Draft version August 14, 2023
Typeset using LaTeX **preprint** style in AASTeX63

FFT algorithm review from the literature

Bettina Posselt

ABSTRACT

This report comprises the following: (i) a brief reminder about the Discrete Fourier Transform and Fast Fourier Transform, (ii) what is FFTW, (iii) which algorithms are currently implemented in Cheetah, (iv) summary of literature on Fast Fourier Transform algorithms and possibilities to improve the Cheetah implementation, e.g., with a streaming Fast Fourier Transform.

1. REMINDER DFT AND FFT

Among the multitude of literature on Fourier techniques, e.g., van der Klis (1989) and Ransom et al. (2002) provide detailed introductions with an astrophysical (and pulsar) context, while Olson (2017) represents a more general textbook on this topic. Here, only a very brief summary is given. The *Discrete Fourier Transform*, DFT, of a signal (time series) x_j with N measurements ($j = 0, 1, \dots, N-1$) is defined by k ($k = 0, 1, \dots, N-1$) complex Fourier coefficients:

$$A_k = \sum_{j=0}^{N-1} x_j \ \mathrm{e}^{-2\pi \mathrm{i} jk/N}, \tag{1}$$

where $\mathrm{i} = \sqrt{-1}$. k is also called the Fourier frequency or wavenumber. Together they form the DFT. It is typically assumed that the time series x_j is uniformly spaced. Mathematically, the equation 1 is the same as a matrix vector product where the matrix is $\mathrm{e}^{-2\pi \mathrm{i} jk/N}$ with $N \times N$ elements and the vector is x_j with N elements. Since N A_ks need to be calculated, this requires N^2 steps. Calculating those directly as described in equation 1 is sometimes called a "brute-force" Fourier transform algorithm.

The Fast Fourier Transform, FFT, reduces the required calculation to $N \log_2(N)$ steps, a huge difference for large N. This is achieved by convenient rewriting of the sums. For this, the following relations are helpful: $\mathrm{e}^{-2\pi \mathrm{i} z} = 1$ for any integer z, and $A_{-k} = A_k^*$ where A_k^* is the complex conjugate. The best-known FFT algorithms use factorisation of N, e.g., by following the methods by Cooley & Tukey (1965). For instance, one writes the sums for A_k as two sums of odd and even numbers (one additionally assumes that N is a power of two, then $j = j_1 + j_2$ in 1, with $j_1 = 2m$ and $j_2 = 2m+1$ and $m = 0, ..., (N-2)/2$). One obtains a $(N/2-1)$ sum over the even x_{j1} and a $(N/2-1)$ sum over the odd x_{j2}, the latter are multiplied with a complex "twiddle factor" ($\mathrm{e}^{-2\pi \mathrm{i} k/N}$). Thus, one has reduced the N DFT to two $N/2$ DFTs.

More generally, in the Cooley-Tukey factorization one expresses $N = N_r N_c$ and defines $j = j_1 N_c + j_2$ ($j_1 = 0, \dots, N_r - 1$; $j_2 = 0, \dots, N_c - 1$) and $k = k_1 + k_2 N_r$ ($k_1 = 0, \dots, N_r - 1$; $k_2 = 0, \dots, N_c - 1$). Using these definitions with equation 1 and simplifying (see, e.g., Ransom et al. 2002), one obtains N_c DFTs of length N_r multiplied with a twiddle factor as inner term of the overall sum expression. Then an outer sum is performed which is N_r DFTs of lengths N_c. The final result is actually a transposed version of equation 1. One basically transforms a 1d DFT of size N into a 2d DFT of size $N_r \times N_c$. There are various other FFT methods. For instance, one can use the prime-factor (Good-Thomas) algorithm if N can be factorised into two (different) prime numbers. Bruun's algorithm or Winograd algorithm use different polynomial-factorization approaches to the DFT.

1.1. *Why are there different FFT algorithms?*

Besides reducing the number of calculations, the speed of a FFT with a computer also depends on how the cache memory is handled. Memory is restricted, and thus it may be wiser to intelligently overwrite existing arrays (e.g., the time series at start) – called *in-place* algorithm – than creating (one or several) new distinct arrays – called *out-of-place* algorithm. The latter often requires auxillary storage. In addition, it matters how the individual elements in the array are accessed in the memory, either consecutively (fastest performance) or in a scattered manner (less efficient). For in-place algorithms in particular, one must therefore choose carefully when which element should be calculated. Reordering, e.g., by specific permutations of the array elements (such as bit-reversal in case of $N = 2N_2$) can maximize (cache) memory access speed. Since the optimisation of FFT speed depends on the hardware specifics as well as the size of the signal array (i.e., the size of the DFT and possibilities of factorisation), there exist many different FFT algorithms. A Cooley-Tukey FFT can be implemented as a recursive, so-called "divide-and-conquer" algorithm, where the 2d matrix can be broken up into smaller pieces to optimally use the cache.

2. WHAT IS FFTW?

The *Fastest Fourier Transform in the West* (FFTW) was developed by Frigo & Johnson (1998). Version 3.3.10 (Sep 15th, 2021) is the latest stable release of FFTW3 (Frigo & Johnson 2005), and many details, references, and links can be found at http://www.fftw.org. FFTW consists of many highly optmized blocks of C-code, codelets (containing, e.g., different FFT algorithms for transform sizes of various lengths). In the planning stage, these codelets are combined for a FFT of size N, taking the computing architecture into account, with various combinations and orders summarized as experimental plans. The execution speeds of these different plans (or experiments) are compared, and the most optimal plan is chosen. The actual (many) FFTs (all of size N) can then be executed according to this optimal plan. This machine-generated, performance-optimised code is adaptive to the hardware and can differ for different machines. When the FFTs are executed, the size of the cache does not need to be given. FFTW can do arbitrary-size transforms, parallel transforms (threads, distributed-memory transforms). It is free and portable using a C compiler. The FFTW-specific *wisdom* is a method for saving plans to disk and restoring them. There are different levels/flavours in the planning method, requiring a different running time in the preparation phase. The *wisdom* mechanism makes FFTW very efficient in use, but needs to be evaluated again if hardware or other conditions change.

3. IS FFTW USED IN CHEETAH?

Cheetah loads FFTW3 version 3.3.7-1 (at least on tengu and hippa) from 2017. The latest version, 3.3.10, has some minor(?) bug fixes https://www.fftw.org/release-notes.html. Searching for 'fftw' in the cheetah source code, I only found it in module PWFT, generator PulsarInjection.cpp, and in utils ConvolvePlan.cpp, but not in the FFT or CXFT modules. The latter two seem not to load definitions from PWFT, ConvolvePlan, or PulsarInjection either. Some algorithm 'plans' are compared however. I did not find any fftw3_export/import_wisdom calls, hence I suspect no FFTW-wisdom is saved/loaded anywhere at the moment.

From FftAlgos.h, it seems that only a cuda-fft and an altera-fft exist. The former is GPU-based, the latter FPGA-based (Intel FPGA OpenCL FFT interface). I don't see an existing CPU-based FFT. The cuda-fft uses Nvidia's cuFFT library which is modelled after FFTW, but works on GPUs (interestingly, it also mentions "Streamed execution"). Similarly to FFTW, it creates an optimized plan after experimenting with different FFT setups for the given specifications. The Cheetah TDAS code was written by E. Barr. Hence, there may be similarities of Cheetah's FFT-module with the code by (Barr 2020).

4. SOME LITERATURE FFTS THAT MAY BE RELEVANT FOR CHEETAH

It is interesting whether there is a faster (preferably free) FFT algorithm which could be used in Cheetah for the FFT or CXFT modules. A streaming FFT could be particularly interesting. We decided to concentrate on CPU and GPU algorithms, not FPGA-based ones. Looking for comparisons with FFTW3 seems to be the best starting point to find other high-performance algorithms. Benchmarking was done for FFTW and various other FFT algorithms as listed at http://www.fftw.org/benchfft/ffts.html. However, the most recent algorithm on the FFTW-list is from 2006. I found several newer codes. Examples (just C/C++) are listed on URL https://community.vcvrack.com/t/complete-list-of-c-c-fft-libraries/9153. However, no benchmarking or comparisons are provided there.

Several times, I came across the FFTE, the *Fastest Fourier Transform of the East*. Nikolić et al. (2014) compared the performances of FFTE and FFTW on different supercomputers and find better scalability of FFTW (speedup larger if more CPU cores are included), but faster execution times of FFTE (for < 256 CPU cores). Nikolić et al. (2014) also noted, however, that the FFTE is badly documented.

More comprehensive is the recent comparison of different FFT algorithms (run on a supercomputer) by Ayala et al. (2021) ("Interim report on benchmarking FFT libraries on high performance systems"). They differentiate between CPU and GPU-based algorithms, list the licenses, and compare performance in dependance of node numbers. Although nodes refer to individual computers with multiple cores each, comparison of the node$= 1$ numbers can give us a good idea about the expected relative performance for our work. The quality of the documentation of each algorithm remains unclear at the moment and would require further checks if we decide to explore further. The paper is included in the shared google drive. Below are FFT library overview tables (Fig. 1 and 2) and figure excerpts from Ayala et al. (2021) to give an idea about the content. Figure 3 shows the performance comparisons for CPU-based libraries, Figure 4 for GPU-based libraries. Although the main focus of Ayala et al. (2021) is navigating the 'communication bottleneck' for multi-nodes/multi-processors by their experiments (scaling to many nodes), there are some general conclusions which may be of interest to us: (i) The GPU runs are about $2\times$ faster than the CPU runs for the same number of nodes used; (2) Performance-wise, "...for CPU-based runs there are two groups: the best performing ones are AccFFT, heFFTe, FFTMPI, FFTE, SWFFT, and 2De-comp&FFT. The second group is $2-3\times$ slower with P3DFFT and FFTW. For GPU-based runs, the best time is achieved by heFFTe,

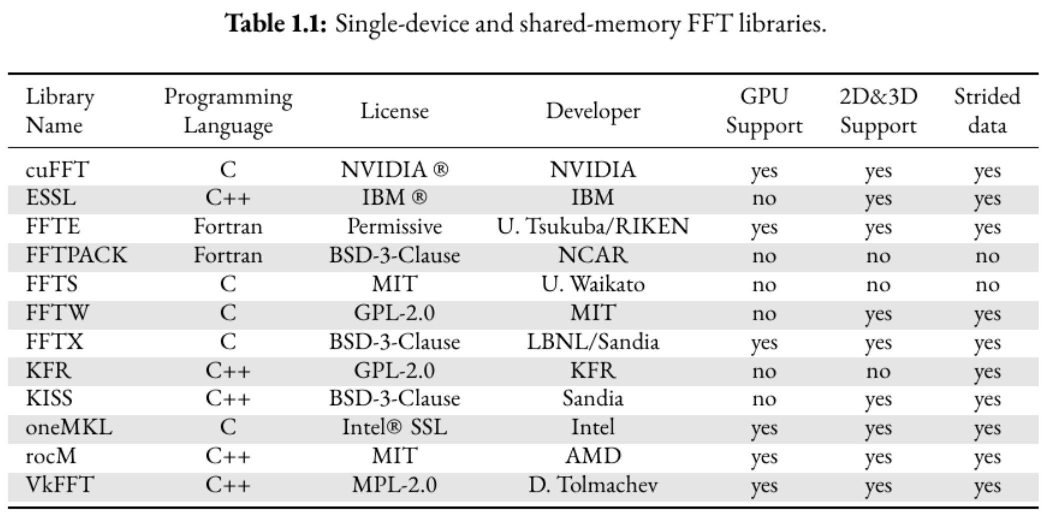

Table 1.1: Single-device and shared-memory FFT libraries.

Library Name	Programming Language	License	Developer	GPU Support	2D&3D Support	Strided data
cuFFT	C	NVIDIA ®	NVIDIA	yes	yes	yes
ESSL	C++	IBM ®	IBM	no	yes	yes
FFTE	Fortran	Permissive	U. Tsukuba/RIKEN	yes	yes	yes
FFTPACK	Fortran	BSD-3-Clause	NCAR	no	no	no
FFTS	C	MIT	U. Waikato	no	no	no
FFTW	C	GPL-2.0	MIT	no	yes	yes
FFTX	C	BSD-3-Clause	LBNL/Sandia	yes	yes	yes
KFR	C++	GPL-2.0	KFR	no	no	yes
KISS	C++	BSD-3-Clause	Sandia	no	yes	yes
oneMKL	C	Intel® SSL	Intel	yes	yes	yes
rocM	C++	MIT	AMD	yes	yes	yes
VkFFT	C++	MPL-2.0	D. Tolmachev	yes	yes	yes

Figure 1. Table 1.1 by Ayala et al. (2021). Strided data means that the data elements are not stored in consecutive memory locations but have a fixed "stride" between them. This helps with cache use optimisation and parallelisation.

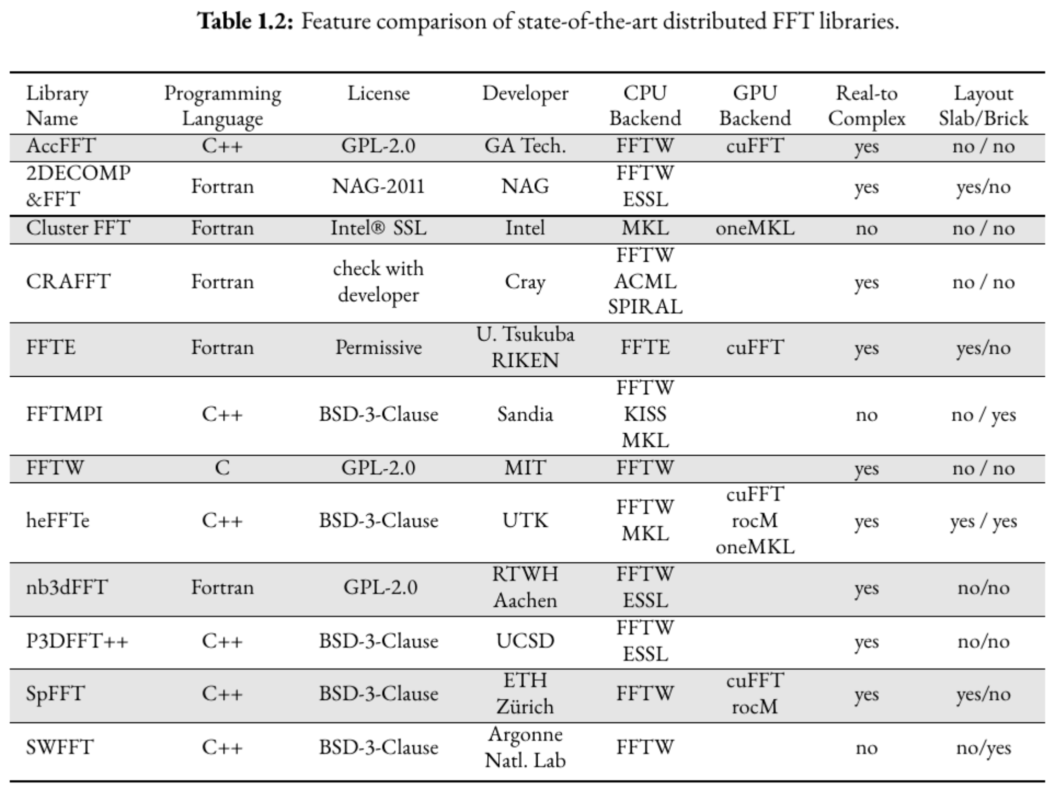

Table 1.2: Feature comparison of state-of-the-art distributed FFT libraries.

Library Name	Programming Language	License	Developer	CPU Backend	GPU Backend	Real-to Complex	Layout Slab/Brick
AccFFT	C++	GPL-2.0	GA Tech.	FFTW	cuFFT	yes	no / no
2DECOMP &FFT	Fortran	NAG-2011	NAG	FFTW ESSL		yes	yes/no
Cluster FFT	Fortran	Intel® SSL	Intel	MKL	oneMKL	no	no / no
CRAFFT	Fortran	check with developer	Cray	FFTW ACML SPIRAL		yes	no / no
FFTE	Fortran	Permissive	U. Tsukuba RIKEN	FFTE	cuFFT	yes	yes/no
FFTMPI	C++	BSD-3-Clause	Sandia	FFTW KISS MKL		no	no / yes
FFTW	C	GPL-2.0	MIT	FFTW		yes	no / no
heFFTe	C++	BSD-3-Clause	UTK	FFTW MKL	cuFFT rocM oneMKL	yes	yes / yes
nb3dFFT	Fortran	GPL-2.0	RTWH Aachen	FFTW ESSL		yes	no/no
P3DFFT++	C++	BSD-3-Clause	UCSD	FFTW ESSL		yes	no/no
SpFFT	C++	BSD-3-Clause	ETH Zürich	FFTW	cuFFT rocM	yes	yes/no
SWFFT	C++	BSD-3-Clause	Argonne Natl. Lab	FFTW		no	no/yes

Figure 2. Table 1.2 (distributed FFT libraries) by Ayala et al. (2021)

followed closely by AccFFT, and FFTE is about 2× slower...". AccFFT was presented by Gholami et al. (2016). On its github, https://github.com/amirgholami/accfft nothing has changed for 5 years. heFFTe (Tomov et al. 2019) was developed in the US Department of Energy's Exascale Computing

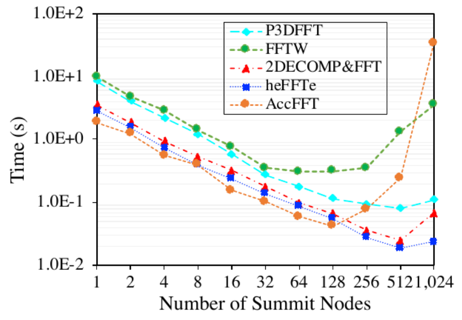

Figure 2.3: Strong scalability for a 3-D FFT of size 1024^3 using 40 MPI processes per node, 1 per IBM POWER9 core, 20 per socket. Using 2 reshapes (transpositions) per FFT direction.

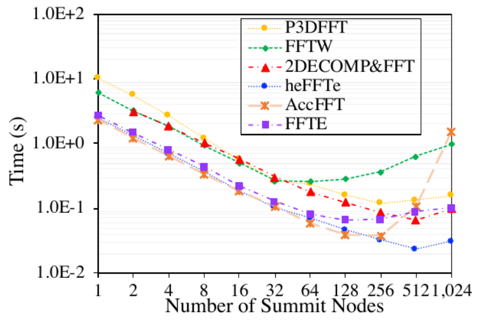

Figure 2.5: Strong scalability for a 3-D FFT of size 1024^3 using 32 MPI processes per node, 1 per IBM POWER9 core, 16 per socket. Using 2 reshapes (transpositions) per FFT direction.

Figure 3. *left*: six CPU libraries going from pencil-shaped input to pencil-shaped output, *right:* using only 32 cores by node because some libraries are size-constrained. Ayala et al. (2021) start from 3D FFTs (tensors) that are decomposed (reshaped) either into 1D *slabs*, 2D *pencils*, or 3D *bricks* to enable parallelised processing. Some libraries can only handle specific shapes with multiple processors/nodes.

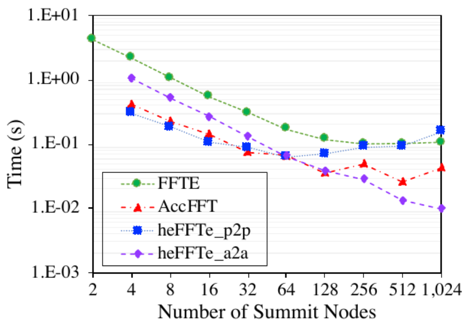

Figure 2.7: Strong scalability for GPU libraries using 4 NVIDIA V100 GPUs per node, 2 per socket.

Figure 4. GPU-based libraries by Ayala et al. (2021)

Project. according to Ayala et al. (2021), heFFTe supports the most single-device backends and enables parallel FFT computations on AMD, Intel, and NVIDIA GPUs, and it includes several features missing by other state-of-the-art libraries. The project is presented on https://icl.utk.edu/fft/, the github link is https://github.com/icl-utk-edu/heffte, and there were updates 2 weeks ago. On a first, quick glance, it looks reasonably well documented. I think it would be worth evaluating (free and easy to install?, ease of use?, benchmark) heFFTe in more detail for PSS-use to see if we could gain a factor 2 in performance, especially because heFFTe can handle CPU and GPU processing.

4.1. *What is about streaming FFT algorithms?*

A streaming FFT computes the FFT of continuously arriving data without needing to buffer or store the entire input before processing. I failed to find a good reference for a recent free streaming FFT code for CPU (but see paragraph at the end of this section). For GPUs, I noted that CuFFT mentions "Streamed cuFFT Transforms" (https://docs.nvidia.com/cuda/cufft/index.html?highlight=streaming#streamed-cufft-transforms). Apparently, different CUDA streams can be handled (with FFTW-like plans), with details on such streams described in the CUDA C++ Programmming Guide. That CuFFT is doing a streaming FFT seems to agree with Lobeiras et al. (2011) (pdf in google drive) who compared a different streaming FFT with an earlier version of CuFFT (with CuFFT doing best of all for large transforms). Further improvements were suggested by Zhao et al. (2023) (pdf in google drive). Their large-scale FFT framework, MFFT, which optimizes parallel FFT with a new mixed-precision optimization technique, outperforms the above mentioned heFFTe (AMD GPU-based version) by a factor 9! However, I don't see any open source code reference or github link for this recent Chinese work, nor any independent check.

Most literature on streaming FFT seems to deal with the implementation via Field-Programmable Gate Arrays (FPGAs). A useful overview about this is provided by Chen (2017), a 2017 PhD thesis on "Optimal Designs for High Throughput Stream Processing using Universal RAM-based Permutation Network" (pdf in google drive). Another helpful FPGA reference for FPGA-based streaming FFT is Serre & Püschel (2018).

I explored the idea of using alternatively "sparse FFTs" (Gilbert et al. 2014 provides some overview, pdf in google drive) because I noticed its mentioning for streaming data or large data. However, a sparse FFT assumes a lot of "meaningless" data, i.e., zeroes, which I think, is not exactly what we have.

Finally, I gave ChatGPT a try (transcript of questions/answers provided on google drive). ChatGPT gave the information that "...one popular streaming FFT algorithm is the "Sliding DFT" [SDFT] algorithm...". Other names for it could be "Overlap-save" or "Overlap-add" method. I was wondering whether one could combine a FFT, say FFTW, with a sliding window (e.g., for half or quarter of the data). ChatGPT's answer was not very useful. It is maybe worth noting that the AI's information is stuck at September 2021 (at least for the open version that I used). I checked very briefly for a few publications on SDFT, some useful standard ones are e.g., Rafii (2018); Grado et al. (2017); Jacobsen & Lyons (2003), an example comparison with FFT was given by Kumar et al. (2015), a SDFT with some astronomical context is in Böhlen & Saad (2019) (all pdfs in google drive). Apparently, the sliding DFT can be very efficient, working recursively and scaling with N instead of $N \log N$. However, there occurs spectral leakage due to the rectangular (sliding) window. It would require further reading, but possibly combining FFT with such a sliding window approach might be worth exploring if other (easier?) options (heFFT, CuFFT) don't perform sufficiently in the benchmarking.

REFERENCES

Ayala, A., Tomov, S., Luszczek, P., et al. 2021, 1

Barr, E. 2020, Peasoup: C++/CUDA GPU pulsar searching library, Astrophysics Source Code Library, record ascl:2001.014. http://ascl.net/2001.014

Böhlen, M. H., & Saad, M. 2019, in Leibniz International Proceedings in Informatics (LIPIcs), Vol. 147, 26th International Symposium on Temporal Representation and Reasoning (TIME 2019), ed. J. Gamper, S. Pinchinat, & G. Sciavicco (Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik), 1:1–1:4, doi: 10.4230/LIPIcs.TIME.2019.1

Chen, R. 2017, PhD thesis, University of Southern California

Cooley, J. W., & Tukey, J. W. 1965, Mathematics of Computation, 19, 297, doi: 10.1090/S0025-5718-1965-0178586-1

Frigo, M., & Johnson, S. G. 1998, in Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing, Vol. 3 (IEEE), 1381–1384

Frigo, M., & Johnson, S. G. 2005, Proceedings of the IEEE, 93, 216

Gholami, A., Hill, J., Malhotra, D., & Biros, G. 2016, AccFFT: A library for distributed-memory FFT on CPU and GPU architectures. https://arxiv.org/abs/1506.07933

Gilbert, A. C., Indyk, P., Iwen, M., & Schmidt, L. 2014, IEEE Signal Processing Magazine, 31, 91, doi: 10.1109/MSP.2014.2329131

Grado, L. L., Johnson, M. D., & Netoff, T. I. 2017, IEEE Signal Processing Magazine, 34, 183, doi: 10.1109/MSP.2017.2718039

Jacobsen, E., & Lyons, R. 2003, IEEE Signal Processing Magazine, 20, 74, doi: 10.1109/MSP.2003.1184347

Kumar, N., Mehra, D. R., & Sharma, b. 2015, International Journal of Engineering Trends and Technology, 30, 346, doi: 10.14445/22315381/IJETT-V30P264

Lobeiras, J., Amor, M., & Doallo, R. 2011, in Proceedings of the 19th International Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP 2011, Ayia Napa, Cyprus, 9-11 February 2011, ed. Y. Cotronis, M. Danelutto, & G. A. Papadopoulos (IEEE Computer Society), 119–126, doi: 10.1109/PDP.2011.31

Nikolić, M., Jovic, A., Jakić, J., Slavnić, V., & Balaz, A. 2014, 2, 163, doi: 10.1007/978-3-319-01520-0-20

Olson, T. 2017, Applied Fourier Analysis: From Signal Processing to Medical Imaging (Springer New York). https://link.springer.com/book/10.1007/978-1-4939-7393-4

Rafii, Z. 2018, IEEE Signal Processing Magazine, 35, 88, doi: 10.1109/MSP.2018.2855727

Ransom, S. M., Eikenberry, S. S., & Middleditch, J. 2002, AJ, 124, 1788, doi: 10.1086/342285

Serre, F., & Püschel, M. 2018, in Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '18 (New York, NY, USA: Association for Computing Machinery), 219ˇ013228, doi: 10.1145/3174243.3174263

Tomov, S., Haidar, A., Ayala, A., Schultz, D., & Dongarra, J. 2019

van der Klis, M. 1989, in NATO Advanced Study Institute (ASI) Series C, Vol. 262, Timing Neutron Stars, ed. H. Ögelman & E. P. J. van den Heuvel, 27, doi: 10.1007/978-94-009-2273-0_3

Zhao, Y., Liu, F., Ma, W., et al. 2023, ACM Trans. Archit. Code Optim., 20, doi: 10.1145/3605148