];

    double a0c = a0 * c;
    double b0d = b0 * d;
    double a0d = a0 * d;
    double b0c = b0 * c;

    double a1c = a1 * c;
    double b1d = b1 * d;
    double a1d = a1 * d;
    double b1c = b1 * c;

    d_x_sampt[i%(2*B)]            += a0c-b0d;
    d_x_sampt[i%(2*B) + 1]        += a0d+b0c;
    d_x_sampt[2*B + i%(2*B)]      += a1c-b1d;
    d_x_sampt[2*B + i%(2*B) + 1] += a1d+b1c;
  }

  fftw_execute(tl_data->fftw_plan_gauss);

  return 0;
}
```

### *4.1.4 Optimizing Individual Instructions*

There are some expensive operations that are very common for the SFFT and occur in inner loops of the algorithms. Especially some modulo calculations turned out to be very expensive. There are a few tricks to get rid of these.

Modulo operations like $x \mod n$ can be computed fast when $n$ is a power of 2. For example, when $n = 2^k$, then $x \mod n$ can then be computed by only keeping the lower $k$ bits of $x$ and setting all other bits to zero. This can be implemented by an `AND` instruction, which is typically very fast on modern CPUs.

Another trick to remove modulo operations entirely can be applied

when the result of the modulo operation is only used as an argument of a trigonometric function like sin or cos. The sin function is periodic, so $\sin(x) = \sin(x + t \cdot 2\pi)$ for all integers $t$. A typical computation in the SFFT is $\sin(2\pi/n \cdot (x \mod n))$. Here, the modulo operation can be removed:

$$\begin{aligned}\sin(2\pi/n \cdot (x \mod n)) &= \sin(2\pi/n \cdot (x \mod n) + \lfloor x/n \rfloor \cdot 2\pi) \\ &= \sin(2\pi \cdot ((x \mod n)/n + \lfloor x/n \rfloor)) \\ &= \sin(2\pi \cdot x/n)\end{aligned}$$

Computations similar to $\sin(\frac{2\pi}{n} \cdot (x \mod n))$ occur in various places of the SFFT algorithms, e.g., in SFFT v3's `update` and `estimate` functions. Since $2\pi/n$ is a constant that can be pre-computed, the expensive modulo computation can be replaced by a simple and fast multiplication.

Another trick is to avoid expensive modulo operations for array indices is to adjust loop structure and orders. This is discussed in 4.2.1.

## 4.2 Cache Usage Optimizations

### 4.2.1 Chunking

A simplified inner Gauss Filter loop in SFFT v1 can roughly be implemented as shown in the following code snippet:

```
for(unsigned j = 0; j < loops; j++)
  {
    int index=b_vec[j];
    for(int i = 0; i < 2*w; i+=2)
      {
        double ac = d_orig_x[index]   * d_filter[i];
        double bd = d_orig_x[index+1] * d_filter[i+1];
        double ad = d_orig_x[index]   * d_filter[i+1];
        double bc = d_orig_x[index+1] * d_filter[i];

        d_x_sampt[j][i % (2*B)]     += ac-bd;
        d_x_sampt[j][i % (2*B) + 1] += ad+bc;

        index = (index + 2*ai[j]) & n2_m_1;
      }
  }

  // Apply DFT, ...
```

Obviously the filter vector is traversed several times, once for each loop. Since the filter can be relatively big this might lead to unsatisfactory cache utilization. Changing the loop order would lead to a bad access pattern on `x_sampt`. The solution to this problem is to iterate in chunks:

```
for(unsigned chunk = 0; chunk < chunks; chunk++)
  {
    unsigned start = chunk*chunksize;
    unsigned end = std::min((chunk+1)*chunksize, 2*w);

    for(unsigned j = 0; j < loops; j++)
      {
        int index=b_vec[j];
        for(int i = start; i < end; i+=2)
          {
            double ac = d_orig_x[index]   * d_filter[i];
            double bd = d_orig_x[index+1] * d_filter[i+1];
            double ad = d_orig_x[index]   * d_filter[i+1];
            double bc = d_orig_x[index+1] * d_filter[i];

            d_x_sampt[j][i % (2*B)]     += ac-bd;
            d_x_sampt[j][i % (2*B) + 1] += ad+bc;

            index = (index + 2*ai[j]) & n2_m_1;
          }
      }
  }
```

The parameter chunksize can be freely chosen, e.g., to match the CPU's cache size appropriately, but in experiments it was shown that the fastest implementation can be done with the chunksize $B$ (the size of the subsampled vector), so that the expensive index computation can be simplified:

```
// ...
chunksize = B;

for(unsigned chunk = 0; chunk < chunks; chunk++)
  {
    unsigned start = chunk*chunksize;
    unsigned end = std::min((chunk+1)*chunksize, w);

    for(int j = 0; j < loops; j++)
      {
        int index=b_vec[j];
        i2_mod_B = 0;
        for(int i = 0; i < 2*filter.sizet; i+=2)
          {
            double ac = d_orig_x[index]   * d_filter[i];
            double bd = d_orig_x[index+1] * d_filter[i+1];
            double ad = d_orig_x[index]   * d_filter[i+1];
            double bc = d_orig_x[index+1] * d_filter[i];

            d_x_sampt[j][i2_mod_B]     += ac-bd;
            d_x_sampt[j][i2_mod_B + 1] += ad+bc;
```

```
            index = (index + 2*ai[j]) & n2_m_1;
            i2_mod_B += 2;
          }
      }
  }
```

The implementation runs at the highest performance when using an iteration scheme similar to the one above. Changing the loop order (making the *j*-loop the innermost loop) showed no effect. When the loop count is fixed (which is usually the case for SFFT v3), it is often a good idea to unroll the *j*-loop and load the filter value only once.

### *4.2.2 Data Layout*

The original SFFT implementations contained many complex data structures like nested arrays. Nested data structures create an additional level indirection, and this can be deficient for cache utilization. Instead, flat arrays, where all data is stored sequentially, are much simpler and can yield higher locality when accessing them, therefore leading to better cache usage. An additional benefit is that some computations can become easier. For example, traversing all elements in order can be implemented by simply iterating over the flat array, without the need of any index computation.

Changing the data layout was also a necessary pre-step to take advantage of some of FFTW's features; external libraries like FFTW often expect the data layout to be a continuous array.

Before the optimization, inner location loops in SFFT v1 and v2 were implemented like this:

```
for(unsigned chunk = 0; chunk < chunks; chunk++)
  {
    unsigned start = chunk*chunksize;
    unsigned end = std::min((chunk+1)*chunksize, w);

    for(int j = 0; j < loops; j++)
      {
        int index=b_vec[j];
        i2_mod_B = 0;
        for(int i = 0; i < 2*filter.sizet; i+=2)
          {
            double ac = d_orig_x[index]   * d_filter[i];
            double bd = d_orig_x[index+1] * d_filter[i+1];
            double ad = d_orig_x[index]   * d_filter[i+1];
            double bc = d_orig_x[index+1] * d_filter[i];

            d_x_sampt[j][i2_mod_B]     += ac-bd;
            d_x_sampt[j][i2_mod_B + 1] += ad+bc;

            index = (index + 2*ai[j]) & n2_m_1;
```

```
            i2_mod_B += 2;
          }
      }
  }
```

The optimized code looks like this:

```
double* d_x_sampt = (double*)x_sampt;
double* d_orig_x = (double*)origx;

/* Permutation and filter application */
int offset = 0;
for(int j = 0; j < loops; j++)
  {
    int index=b[j];
    int i2_mod_B = 0;
    for(int i = 0; i < 2*filter.sizet; i+=2)
      {
        double ac = d_orig_x[index]   * d_filter[i];
        double bd = d_orig_x[index+1] * d_filter[i+1];
        double ad = d_orig_x[index]   * d_filter[i+1];
        double bc = d_orig_x[index+1] * d_filter[i];

        d_x_sampt[i_mod_B+offset]     += ac-bd;
        d_x_sampt[i_mod_B+offset + 1] += ad+bc;

        index = (index + 2*ai[j]) & n2_m_1;
        i2_mod_B += 2;
      }
    offset += 2*B;
  }
```

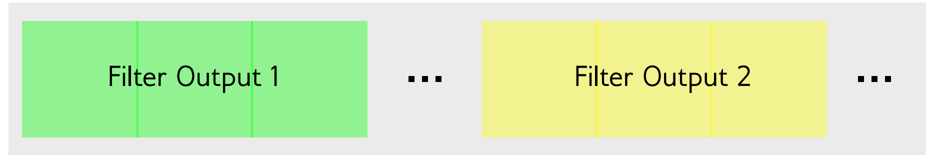
#### *4.2.3 Stride-2 FFTs*

It was already discussed that it is advantageous to store the outputs of measurements in a continuous array. The first, straight-forward approach was to store them sequentially, one after the other, i.e., the $j$-th location loop output is stored at $x_sampt[j \cdot B \ldots (j+1) \cdot B - 1]$. Alternatively, the outputs could also be stored interleaved. Figure 4.2 shows an illustration of both approaches.

This storage scheme increases locality in the inner loops of the measurements. In fact experiments showed an increase in performance when this other storage scheme was applied. To make this work, the FFTW plans have to be adapted. The FFTW interface supports this storage format directly via a stride parameter:

```
fftw_plan
fftw_plan_many_dft(int rank, const int *n, int howmany,
                   fftw_complex *in, const int *inembed,
```

Original Data Layout




Interleaved Data Layout

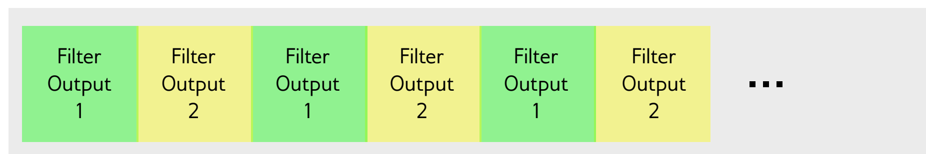



Figure 4.2: An illustration of the different storage formats.

```
                int istride, int idist,
                fftw_complex *out, const int *onembed,
                int ostride, int odist,
                int sign, unsigned flags);
```

## 4.3 Vectorization

### *4.3.1 SSE Support and Memory Alignment*

Vectorization allows to speed up simple data parallel tasks by allowing CPU instructions to work on multiple data at once (the so-called SIMD approach: single instruction, multiple data).

To vectorize the the SFFT library I used the SSE2 instruction set, which is available on many modern x86 CPUs. More precisely, I used compiler intrinsics of the GNU and Intel compilers to explicitly emit specific instructions.

For successful vectorization it is necessary to work with aligned memory. Since input vectors are allocated by the user, the user has to take care of this. To ensure proper alignment, the method `sfft_malloc` was implemented as a replacement for `malloc`.

### 4.3.2 SSE Implementations of Compute Intensive Functions

The profiling in chapter 3 showed that the filter applications in all algorithm versions are very expensive. The measurement routines of all SFFT algorithms, i.e., the inner loops in SFFT v1 and v2 as well as the Mansour-, Gauss- and Permuted Gauss-Filters in SFFT v3 can be vectorized in a straightforward way. These routines mainly consist of complex multiplications and additions.

There are two kinds of approaches for SSE-based vectorization of double precision floating point operations, where exactly two values fit into a vector. The first one is to treat one 128 Byte vector as a single complex number. The second approach is to unpack the complex data into vectors of real and imaginary data. The first approach is slightly simpler, but the second approach showed a better performance. Plus, the second approach can be extended to 4-way vectorization with AVX or other vectorization technologies with even bigger vectors. The following code example shows a vectorized SFFT v3 filter (other previously discussed optimizations are included as well):

```
int Gauss_Filt_loops2(sfft_v3_data* data, complex_t *origx, int n,
                      complex_t *filter, int w, int B,
                      complex_t *x_gauss, int init_G_offset)
{
  // [...]

  const unsigned n2_m_1 = 2*n - 1;
  const unsigned origx_offset = (2*init_G_offset+2) & n2_m_1;

  const unsigned chunksize = 2*B;
  const unsigned chunks = 2*w/chunksize;

  for(unsigned chunk = 0; chunk < chunks; chunk++)
  {
    unsigned start = chunk*chunksize;
    unsigned end = std::min((chunk+1)*chunksize, (unsigned)2*w);

    __m128d a2b2 = _mm_load_pd(d_origx+((2*init_G_offset+start)&
        n2_m_1));
    unsigned i2_mod_B = 0;
    for(unsigned i = start; i < end; i+=2)
    {
      __m128d ab = a2b2;
      a2b2 = _mm_load_pd(d_origx+((origx_offset+i)&n2_m_1));
      __m128d cd = _mm_load_pd(d_filter+i);

      __m128d cc = _mm_unpacklo_pd(cd, cd);
      __m128d dd = _mm_unpackhi_pd(cd, cd);
```

```
      __m128d a0a1 = _mm_unpacklo_pd(ab, a2b2);
      __m128d b0b1 = _mm_unpackhi_pd(ab, a2b2);

      __m128d ac = _mm_mul_pd(cc, a0a1);
      __m128d ad = _mm_mul_pd(dd, a0a1);
      __m128d bc = _mm_mul_pd(cc, b0b1);
      __m128d bd = _mm_mul_pd(dd, b0b1);

      __m128d ac_m_bd = _mm_sub_pd(ac, bd);
      __m128d ad_p_bc = _mm_add_pd(ad, bc);

      __m128d ab_times_cd = _mm_unpacklo_pd(ac_m_bd, ad_p_bc);
      __m128d a2b2_times_cd = _mm_unpackhi_pd(ac_m_bd, ad_p_bc);

      __m128d xy = _mm_load_pd(d_x_sampt+i2_mod_B);
      __m128d x2y2 = _mm_load_pd(d_x_sampt+i2_mod_B+2);

      __m128d st = _mm_add_pd(xy, ab_times_cd);
      __m128d s2t2 = _mm_add_pd(x2y2, a2b2_times_cd);

      _mm_store_pd(d_x_sampt+i2_mod_B, st);
      _mm_store_pd(d_x_sampt+i2_mod_B+2, s2t2);

      i2_mod_B += 4;
    }
  }

  fftw_execute(tl_data->fftw_plan_gauss);

  return 0;
}
```

### *4.3.3 More Vectorization*

Other functions were also vectorized, but it was not always possible in a straightforward way. A typical code construct that blocks vectorization are branches. For example, SFFT v3's `estimate` functions consist of loops with several branches. Some important checks are performed in these branches, like a check if the current buck is empty or a check if a hash collision occurred, and it is not possible to skip them. Sometimes techniques like masking can help to vectorize such code anyway, but it was not possible in this case. Masking works by first ignoring the branches and performing the relevant computations on all data. Later, some of the data that are not desired can be filtered out. This only works when relatively few data are filtered out, so that the overhead of the additional computations is negligible. Unfortunately this is not the case in the functions described above.

Other routines, like the `update` functions in SFFT v3, can be vectorized, but have complex memory access patterns. The performance of these functions is mainly dependent on the caching strategy and memory bandwidth of the CPU. Therefore SSE vectorization does not always show the expected 2x speedups.

This is a version of SFFT v3's `estimate_freq_gauss` function, where only little parts could be vectorized:

```
static int estimate_freq_mansour_loops2(sfft_v3_data* data,
          int BUCKETS, complex_t *SAMP,
          int init_offset, int LOOPS, int n, int a, int b, int
              jump,
          complex_t *filterf, int *EST_FREQS, complex_t *
              EST_VALUES)
{
  // [...]

  double zero_buck_check[2];

  __m128d norm2vec = _mm_set1_pd(NORM2);

  for(int i = 0; i < BUCKETS; i+=2)
    {
      __m128d a0b0 = _mm_load_pd(d_SAMP+4*i);
      __m128d a1b1 = _mm_load_pd(d_SAMP+4*i+2);
      __m128d a2b2 = _mm_load_pd(d_SAMP+4*i+4);
      __m128d a3b3 = _mm_load_pd(d_SAMP+4*i+6);

      __m128d a0b0_sq = _mm_mul_pd(a0b0, a0b0);
      __m128d a1b1_sq = _mm_mul_pd(a1b1, a1b1);
      __m128d a2b2_sq = _mm_mul_pd(a2b2, a2b2);
      __m128d a3b3_sq = _mm_mul_pd(a3b3, a3b3);

      __m128d c0c1 = _mm_hadd_pd(a0b0_sq, a1b1_sq);
      __m128d c0c1_normed = _mm_mul_pd(c0c1, norm2vec);
      __m128d c2c3 = _mm_hadd_pd(a2b2_sq, a3b3_sq);
      __m128d c2c3_normed = _mm_mul_pd(c2c3, norm2vec);

      __m128d zbc = _mm_hadd_pd(c0c1_normed, c2c3_normed);

      _mm_store_pd(zero_buck_check, zbc);

      for(unsigned j = 0; j < 2; j++)
        {
          if(zero_buck_check[j] > ZERO_BUCK_CHECK_CUTOFF)
            {
              real_t a0 = d_SAMP[4*i+4*j];
              real_t b0 = d_SAMP[4*i+4*j+1];
              real_t a1 = d_SAMP[4*i+4*j+2];
              real_t b1 = d_SAMP[4*i+4*j+3];
```

```
            real_t c0 = (a0*a0+b0*b0)*NORM2;
            real_t c1 = (a1*a1+b1*b1)*NORM2;
            real_t atan_real[] = { a0*NORM, a1*NORM };
            real_t atan_imag[] = { b0*NORM, b1*NORM };
            real_t atan_result[2];
            approx_atan2_vec2(atan_imag, atan_real, atan_result)
                ;
            real_t d0 = (real_t)atan_result[0];
            real_t d1 = (real_t)atan_result[1];
            real_t collision_threshold = 1e-10, median_phase=0;
            double slope=0;
            real_t median_abs = c0;
            real_t median_abs_inv = 1./median_abs;

            real_t b = (c1*median_abs_inv) - 1;
            real_t error = b*b;

            if((error < n*collision_threshold) && (median_abs >
                0.01))
              {
                slope = d1-d0;
                freq1 = lrint(slope * N_OVER_PI2);
                freq2 = freq1 & FREQ_MASK;
                freq3 = freq2 | (i+j);
                int freq_offset = freq3 * init_offset;
                median_phase = d0 - PI2_OVER_N * freq_offset;

                approx_sqrt(&median_abs);
                real_t median_phase_cos;
                real_t median_phase_sin;
                approx_sincos (&median_phase, &median_phase_sin,
                     &median_phase_cos);
                real_t median_value_real = median_abs *
                    median_phase_cos;
                real_t median_value_imag = median_abs *
                    median_phase_sin;

                EST_FREQS[found] = freq3;
                d_EST_VALUES[2*found] = median_value_real;
                d_EST_VALUES[2*found+1] = median_value_imag;
                found++;
              }
          }
      }
    }

  return found;
}
```

## 4.4 Multithreading

### 4.4.1 *Parallelizing Filters using OpenMP*

The first approach towards multi-threaded SFFT algorithms was to exploit data-parallelism in the compute-intensive parts, e.g., in the filter applications in all SFFT versions (which were shown to be very compute-intensive). OpenMP was used to distribute the loop iterations of these measurements onto the different cores of the CPU. Unfortunately, this simple approach did not work: no speedup was achieved. The overhead of multithreading here is too big compared to the relatively small loops, even though GCC's OpenMP implementation is implemented using thread pools to avoid frequent spawning of new threads. This could be verified with Intel's VTune performance analysis tool.

Parallelizing FFTW calls does not work, either. FFTW supports parallel FFT algorithms, but this only pays off with relatively long vectors.

Another possible explanation for the performance drop is that in the given scenario all threads are writing to one relatively short output vector. This can lead to *false sharing*, i.e., individual core caches are frequently invalidated due to writes by other cores. This can be resolved by giving each thread its own output vector and combining the individual results later. However, the combination step generates an overhead, and the performance was still below the single core performance.

Other compute-intensive functions like the *update* and *estimate* steps in SFFT v3 need careful synchronization in several parts. Each synchronization approach comes with an overhead, and unfortunately none of the parallelization attempts increased performance.

Since exploiting simple data-parallelism does not seem to work, task-based parallelism could be a way to distribute work among the processor cores. Due to the dependencies between the individual tasks in the algorithms this approach is limited. Experiments showed that this approach is still not successful.

### 4.4.2 *Coarse Multithreading*

Since no approach to parallelize a single SFFT call work, the question arises if there are other ways to benefit from modern multi-core architectures. A simple way to use parallelism is to run several SFFTs in parallel with different input data.

The trivial way of implementing this would be to allocate memory for $t$ SFFT calls (when $t$ is the number of threads) and then call the algorithm parallel in $t$ threads.

This is obviously not an elegant way since a lot of memory is allocated and some computations are performed multiple times. A smarter implementation should share as much data as possible among the threads (but avoid problems like false sharing). For example, the Gauss filter only need to be computed and stored once. This is especially useful on architectures with a shared cache, since the shared vectors only have to be transferred once (assuming a big enough cache).

Experiments shows that this approach with shared data slightly better than simple parallelism without any data sharing (see experiment in next chapter).

## 4.5 Miscellaneous Optimizations

### *4.5.1 Compilers and Compiler Options*

The cheapest optimization is to tell the compiler to do it. Initially, GCC was used to compile the SFFT library. Additionally, I ported the library to the Intel compiler suite. On a Core 2 Duo test machine, it showed slightly better performance. Compiler options used were:

- `-O3` to enable full optimization in gcc and icc (was only `-O2` initially).
- `-march=native` to use all available architecture features. For example, this is important to use SSE-optimized standard library functions.
- `-ffast-math` to allow the compiler to ignore IEEE 754 standard compliance. This is not needed in the SFFT algorithms, and by enabling this option the compiler can do additional optimizations, e.g., reorder floating point operations.

Compiler versions used were GCC 4.4.7 and ICC 13.

### *4.5.2 High-Performance Trigonometric Functions and Intel IPP*

In many places in the algorithms compute-intensive trigonometric functions like sin, cos or atan2 are called. For such functions, Intel offers some very optimized, vectorized methods in the IPP library. The approximation accuracy is also adjustable, and it turned out that the lowest approximation level is sufficient for the SFFT algorithms.

Often in the algorithm one of the trigonometric functions has to be computed on multiple values at once. In this scenario the IPP routines show even higher performance, since vectorized code can be used.

IPP also offers a function sincos to compute both sine and cosine at once, which is useful in various places of the algorithm.

#### 4.5.3 Result Storage Data structure

The return type of an SFFT call is an associative array mapping the found frequencies to the corresponding coefficients. This supports the sparse nature of the result very well. The original implementation used `std::map` from the C++ standard template library as data type. `std::map` is implemented as binary tree, and therefore has an access time of $\mathcal{O}(\log k)$ (when $k$ is the number of elements in the tree). Binary trees are useful when the data has to be traversed in a specific, sorted order. But this is not the case in the SFFT. Therefore, a real hash map with $\mathcal{O}(1)$ access time is a much better choice. Using `std::unordered_map`, the performance of the algorithm could be increased significantly. Other special high-performance data structures like Google's `sparse_hash` [Sil] did not increase the performance further (and even decreased the performance slightly), so `std::unordered_map` was kept as data structure.

# Chapter 5

# Results

In the previous chapter various performance optimizations were described. In this chapter the performance of the optimized SFFT library will be evaluated and compared to the analysis results of chapter 3.

## 5.1 Runtime Benchmarks

Figure 5.1 shows the results of a runtime benchmark of the optimized SFFT implementations and FFTW. Compared to Figure 3.1, the runtime of the SFFT versions has reduced significantly. The speedup plot in Figure 5.2 illustrates the difference more clearly. Especially SFFT Version 3 has improved, as it often shows a speedup above 5. Versions 1 and 2 have also improved, though the achieved speedup is smaller (it is usually in the range 1.5–3).

The speedup decreases for larger $n$. For larger $n$ the memory consumption increases and thus the CPU's cache sizes become a more and more significant factor. Thus, the optimizations are less successful for large $n$ and this explains the decrease.

SFFT v1 now beats FFTW for $n > 2^{16}$, while in the original implementation a signal size of $n > 2^{17}$ was necessary. Thus, the minimum sparsity-signal ratio for which SFFT v1 becomes useful has doubled.

A benchmark measuring algorithm runtime against signal sparsity $k$ is shown in Figure 5.3. Like the other benchmarks, this one has also improved compared to the results of chapter 3 (Figure 3.2).

## 5.2 Performance

The performance benchmarks of chapter 3 (Figure 3.3) were repeated on the same hardware. The results are shown in Figure 5.4. The performance

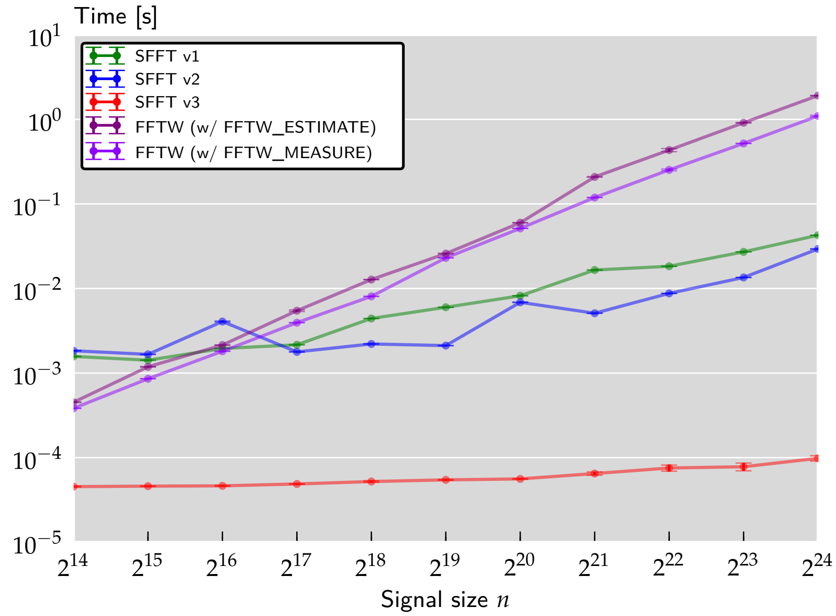



Figure 5.1: Runtime of different non-optimized SFFT versions versus signal size $n$ ($k = 50$).

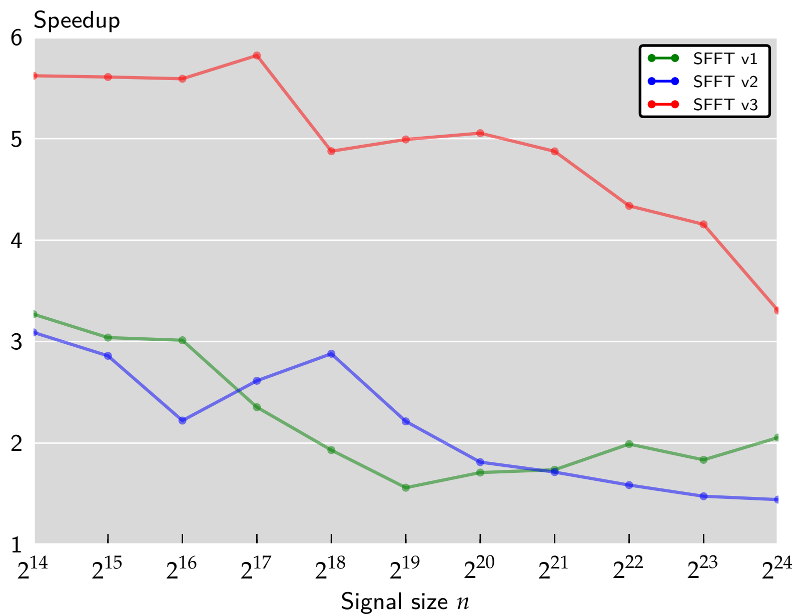



Figure 5.2: Speedup of the optimized SFFT implementation compared to the reference implementation ($k = 50$).

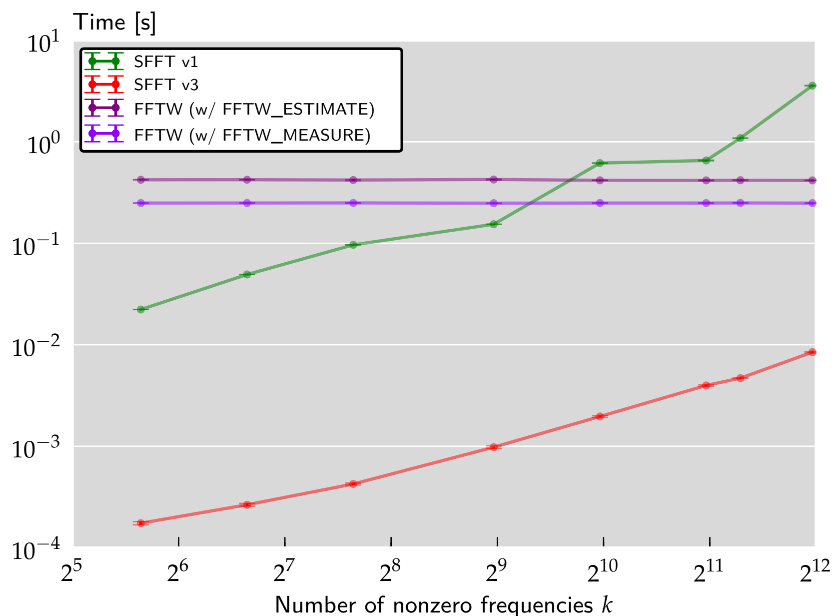



Figure 5.3: Runtime benchmark with varying signal sparsity $k$ ($n = 2^{22}$).

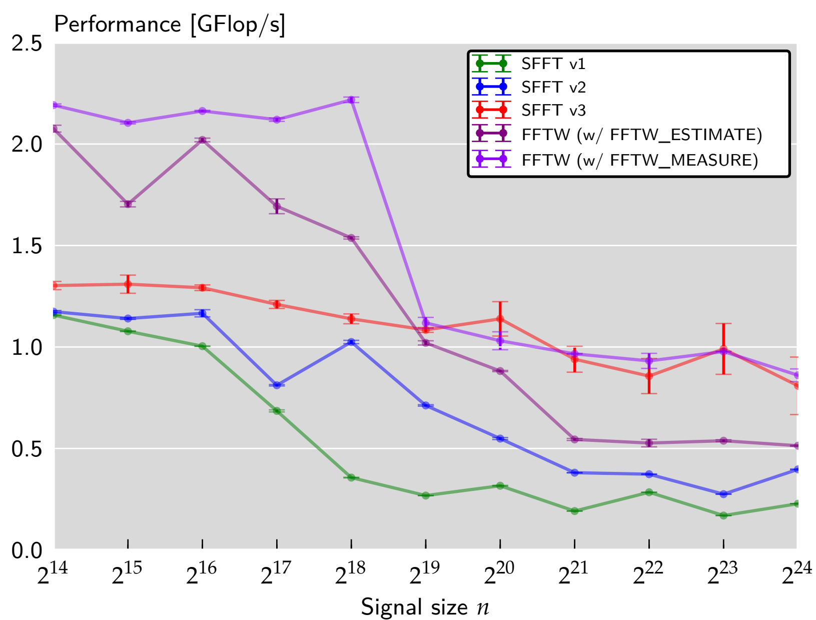



Figure 5.4: Performance of optimized SFFT v1, v2 and v3 ($k = 50$).

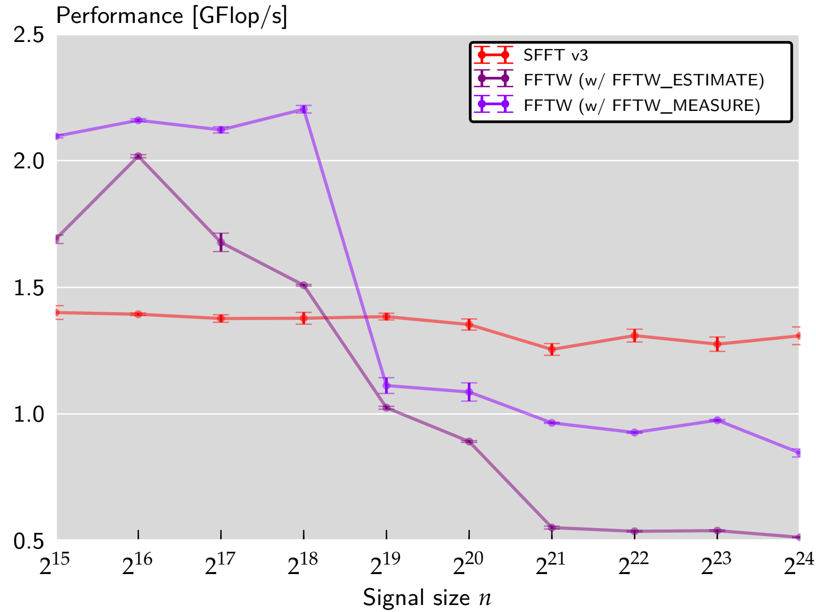



Figure 5.5: Performance of SFFT v3 with $k = 2000$.

of the algorithms has improved significantly. In the particular scenario with $k = 50$ and input sizes larger than $n = 2^{19}$, SFFT v3's performance is competitive to FFTW's performance. SFFT v1 and v2 performance has also improved, though the improvement is slightly smaller compared to SFFT v3. SFFT v3's performance is even higher for a larger number of nonzero Fourier coefficients, e.g., for $k = 2000$ as shown in Figure 5.5.

Note that performance comparison of different algorithms is difficult. An $n$-dimensional SFFT calls much lower-dimensional FFTs, where FFTW's performance is much better than FFTW's performance with an $n$-dimensional input vector. The performance measurements here are intended as efficiency comparisons.

## 5.3 Cold-Cache Benchmarks

The benchmarks shown so far are warm-cache measurements, i.e., it is assumed that much of the data is already present in the cache. This makes sense for applications where the SFFT is only one of many steps. For comparison, Figure 5.6 shows cold-cache performance measurements, and Figure 5.7 shows the speedup of the optimized SFFT implementation in a cold-cache scenario. The cold-cache performance of the SFFT algorithms is lower, especially for SFFT v3.

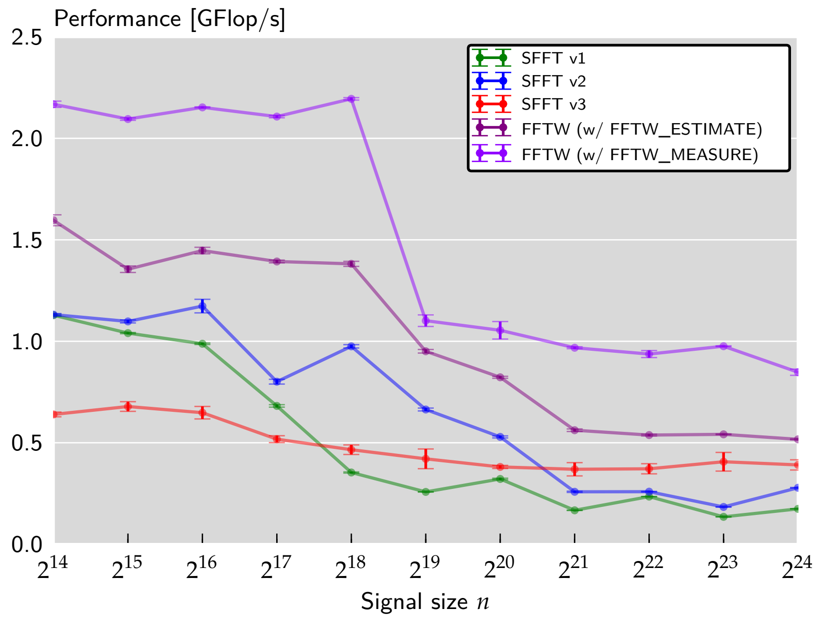



Figure 5.6: Cold-cache performance of SFFT v1, v2, v3 and FFTW ($k = 50$).

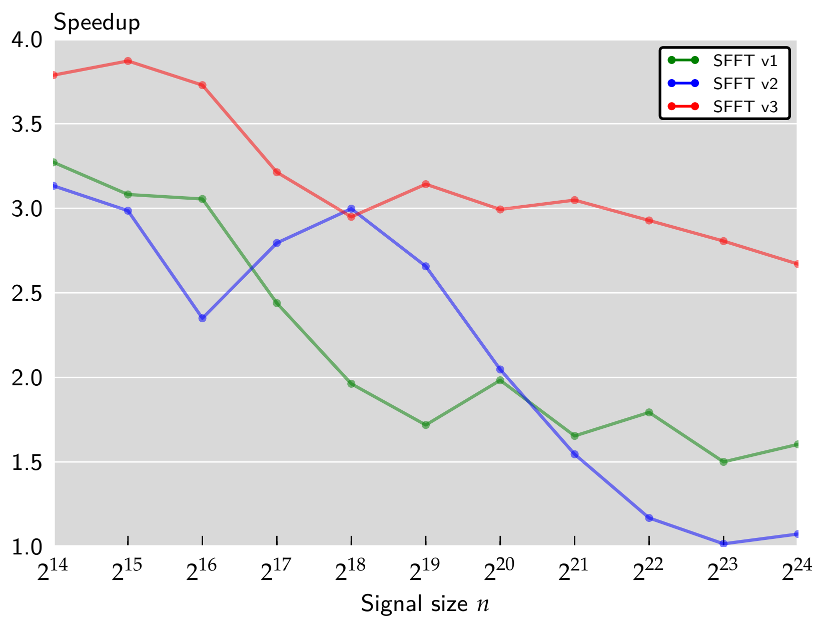



Figure 5.7: Speedup of the optimized SFFT library compared to the reference implementation (cold-cache measurements).

| *Function* | *Runtime [s]* | *% of Total Time* | *Performance [GFlop/s]* |
|---|---|---|---|
| Inner Loops | 6.16e-03 | 86.21 | 0.41 |
| Estimate Values | 9.74e-04 | 13.64 | 0.10 |

Table 5.1: Profile of an SFFT v1 run (optimized).

| *Function* | *Runtime [s]* | *% of Total Time* | *Performance [GFlop/s]* |
|---|---|---|---|
| Mansour Filter | 2.58e-03 | 50.08 | 0.82 |
| Inner Loops | 2.04e-03 | 39.56 | 0.54 |
| Estimate Values | 5.27e-04 | 10.22 | 0.10 |

Table 5.2: Profile of an SFFT v2 run (optimized).

## 5.4 Profiling

Comparing algorithm profiles before and after optimization shows some interesting characteristics of the algorithms and optimizations. This kind of analysis gives insight into which optimizations have been successful and which parts of the algorithms could be optimized well. The profiles are based on cold-cache measurements.

In SFFT v1 (Table 5.1), the runtime distribution has only changed slightly. The *inner loops* part is still dominating. The runtime of each part has decreased, and thus, the performance of each part has increased. The performance of the *estimate values* routine has increased by a whole order of magnitude (0.1 GFlop/s vs. 0.012 GFlop/s). This can only partly be explained by the decreased runtime; the algorithm also performs more floating point operations in the optimized variant.

The SFFT v2 profile (Table 5.2) changed in a similar way. There is one major difference to SFFT v1, though. The *Mansour Filter* part, which is an addition in SFFT v2 and not present in SFFT v1, did not improve much. This is because there is not much to improve in the Mansour Filter. The filter performs the following operations:

- It copies data while traversing a vector in large steps. Because of the large step size this part of the algorithm yields a bad spatial locality.
- It performs a DFT on the copied data, which is implemented as a call to FFTW. Besides some minor improvements on the FFTW call (in-place FFT, pre-planning), there is no room for additional optimizations.

| *Function* | *Runtime [s]* | *% of Total Time* | *Performance [GFlop/s]* |
|---|---|---|---|
| Mansour Filter | 1.45e-05 | 9.67 | 1.04 |
| Estimate Frequencies | 7.36e-05 | 48.97 | 0.10 |
| Gaussian Filter | 7.39e-06 | 4.91 | 1.44 |
| Update Gaussian | 9.77e-06 | 6.50 | 0.50 |
| Estimate Frequencies | 4.05e-06 | 2.69 | 0.54 |
| Update Mansour and Gaussian | 3.10e-06 | 2.06 | 0.50 |
| Permuted Gaussian Filter | 4.53e-06 | 3.01 | 0.96 |
| Update Gaussian | 1.03e-05 | 6.81 | 0.56 |
| Estimate Frequencies | 5.72e-06 | 3.80 | 0.16 |
| Update All | 1.91e-06 | 1.27 | 0.42 |
| Loop Between Filter | 1.55e-05 | 10.30 | 0.61 |

Table 5.3: Profile of an SFFT v3 run (optimized).

Thus, it is no surprise that the Mansour Filter takes about 50 % of the runtime in the optimized SFFT v2 call. Because the other functions could be optimized well, the Mansour Filter has become the most time-consuming part of the algorithm.

The profile of the optimized SFFT v3 is shown in Table 5.3. Compared to the original profile from chapter 3 (Table 3.4), especially the filters have improved. In the original profile the filters made up more than 63 % of the runtime (Mansour Filter 19.81 %, Gauss Filter 26 %, Permuted Gauss Filter 17.25 %). In the optimized version the filters sum up to less than 18 % of the runtime (Mansour Filter 9.67 %, Gauss Filter 4.91 %, Permuted Gauss Filter 3.01 %). Thus, all filters, especially the Gaussian Filters, could be improved significantly.

The Gaussian Filters in SFFT v3 are well suited for vectorization, since mainly arithmetic operations on vectors are performed. This, in combination with an improved data layout, a different iteration scheme and other optimizations (see chapter 4 for details), lead to a well performing implementation of these filters.

It may be surprising that the Mansour Filter in SFFT v3 could be optimized so well compared to the Mansour Filter in SFFT v2. In SFFT v3, the Mansour Filter can be implemented with a fixed loop count of 2, which is usually enough (see section 4.1.3). Additionally, both loops access the same elements of the vector, but with an offset of 1. Thus, always two neighboring elements are accessed. This can be exploited by loading the two neighboring elements at once, so that the spatial locality of the memory access pattern is increased.

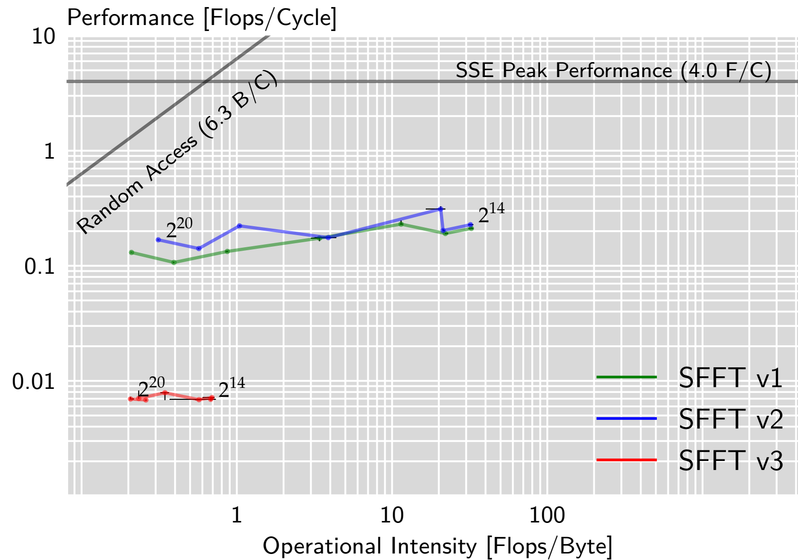



Figure 5.8: Roofline plots of SFFT v1, v2 and v3 (optimized, $n = 2^{14} \ldots 2^{20}$, $k = 50$).

The *Estimate* and *Update* functions have been split in the optimized version, so more parts show up in the profile. With a relative runtime of 48.97 %, the first *Estimate* part is clearly the new bottleneck of the implementation. For various reasons, discussed in section 4.3.3, this part is hard to optimize.

## 5.5 Roofline Analysis

The roofline analysis gives insight in the success of optimizations specifically targeting CPU caches; these optimizations were discussed in section 4.2. Figure 5.8 shows the roofline plots of the optimized SFFT versions. In a first order approximation the operational intensity, regardless of the version, has nearly doubled. Thus, the applied optimizations can be considered successful. However, especially SFFT v3 is still memory-bounded.

## 5.6 Multithreading

As discussed in section 4.4, it is hard to take advantage of multi-core processors when implementing SFFT algorithms. The only approach that works sufficiently well is to run multiple SFFTs in parallel. To implement

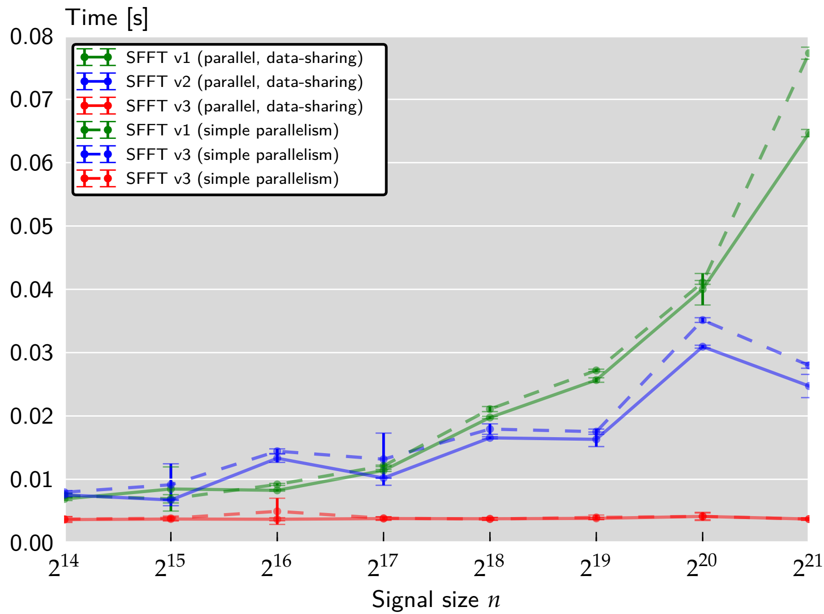



Figure 5.9: An experiment comparing the different multithreading implementations.

this method as good as possible, a data-sharing method was described in chapter 4. Figure 5.9 shows the difference between the simple approach (running multiple SFFTs in parallel without data-sharing) and the more involved approach (running multiple SFFTs in parallel with data-sharing).

The benefit of the data-sharing approach is only very small, and sometimes even non-significant (within error bounds). Nonetheless, there is no reason not to share data between the different threads.

# Chapter 6

# Conclusions

## 6.1 Evaluation

In the last few chapters the SFFT algorithms were analyzed in detail, possible optimizations were described and the optimized SFFT implementation has been benchmarked. Now, the results of this thesis will be evaluated.

One question to answer is whether the SFFT library's performance is competitive to other high-performance DFT libraries. As seen in section 5.2, SFFT v3's performance is generally above 1 GFlop/s or higher, especially for larger $k$. SFFT v1 and v2's performance is more dependent on the signal size $n$ than SFFT v3, and typically lower than SFFT v3's performance. For smaller input sizes, however, the measured performance was still above 1 GFlop/s on the particular machine.

Compared to the benchmark system's peak performance (4.4 GFlop/s single-threaded, scalar execution), this seems small. The roofline analysis revealed that the algorithms are typically memory-bounded for big input sizes. The high-performance library FFTW is also memory-bounded for big input sizes and its maximum performance on the same machine for such big input sizes is also around 1 GFlop/s. Thus, the SFFT implementation's performance can be regarded competitive.

Analysis of the algorithms and the optimizations revealed that some parts of the algorithms can be optimized very successfully, whereas other parts tend to be performance blockers. The Gaussian filter applications, consisting of random spectrum permutation, multiplication with a window function, computing subsamplings and a DFT, belong to the first group. These filters occur in different variations in all SFFT algorithms, and will certainly also be part of improved SFFT algorithms in the future. Other algorithm parts, like the estimation functions in SFFT v3, are particularly hard to optimize, because of their memory access patterns, short nested loops with many branches, or the kind of operations that occur in them.

There are some drawbacks, however. SFFT v1 and v2 implementations only work reliably for very specific input parameters and involve a very fine grained configuration. SFFT v3 does not have this restriction and gives reliable results for all input configurations, but it can only be applied to noiseless signals. Therefore SFFT v3 can only be used in certain scenarios. SFFT v4, which eliminates this last issue, is so complex that at the time of this writing no implementation of it exists.

## 6.2 Outlook

Future work on the SFFT can concentrate on theoretical improvements of the algorithms as well as implementation-targeted performance improvements.

Theoretical work could further reduce the lower asymptotic runtime bound for general signals. At the moment, the lower bound for this is $\mathcal{O}(k\,log(n)\,log(n/k))$ for SFFT v4.

Another project could investigate the possibilities of defining SFFT algorithms without those parts that turned out to be performance blockers. An implementation of such an algorithm could then be further improved, based on the results found in this thesis.

Implementation-targeted work could port the SFFT algorithms to specific hardware or make use of accelerator technologies. The implementation presented in this thesis was optimized towards high performance on general modern x86-based CPUs, and more specifically on Intel CPUs (the implementation can optionally use the Intel specific IPP library). Future work could address specific hardware platforms like GPUs, FPGAs, DSPs or other (co-)processors.

## 6.3 Summary

The SFFT algorithms are a fast way of performing Discrete Fourier Transforms on signals with only a few nonzero Fourier coefficients. While the reference implementations of these algorithms are already faster than state-of-the-art FFT libraries, performance analysis in this thesis showed that the existing implementations can be improved. Guided by this analysis various performance optimization strategies were developed and packaged in a high-performance C++ library. An evaluation of the resulting optimized SFFT library quantified the performance improvements.

# Acknowledgments

I would like to thank Professor Markus Püschel, whose supervision and guidance helped to form this thesis.

The analysis tool *perfplot* was of great use in this thesis. I would like to acknowledge Georg Ofenbeck for his help on installation and usage of the tool.

Furthermore, I would like to express my gratitude to Haitham Hassanieh, Piotr Indyk, Dina Katabi and Eric Price from MIT. Without their work on the Sparse Fast Fourier Transform, this thesis would not have been possible. I am especially grateful for getting access to the unpublished reference implementation of SFFT v3.

# Appendix A

# Manual

## A.1 Introduction

The *Sparse Fast Fourier Transform* is a DFT algorithm specifically designed for signals with a sparse frequency domain. This library is a high-performance C++ implementation of versions 1, 2, and 3 of the different SFFT variants.

### *A.1.1 When Should I use the SFFT library?*

You should use the SFFT library when you want to compute the Discrete Fourier Transform of a signal and only a few frequency components occur in the signal. Your signal may be noisy or not, but currently there are some limitations for noisy signals (see *Limitations and Known Bugs*).

### *A.1.2 Target Platform*

The SFFT library was optimized to run on modern x86 desktop CPUs with SSE support (at least SSE2). Optionally the implementation can use the Intel IPP library, which is only available on Intel platforms.

### *A.1.3 Limitations and Known Bugs*

The SFFT library features implementations of SFFT v1, v2, and v3. SFFT v1 and v2 currently only work with a few specific input parameters. SFFT v3 cannot handle signals with noise.

There are no known bugs so far.

### *A.1.4 Credits*

The SFFT algorithms were invented by Haitham Hassanieh, Piotr Indyk, Dina Katabi and Eric Price. The implementation of this library is based on

their reference implementation of the SFFT. Their results and publications are found on the Sparse Fast Fourier Transform Website.

## A.2 Installation

### *A.2.1 Prerequisites*

The SFFT library was only tested on Linux systems and is only guaranteed to work there. However, the library should also be able to compile on other platforms and operating systems.

The following packages have to be installed to compile the library:

- Python (any version > 2.3, including Python 3), used by the waf build system
- FFTW 3
- *(optionally)* Intel Integrated Performance Primitives

If you want to build benchmark tools, also install

- Valgrind

The SFFT library is known to work the following compilers:

- GCC (tested with GCC 4.4 and 4.7)
- Intel C++ Compiler (only versions >= 13, does NOT work with ICC 12)

### *A.2.2 Compiling From Source and Installation*

Unpack the tarball and change into the newly created directory (sfft-*version*). Then, the SFFT library can be built with a simple:

```
$ ./configure
$ make
```

and installed with:

```
$ make install
```

Some configuration options can be passed to the configuration script. The most important are:

```
$ ./configure --help

[...]
--debug                 compile in debug mode
```

```
--profile               add source-level profiling to instruction
    counting programs
--without-ipp           do not the Intel Performance Primitives
    library
[...]
```

Use `--debug` and `--profile` are only useful when developing (see *Development*). The option `--without-ipp` is to be used when you do not have Intel IPP installed.

When these steps succeded, you should be ready to use the SFFT library.

### *A.2.3 Linking against the SFFT Library*

Two versions of the SFFT library are built when compiling the sourcecode: a static library (libsfft.a) and a shared library (libsfft.so). You can link these libraries in your programs like any other library, but you have to make sure that you link dependencies as well.

Do not forget to link:

- FFTW, for example via *pkg-config*: `pkg-config --cflags --libs fftw3`
- Intel IPP (if not disabled via `--without-ipp`), e.g. `-lippvm -lipps -pthread`
- Your compilers OpenMP library, for example `-lgomp` for GCC
- *libm* and *librt* (`-lm -lrt`)

## A.3 Usage

All types and functions of the SFFT library are defined in the header `sfft.h`. Include it at the beginning of your program.

### *A.3.1 Computing Sparse DFTs*

#### Creating Plans

SFFT executions consist of two seperate steps: planning and execution. The planning phase is only executed once for specific input parameters. After that, many Sparse DFTs with these input parameters can be computed (on different input vectors). This concept is similar to FFTW's concept of plans.

You can create a plan with a call to `sfft_plan`:

```
sfft_plan* sfft_make_plan(int n, int k, sfft_version version,
                          int fftw_optimization);
```

The call returns a pointer to a struct of type `sfft_plan`, which has to be manually freed with `sfft_free_plan`. Parameters of `sfft_make_plan` are:

`n` The size of the input vector.

`k` The number of frequencies in the signal, i.e. the signal's *sparsity*.

`version` The SFFT algorithm version to use. This must be one of the values `SFFT_VERSION_1`, `SFFT_VERSION_2`, or `SFFT_VERSION_3`.

`fftw_optimization` FFTW optimization level. Usually one of `FFTW_MEASURE` and `FFTW_ESTIMATE`. Since experiments showed that there is little benefit in using the more expensive `FFTW_MEASURE`, the best choice is typically `FFTW_ESTIMATE`.

#### Creating Input Vectors

The storage for SFFT input vectors has to allocated using `sfft_malloc`:

```
void* sfft_malloc(size_t s);
```

The reason for this is that the implementation requires a specific memory alignment on the input vectors. You can use `sfft_malloc` as a drop-in replacement for `malloc`.

Input vectors should be of type `complex_t`, which is a typedef to the C standard library's type `double complex`.

Storage allocated with `sfft_malloc` must be freed with this function:

```
void sfft_free(void*);
```

#### Creating the Output Datastructure

The output of the SFFT is stored in an associative array that maps frequency coordinates to coefficients. The array should be of type `sfft_output`, which is a typedef to an `std::unordered_map`. Before executing the SFFT plans, you need to create the output datastructure. A pointer to it is passed to the SFFT execution call and the datastructure filled with the result.

#### Computing a Single Sparse DFT

Once a plan is created, input vectors are created filled with data, and an output object was allocated, the SFFT plans can be executed. The function for this is:

```
void sfft_exec(sfft_plan* plan, complex_t* in, sfft_output* out);
```

Parameters should be self-explanatory. After execution of this function, the output of the DFT is stored in `*out`.

| Signal size $n$ | # frequencies $k$ | Signal size $n$ | # frequencies $k$ |
|---:|---:|---:|---:|
| 8192 | 50 | 8388608 | 50 |
| 16384 | 50 | 16777216 | 50 |
| 32768 | 50 | 4194304 | 50 |
| 65536 | 50 | 4194304 | 100 |
| 131072 | 50 | 4194304 | 200 |
| 262144 | 50 | 4194304 | 500 |
| 524288 | 50 | 4194304 | 1000 |
| 1048576 | 50 | 4194304 | 2000 |
| 2097152 | 50 | 4194304 | 2500 |
| 4194304 | 50 | 4194304 | 4000 |

Table A.1: Valid input parameter combinations for SFFT v1 and v2.

**Computing Multiple Sparse DFTs**

If you want to run multiple SFFT calls on different inputs (but with the same input sizes), you can use `sfft_exec_many` to run the calls in parallel:

```
void sfft_exec_many(sfft_plan* plan,
                    int num, complex_t** in, sfft_output* out);
```

The function is very similar to `sfft_exec`, but you can pass it put `num` input-vectors and `num` output-objects. The SFFT library used OpenMP for parallelization; thus, you can use either the environment variable `OMP_NUM_THREADS` or OpenMP library functions to adjust the number of threads. Be careful: do *not* use different thread number configuration for the call to `sfft_make_plan` and `sfft_exec_many`. Otherwise your program will crash!

### *A.3.2 SFFT Versions*

Currently, three different SFFT versions are implemented: SFFT v1, v2, and v3.

SFFT v3 is the algorithm of choice when your input signals are exactly-sparse; that is, there is no additional noise in the signals. SFFT v3 will not work with noisy signals.

SFFT v1 and v2 can also be applied to noisy signals, but they only work with certain input parameter combinations. Valid input parameters combinations are shown in Table A.1.

## A.4 Development

### A.4.1 Development and Benchmark Tools

The SFFT library includes some useful tools for development and benchmarking. To enable them, you have to configure with the `--develop` flag. Then, the following programs will be built additionally:

`sfft-cachemisses` Runs an SFFT on random input. The tool is handy when used with Valgrind's cachegrind tool. The program includes some instructions to disable valgrind during the input-generation and planning phases. Thus, when the program is analyzed with cachegrind, only the execution phase will be observed.

`sfft-instruction_count` Counts the floating point instructions of the specified SFFT call (configured with program parameters, see below) and prints them. When the configuration option `--profile` was defined, this will also print a profile of the SFFT call.

`sfft-profiling` Another program that runs a configurable SFFT call. This program will be compiled with the profiling flags `pg`, so that it can be analyzed with the `gprof` profiling tool.

`sfft-timing` A program that accurately measures the runtime of the specified SFFT call. This can be used by benchmark scripts.

`sfft-timing_many` Similar to `sfft-timing`, but measures the parallel execution of multiple SFFT calls.

`sfft-verification` This program runs the specified SFFT call and checks that the output is correct. This is useful for testing.

All of the programs run one or many SFFT executions. Random input data is generated automatically. The programs share the following common options:

`-n SIZE` The size of the input signal.

`-k NUMBER` Number of frequencies generated in the random input signal.

`-r REPETITIONS` *NOT available for sfft-timing_many.* Allows to compute multiple SFFTs. Default: 1.

`-i NUM` *Only available for sfft-timing_many.* Generate NUM inputs.

`-s` *Only available for sfft-timing_many.* Do not share data between threads. This is slower.

`-v VERSION` Selects the algorithm version to use. `VERSION` is either 1, 2, or 3. “

`-o` When `-o` is used, `FFTW_MEASURE` is used for FFTW calls instead of `FFTW_ESTIMATE`.

`-h` Displays help.

*A.4.2 An Overview of the Sourcecode*

Here is an overview of the purpose of different sourcefiles:

**cachemisses.cc, timing.cc, timing_many.cc, instruction_count.cc,**

**instruction_count.cc, verification.cc, simulation.[cc,h]**
The `main` routines and some support code for all development tools are located in these files.

**computefourier-1.0-2.0.[cc,h]** Algorithm sourcecode for SFFT v1 and v2.

**computefourier-3.0.[cc,h]** Algorithm sourcecode for SFFT v3.

**fft.h, common.[cc,h], utils.[cc,h]** Some common code and datatypes.

**fftw.[cc,h]** Interface code for FFTW calls.

**filters.[cc,h]** The routines to generate filter vectors are in here.

**intrinsics.h** Some compiler-specific abstractions to include the correct intrinsics header.

**parameters.[cc,h]** Parameter configuration for SFFT v1, v2.

**profiling_tools.h** Some preprocessor tools to allow profiling, used when compiled with `--profile`.

**roofline.cc** A program to use with the roofline tool perfplot. Can be built with `tools/build-roofline.sh`.

**sfft.[cc,h]** User interface code and basic datastructures. The headerfile is to be included by users.

**timer.[cc,h]** Functions for accurate timing, used by `sfft-timing`.

**flopcount/** Files in this directory are used to count floating point operations, used by `sfft-instruction_count`.

# List of Figures


1.1 Performance of DFTs of signals with $k = 50$ frequency components. . . . 8

2.1 A signal $x$ before and after permutation. . . . 15
2.2 Amplitude spectrum of a simple flat window function based on a Gaussian function. . . . 16
2.3 Gaussian Standard Window Function applied to a sample signal. . . . 18
2.4 Effects of the individual signal manipulation steps in the frequency domain. . . . 21
2.5 A simplified flow diagram of SFFT v1. . . . 22
2.6 A simplified flow diagram of SFFT v3. . . . 26

3.1 Runtime of different non-optimized SFFT versions versus signal size $n$ ($k = 50$). . . . 34
3.2 Runtime of different non-optimized SFFT versions versus signal sparsity $k$ ($n = 2^{22}$). . . . 34
3.3 Performance of SFFT v1, v2 and v3 (non-optimized, $k = 50$) and FFTW. . . . 35
3.4 An exemplary roofline plot. . . . 38
3.5 Roofline plots of SFFT v1, v2 and v3 (non-optimized, $n = 2^{14} \dots 2^{20}$, $k = 50$). . . . 39

4.1 A comparison of the different FFTW options for the internal use in the SFFT algorithms. . . . 42
4.2 An illustration of the different storage formats. . . . 50

5.1 Runtime of different non-optimized SFFT versions versus signal size $n$ ($k = 50$). . . . 60
5.2 Speedup of the optimized SFFT implementation compared to the reference implementation ($k = 50$). . . . 60
5.3 Runtime benchmark with varying signal sparsity $k$ ($n = 2^{22}$). . . . 61
5.4 Performance of optimized SFFT v1, v2 and v3 ($k = 50$). . . . 61

5.5 Performance of SFFT v3 with $k = 2000$. . . . . . . . . . . . . . 62
5.6 Cold-cache performance of SFFT v1, v2, v3 and FFTW ($k = 50$). 63
5.7 Speedup of the optimized SFFT library compared to the reference implementation (cold-cache measurements). . . . 63
5.8 Roofline plots of SFFT v1, v2 and v3 (optimized, $n = 2^{14} \dots 2^{20}$, $k = 50$). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
5.9 An experiment comparing the different multithreading implementations. . . . . . . . . . . . . . . . . . . . . . . . . . . 67

# List of Tables


1.1 Different DFT algorithms for sparse signals and their properties. . . . . . 11

3.1 The different versions of the Sparse Fast Fourier Transform and their asymptotic runtimes. . . . . . 29
3.2 Profile of an SFFT v1 run (non-optimized). . . . . . 36
3.3 Profile of an SFFT v2 run (non-optimized). . . . . . 36
3.4 Profile of an SFFT v3 run (non-optimized). . . . . . 37

5.1 Profile of an SFFT v1 run (optimized). . . . . . 64
5.2 Profile of an SFFT v2 run (optimized). . . . . . 64
5.3 Profile of an SFFT v3 run (optimized). . . . . . 65

A.1 Valid input parameter combinations for SFFT v1 and v2. . . 77

# Bibliography


[CT65] James W. Cooley and John W Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–297, May 1965.

[DS00] J. Dongarra and F. Sullivan. Guest Editors Introduction to the top 10 algorithms. *Computing in Science & Engineering*, 2(1):22–23, January 2000.

[FJ] M. Frigo and S.G. Johnson. FFTW Library. http://fftw.org.

[FJ05] M. Frigo and S.G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, February 2005.

[FP09] Franz Franchetti and Markus Puschel. Generating high performance pruned FFT implementations. In *2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 549–552. IEEE, April 2009.

[GST08] AC Gilbert, MJ Strauss, and JA Tropp. A tutorial on fast fourier sampling. *...Processing Magazine, IEEE*, (March 2008):57–66, 2008.

[HAKI12] Haitham Hassanieh, Fadel Adib, Dina Katabi, and Piotr Indyk. Faster GPS via the sparse fourier transform. *Proceedings of the 18th annual international conference on Mobile computing and networking - Mobicom '12*, page 353, 2012.

[Har78] Fredric J Harris. On the use of windows for harmonic analysis with the discrete fourier transform. *Proceedings of the IEEE*, 66(1):51–83, 1978.

[HIKP12a] Haitham Hassanieh, Piotr Indyk, Dina Katabi, and Eric Price. Nearly Optimal Sparse Fourier Transform. *Arxiv preprint arXiv:1201.2501*, (1):28, January 2012.

[HIKP12b] Haitham Hassanieh, Piotr Indyk, Dina Katabi, and Eric Price. Simple and practical algorithm for sparse Fourier transform. In *SODA '12 Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1183–1194. SIAM ©2012, January 2012.

[IGS07] MA Iwen, A Gilbert, and M Strauss. Empirical evaluation of a sub-linear time sparse DFT algorithm. *Communications in Mathematical ...*, 5(4):981–998, 2007.

[IM] Piotr Indyk and D S Mar. Sample-Optimal Average-Case Sparse Fourier Transform in Two. pages 1–30.

[Iwe] Mark Iwen. AAFFT. http://aafftannarborfa.sourceforge.net/.

[KHPI] Dina Katabi, Haitham Hassanieh, Eric Price, and Piotr Indyk. The SFFT Algorithms. http://groups.csail.mit.edu/netmit/sFFT/.

[Man95] Y Mansour. Randomized interpolation and approximation of sparse polynomials. *SIAM Journal on Computing*, 1995.

[Mar71] J. Markel. FFT pruning. *IEEE Transactions on Audio and Electroacoustics*, 19(4):305–311, December 1971.

[Nvi07] Nvidia. CUDA CUFFT Library. https://developer.nvidia.com/cufft, 2007.

[per13] Perfplot Roofline Analysis Tool. https://github.com/GeorgOfenbeck/perfplot, 2013.

[RKH10] K.R. Rao, D.N. Kim, and J.-J. Hwang. *Fast Fourier Transform - Algorithms and Applications*. Signals and Communication Technology. Springer Netherlands, Dordrecht, 2010.

[Sil] Craig Silverstein. The Google SparseHash Library. https://code.google.com/p/sparsehash/.

[SR79] T. Sreenivas and P. Rao. FFT algorithm for both input and output pruning. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 27(3):291–292, June 1979.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. Roofline. *Communications of the ACM*, 52(4):65, April 2009.