High-Performance Sparse Fast Fourier Transforms

Jörn Schumacher
CERN
Geneva, Switzerland
joern.schumacher@cern.ch

Markus Püschel
Department of Computer Science
ETH Zurich, Switzerland
pueschel@inf.ethz.ch

***Abstract*—The sparse fast Fourier transform (SFFT) is a recent novel algorithm to compute discrete Fourier transforms on signals with a sparse frequency domain with an improved asymptotic runtime. Reference implementations exist for different variants of the algorithm and were already shown to be faster than state-of-the-art FFT implementations in cases of sufficient sparsity. However, to date the SFFT has not been carefully optimized for modern processors. In this paper, we first analyze the performance of the existing SFFT implementations and discuss possible improvements. Then we present an optimized implementation. We achieve a speedup of 2–5 compared to the existing code and an efficiency that is competitive to high-performance FFT libraries.**

***Index Terms*—Fast Fourier transform, Software performance, SIMD processors**

I. Introduction

The sparse fast Fourier transform (SFFT) [1], [2] is a recent novel algorithm for computing a discrete Fourier transform (DFT) if the frequency domain is approximately or exactly sparse, a situation that is common in signal processing. A concrete application is presented in [3], where a particular algorithm in the GPS system is improved by using techniques similar to the SFFT. Other applications from [1], [2] include audio and video compression, where signals are naturally sparse in the frequency domain. The algorithm is fundamentally different to prior methods for this situation, which are based on pruning, and achieves an improved asymptotic runtime. Four different versions of the algorithm have been proposed; prototypical implementations of two have been made available at [4] and a third one has been implemented. To date, the code has not yet been packaged as an easy-to-use library since it requires several parameters to be set that are not easy to determine. Further, the SFFT code has not yet been optimized to map well to the memory hierarchy and SIMD instructions available on modern processors.

Related work. The classical approach to FFTs on signals of length n with k-sparse (k nonzero entries) frequency domain is pruning, which removes unneeded computations to reduce the operations count from $O(n \log n)$ to $O(n \log k)$ [5]. However, this approach requires that the location of the k nonzero outputs are known. Pruned FFTs can be optimized and also efficiently implemented using SIMD instructions [6]. A different approach is taken by the FADFT-2 [7], [8], which is a probabilistic algorithm that requires $O(k \text{ polylog}(n))$ many operations and does not require the location of the nonzero outputs. The SFFTs have the same property and further improve on that by achieving up to $O(k \log n)$.

It is known that careful optimizations are needed to achieve optimal performance, which has been done for (ordinary) FFTs in [9], [10], [11], [12]. These optimizations include SIMD vectorization, optimization for the memory hierarchy, and the creation of efficient basic blocks for the small FFTs that are needed in the recursive computation [13].

For the SFFT to date no such optimized implementation exists.

Contributions. The main contribution of this paper are implementations of the existing SFFT algorithms to achieve high performance on modern processors. Specifically,

- we first analyze the performance of the existing SFFT implementations to identify possible bottlenecks;
- we identify and perform various optimization including improving locality and using SIMD instructions (focussing on Intel CPUs with SSE);
- we show experimental results that demonstrate a runtime improvement of up to five times compared to the prior code; in particular,
- we show that for large sizes we can achieve an efficiency (performance) similar to the highly optimized library FFTW [10].

This work is based on [14], where more details can be found. The code for all optimized SFFT implementations is available for download at [15].

II. Background

We briefly provide background on the SFFT and describe its structure. The presentation of all details is beyond the scope of this paper and we refer to [1], [2] for more information.

Overview of SFFTs. In the following we will assume that $x \in \mathbb{C}^n$ is an n-dimensional signal, and that $\widehat{x} \in \mathbb{C}^n$ is the k-sparse DFT of x.

The SFFT exists at the time of this writing in four different versions (v1–v4), each of which has different characteristics. Currently no implementation of SFFT v4 exists. All SFFTs are probabilistic, i.e., the correct result is only guaranteed with a certain probability (which can be controlled by parameters). Versions 1, 2, and 4 work on signals with noise; this means the $n - k$ non-significant Fourier coefficients do not have to be exactly zero, but only small compared to the k significant coefficients. Tables I and II give an overview of the asymptotic runtime (or operations count) of the four SFFTs and the other approaches mentioned before in the introduction.

	SFFT v1	SFFT v2	SFFT v3	SFFT v4
Asymptotic Runtime	$\mathcal{O}\left(\log n\sqrt{nk\log(n)}\right)$	$\mathcal{O}\left(\log n\sqrt[3]{nk^2\log(n)}\right)$	$\mathcal{O}\left(k\log n\right)$	$\mathcal{O}\left(k\log n\ \log(n/k)\right)$
Algorithm	Probabilistic	Probabilistic	Probabilistic	Probabilistic
Constraints	Restricted set of input parameters	Restricted set of input parameters	Only exactly k-sparse signals	—
Implementation	[4]	[4]	Unpublished	None

TABLE I
SFFT V1–4 FROM [1], [2]

	Pruning	AAFFT
Asymptotic Runtime	$\mathcal{O}(n\log k)$	$\mathcal{O}(k\,\mathrm{polylog}(n))$
Algorithm	Deterministic	Probabilistic
Constraints	Sparsity pattern must be known in advance	—
Implementation	[6]	[8]

TABLE II
PRIOR FFTS FOR SIGNALS WITH SPARSE FREQUENCY DOMAIN

Common structure of SFFTs. All SFFT versions have a similar structure and essentially consist of two steps. First, there are multiple rounds of *HashToBins* calls. HashToBins is a function that takes a signal x as input and hashes the k Fourier coefficients of $\widehat{x}$ into a small number of bins using a random hash function. A filter function is used, which is typically a Gaussian filter. In some versions, an additional Mansour filter is used as a heuristic improvement. The HashToBins routine is very similar in all SFFT versions and described in detail in the next section.

Finally, there is a *Frequency Estimation* Phase, in which the output of the HashToBins calls is used to construct the DFT $\widehat{x}$ of the input signal x. That involves finding the correct frequency locations and magnitudes. The Frequency Estimation Phase can be very different in the individual SFFT versions.

We describe the two parts in greater detail next.

HashToBins. The HashToBins routine is, in various forms, the core of all SFFT versions. The basic idea behind this function is to hash the k non-zero Fourier coefficients into a small number of B bins. From these bins, the actual locations of the Fourier coefficients are then approximated.

The following steps are performed in a HashToBins call:

1) *Permute* the input vector x with random parameters. It is not necessary to copy or rearrange the vector. Instead, it is sufficient to traverse it in the permuted order. This can be done by choosing a random step size and offset (see the pseudo code below).
2) *Multiply* the permuted vector x' with a filter vector $G \in \mathbb{C}^w$, where $w < n$. For example, G could be a Gaussian Filter. The result of this operation is $x'' \in \mathbb{C}^w$.
3) Compute the *sums* $z_i = \sum_{j=1}^{B} x''_{j\cdot w/B+i}$ for $i = 0,\ldots,B-1$.
4) Compute the B-dimensional *DFT* of z and output $\widehat{z}$.

The first three steps can be performed jointly in a single loop as shown in this pseudo code (assuming z is initialized to zero):

```
for i = 0 ... w
  z[i%B] += x[(i*stepsize+offset)%n] * filter[i]
end
```

Frequency estimation in different SFFT versions. The different SFFT versions utilize the HashToBins routine in different ways to compute the correct result.

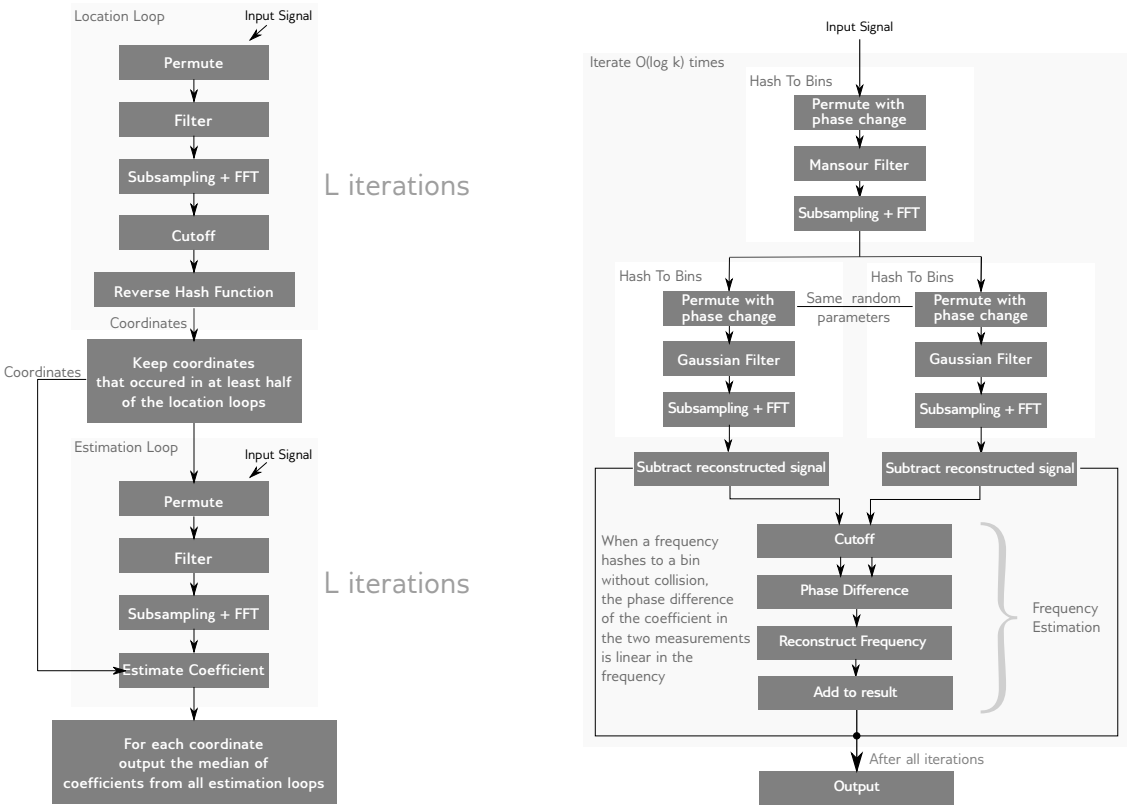
Version 1 of the SFFT consists of multiple executions of two kinds of HashToBins rounds: *location loops* and *estimation loops*. The purpose of the first kind, location loops, is to generate a list of candidate coordinates I. Candidate coordinates $i \in I$ have a certain probability of being indices of one of the k significant, nonzero coefficients in $\widehat{x}$. This probability is bigger for candidate coordinates occurring in more than one location loop iteration. By running multiple iterations of the location loops it is possible to find candidate coordinates with a high probability of being on of the k nonzero coordinates. The second type, estimation loops, are used to exactly determine the coefficients $\widehat{x}_I$ for a given set of coordinates I. This is done by reversing the filter applications in the HashToBins rounds. If there was no hash collision, i.e., only up to one coefficient was hashed to each bin, the coefficient's magnitude can be restored. The overall structure of SFFT v1 is shown in Fig. 1(a).

SFFT v2 is very similar to version v1, but additional HashToBins rounds are used with a special filter, which is a modified version of the algorithm described in [16]. Here, it will be referred to as *Mansour filter*.

While the core ideas of SFFT v3 are still similar to v1 and v2, 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. It is sufficient to update the B-dimensional output of a HashToBins round.

The second important addition in SFFT v3 is an improved scheme for finding the signal's significant frequency coordinates using individual HashToBins rounds. In SFFT v1 and v2, multiple HashToBins rounds were run and their results combined in order to get correct candidate coordinates at a high probability. [1] proves that two distinct calls to



(a) SFFT v1

(b) SFFT v3

Fig. 1. Algorithm structure of SFFTs

HashToBins, one with a phase-shift of the signal and one without, are enough. If a coefficient is hashed to a bin, the phase difference of the two HashToBins results is then linear in the coefficient's coordinate. The structure of an SFFT v3 run is shown in Fig. 1(b).

SFFT v4 uses the same ideas as version 3, but eliminates the restriction that only exact k-sparse signals can be used. The details of this algorithm are very complex, and at the time of this writing no implementation of SFFT v4 exists. Therefore it will not be discussed here.

III. Optimizations

Here we present the main optimizations that we applied to improve the performance and show profiling results for the reference and optimized SFFT v3. More profiles can be found in [14].

Optimization 1: Iteration in chunks. There are several executions of the inner loop, and each time the filter vector is traversed. We improved this by iterating in chunks over the inner loop. It is inefficient when the whole filter vector is traversed multiple times because it may not completely fit into the CPU cache. By iterating over shorter chunks, one can ensure that a chunk fits into cache and thus there may be less cache misses. In other words, the order of the loops is changed from

```
for loop = 1, 2, ..., m {
  for i = 1, 2, ..., filter_length {
    // Compute ...
  }
}
```

to

```
for chunk = 1, 2, ..., nr_of_chunks {
  for loop = 1, 2, ..., m {
    for i = chunk*chunksize, ...,
                (chunk+1)*chunksize {
      // Compute
    }
  }
}
```

It turned out that the best choice for the chunk size is the number of bins B, as the expensive modulo computation for the index of the output vector can be avoided in this special case.

Optimization 2: Interleaved data layout. In the original data layout, the individual loop outputs were stored sequentially in the output vector `x_sampt`. An interleaved storage scheme is better because it improves spatial locality and thus allows for better caching when accessing the memory. This means that the i-th element of the j-th output vector is stored at position $i \cdot m + j$, when m is the number of loops.

Optimization 3: Vectorization. We replaced the built-in complex arithmetic routines that are used with the C complex data type by explicit complex arithmetic using the double data

type. After that we implemented 2-way vectorization using SSE intrinsics. There are two approaches when using vector instructions: either a vector represents a complex number with real and imaginary part, or a vector stores either two real or imaginary parts. The second approach showed a better performance and can also be easily extended to vector architecture that provide longer vectors such as AVX (Advanced Vector Extensions, a SIMD instruction set with 256 bit vectors).

Other Optimizations. The FFTs in the end of the routine can be computed all at once, instead of several independent FFTs. FFTW has an interface to allow the computation of multiple DFTs, that also allows FFTW to schedule the computations in the most efficient way.

We split the execution of the SFFT run into a planning and an execution phase (the planning is required by FFTW to at least precompute the twiddles). This way, the planning cost can be shared among multiple SFFT calls and is only a one-time cost.

In SFFT v3 the loop count in the HashToBins routine or the Frequency Estimation phase is often two, and thus it makes sense to implement a special version of HashToBins with a fixed, constant loop count. The loop was then unrolled and further optimizations applied.

The Frequency Estimation phase in all algorithm versions was vectorized using SSE intrinsics, though the success of this optimization was much smaller than for the HashToBins routine. The code here is not as suited for vectorization, for example because of the many branches inside the loops.

We replaced the $O(\log n)$-access-time result data type `std::map` with the hash-map implementation `std::unordered_map` (C++11) ($O(1)$ access-time), which significantly improved the algorithm speed.

An OpenMP-based multithreaded implementation of the SFFT algorithm showed no performance gain as the overhead of the multithreading was too big. Instead, the SFFT library was modified so that multiple SFFTs on different input data can be run in parallel. Now multiple cores can now be used to compute multiple SFFTs at once, where as much data as possible shared among the threads.

The source code was ported to support the newest Intel compiler, which performs additional optimizations. Also, the IPP library (Intel Performance Primitives) was used in the final implementation, mainly for its highly optimized and vectorized trigonometric functions.

Profiling. Table III shows a runtime profiles of SFFT v3 before and after optimizations for $n = 2^{20}$ and $k = 50$. The flop count is obtained by measurement using the processors performance counters. Profiles for SFFT v1 and v2 show similar results, though the optimizations were more successful for SFFT v3, mostly because the implementation of SFFT v3 could be reduced by fixing the number of inner loops and applying loop unrolling.

Compared to the baseline, especially the filters have improved. In the baseline profile the HashToBins calls 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 HashToBins calls sum up to less than 18 % of the runtime (Mansour Filter 9.67 %, Gauss Filter 4.91 %, Permuted Gauss Filter 3.01 %). Thus, all HashToBins routines, especially the ones with Gaussian Filters, could be improved significantly.

The HashToBins calls with 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, lead to a well performing implementation of these filters.

The Mansour Filter HashToBins call in SFFT v3 could be optimized so well since the it could be implemented with a fixed loop count of 2. Additionally, both loops access the same elements of the vector, but with an offset of 1, which yields excellent spatial locality.

With a relative runtime of 48.97 %, the first *Frequency Estimation* part is clearly the new bottleneck of the implementation.

IV. Results

In this section we present some benchmark results comparing runtime and performance of our optimized SFFTs to prior versions and FFTW. FFTW is known to be one of the fastest FFT libraries available but has no special support for sparsity.

Experimental Setup. The experiments presented were performed on a single core of an Intel(R) Xeon(R) E5-2660 Sandy Bridge CPU, 2.20 GHz. Each core has a 64 KB L1-cache and a 256 KB L2-cache. An additional 20 MB L3-cache is shared among the cores. All measurements are done with warm cache, i.e., as an average over a sufficient number of iterations.

Speedup. The first question we address is how our optimized SFFT code compares to the original implementation. Fig. 3 shows the speedup we achieved for all three implemented SFFT versions for a sparosty of $k = 50$. Higher is better. For v3 we achieved between 4–5x. This shows that processor-cognizant optimizations (vectorization, cache-friendly memory access patterns) are worthwhile and essential.

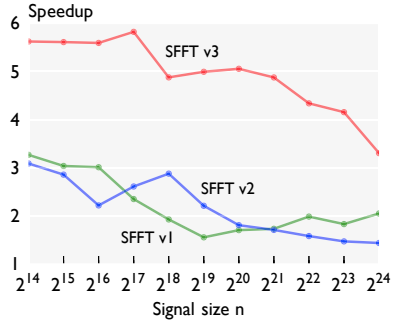
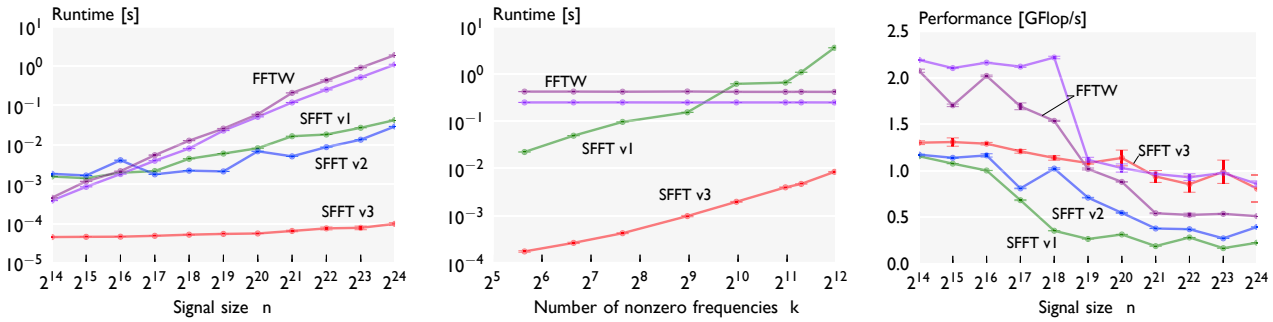


Fig. 3. Speedup: our optimized SFFT versus original SFFT ($k = 50$)

Runtime. Next, we investigate the actual runtime of SFFT v1–3 compared to FFTW, which is run in both MEASURE and ESTIMATE mode. The former uses search to find the best FFT recursion; the latter uses a heuristic. Fig. 2(a) shows the results

	Unoptimized			*Optimized*		
Function	*Runtime [s]*	*% of Total Time*	*Performance [Gflop/s]*	*Runtime [s]*	*% of Total Time*	*Performance [Gflop/s]*
HashToBins (Mansour Filter)	7.71e-05	19.81	0.20	1.45e-05	9.67	1.04
Frequency Estimation	7.33e-05	18.83	0.12	7.36e-05	48.97	0.10
HashToBins (Gauss Filter)	1.01e-04	26.01	0.14	7.39e-06	4.91	1.44
Frequency Estimation	2.46e-05	6.34	0.16	1.69e-05	11.25	0.51
HashToBins (Permuted Gauss F.)	6.72e-05	17.25	0.14	4.53e-06	3.01	0.96
Frequency Estimation	5.87e-06	1.51	0.13	1.79e-05	11.88	0.41
Other	3.99e-05	10.25	0.15	1.55e-05	10.30	0.61
Sum	3.89e-04	100.00		1.50e-04	99.99	

TABLE III
RUNTIME PROFILE OF SFFT V3



(a) Runtime for $k = 50$ (b) Runtime for $n = 2^{22}$ (c) Performance for $k = 50$

Fig. 2. Runtime and performance benchmarks of our optimized SFFT v1–3 against FFTW. For the performance plots note that the operation counts of the implementations have been reduced compared to the original version.

again for fixed $k = 50$. The gain becomes very significant for large sizes, in particular for v3, which is 4 orders of magnitude faster for $n = 2^{24}$. Fig. 2(b) fixed the size to $n = 2^{22}$ and varies k. Even for $k = 2^{12}$, SFFT v3 is still about 50 times faster than FFTW.

Performance. Finally, we investigate the performance of the different implementations, measured in Gflop/s. The performance measures the efficiency of the code, i.e., how well it is optimized. Fig. 2(c) shows the results. Note that the algorithms have vastly different operation counts (even asymptotically; see Table I), so the performance is not (inversely) proportional to the runtime. We observe that the performance of the SFFT is mostly lower than that of FFTW, but for large sizes and v3 we achieve about the same level. Since FFTW is known to be highly optimized, this shows that the same can be achieved for the SFFT.

V. CONCLUSION

Once fully implemented including the automatic choice of the needed parameters, the SFFT has the potential to become the algorithm of choice for many performance-critical sparse DFTs. In this paper we contributed a speedup of about 5x for the case of exact sparseness (SFFT v3), which may translate to similar saving for SFFT v4 once it is implemented. With this speed we showed that at least for large sizes we achieve an efficiency (performance) similar to the highly optimized FFTW.

VI. ACKNOWLEDGEMENTS

We thank the authors of SFFT for helpful discussions, for making SFFT v3 available to us and for allowing us to post the optimized version at [15].

REFERENCES

[1] H. Hassanieh, P. Indyk, D. Katabi, and E. Price, "Nearly Optimal Sparse Fourier Transform," in *ACM Symposium on Theory of Computing (STOC)*, 2012, pp. 563–578.
[2] ——, "Simple and practical algorithm for sparse Fourier transform," in *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2012, pp. 1183–1194.
[3] H. Hassanieh, F. Adib, D. Katabi, and P. Indyk, "Faster GPS via the sparse Fourier transform," *Proc. International Conference on Mobile Computing and Networking (Mobicom)*, p. 353, 2012.
[4] D. Katabi, H. Hassanieh, E. Price, and P. Indyk, "SFFT Website," groups.csail.mit.edu/netmit/sFFT/.
[5] J. Markel, "FFT pruning," *IEEE Trans. on Audio and Electroacoustics*, vol. 19, no. 4, pp. 305–311, 1971.
[6] F. Franchetti and M. Püschel, "Generating high-performance pruned FFT implementations," in *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2009, pp. 549–552.
[7] M. Iwen, A. Gilbert, and M. Strauss, "Empirical evaluation of a sub-linear time sparse DFT algorithm," *Communications in Mathematical Sciences*, vol. 5, no. 4, pp. 981–998, 2007.
[8] M. Iwen, "AAFFT," aafftannarborfa.sourceforge.net.
[9] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 3, 1998, pp. 1381–1384, www.fftw.org.
[10] ——, "The design and implementation of FFTW3," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 216–231, 2005.

[11] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232– 275, 2005.

[12] F. Franchetti, M. Püschel, Y. Voronenko, S. Chellappa, and J. M. F. Moura, "Discrete Fourier transform on multicore," *IEEE Signal Processing Magazine, special issue on "Signal Processing on Platforms with Multiple Cores"*, vol. 26, no. 6, pp. 90–102, 2009.

[13] M. Frigo, "A fast Fourier transform compiler," in *Programming Language Design and Implementation (PLDI)*, 1999, pp. 169–180.

[14] J. Schumacher, "High performance sparse fast Fourier transform," Master's thesis, Computer Science, ETH Zurich, Switzerland, 2013, http://spiral.ece.cmu.edu:8080/pub-spiral/abstract.jsp?id=169.

[15] J. Schumacher and M. Püschel, "Optimized sparse fast Fourier transform: Website and code," www.spiral.net/software/sfft.html.

[16] Y. Mansour, "Randomized interpolation and approximation of sparse polynomials," *SIAM Journal on Computing*, vol. 24, no. 2, pp. 357–368, 1995.