SFFT Documentation

Release 0.1

Jörn Schumacher

June 11, 2013

CONTENTS

Contents:

CHAPTER

ONE

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.

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*).

1.2 Target Platform

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

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.

1.4 Disclaimer

The current SFFT implementation is in an experimental state. It is NOT intended to be used as a drop-in replacement for the FFT library of your choice. Be prepared to find bugs. There is absolutely NO WARRANTY for the correct functioning of this software.

1.5 Credits

The original SFFT sourcecode was developed by Haitham Hassanieh, Piotr Indyk, Dina Katabi, and Eric Price at the Computer Science and Artifical Intelligence Lab at MIT. The original sourcecode and contact information can be found at their website Sparse Fast Fourier Transform Website.

Performance optimizations were developed by Jörn Schumacher as part of his Master Thesis Project at the Computer Science Department of ETH Zurich in 2013, under the supervision of Prof. Markus Püschel.

1.6 Contact Information

If you are interested in the theory behind the Sparse Fast Fourier Transform, contact the inventors of the SFFT, Haitham Hassanieh, Piotr Indyk, Dina Katabi, and Eric Price, at their Sparse Fast Fourier Transform Website.

If you are interested in performance optimizations that were applied, contact Jörn Schumacher at joerns@student.ethz.ch.

CHAPTER

TWO

INSTALLATION

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)

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.

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`)

CHAPTER

THREE

USAGE

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

3.1 Computing Sparse DFTs

3.1.1 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. Either `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`.

3.1.2 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*);
```

3.1.3 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.

3.1.4 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`.

3.1.5 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!

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:

Signal Size	Sparsity
8192	50
16384	50
32768	50
65536	50
131072	50
262144	50
524288	50
1048576	50
2097152	50
4194304	50
8388608	50
16777216	50
4194304	50
4194304	100
4194304	200
4194304	500
4194304	1000
4194304	2000
4194304	2500
4194304	4000

CHAPTER

FOUR

DEVELOPMENT

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.

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, 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`.

CHAPTER

FIVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*