Single Point Incremental Fourier Transform on 2D Data Streams

Muhammad Saad*, Abraham Bernstein*, Michael H. Böhlen*, Daniele Dell'Aglio†*
* Department of Informatics, University of Zurich, Zurich, Switzerland
† Department of Computer Science, Aalborg University, Aalborg, Denmark
saad@ifi.uzh.ch, bernstein@ifi.uzh.ch, boehlen@ifi.uzh.ch, dade@cs.aau.dk

2021 IEEE 37th International Conference on Data Engineering (ICDE) | 978-1-7281-9184-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICDE51399.2021.00079

***Abstract*—In radio astronomy, antennas monitor portions of the sky to collect radio signals. The antennas produce data streams that are of high volume and velocity (∼2.5 GB/s) and the inverse Fourier transform is used to convert the collected signals into sky images that astrophysicists use to conduct their research. Applying the inverse Fourier transform in a streaming setting, however, is not ideal since its computational complexity is quadratic in the size of the image.**

In this article, we propose the Single Point Incremental Fourier Transform (`SPIFT`), a novel incremental algorithm to produce sequences of sky images. `SPIFT` computes the Fourier transform for a new signal in a linear number of complex multiplications by exploiting twiddle factors, multiplicative constant coefficients. We prove that twiddle factors are periodic and show how circular shifts can be exploited to reuse multiplication results. The cost of the additive operations can be curbed by exploiting the embarrassingly parallel nature of the additions, which modern big data streaming frameworks can leverage to compute slices of the image in parallel. Our experiments suggest that `SPIFT` can efficiently generate sequences of sky images: it computes the complex multiplications 4 to 12x faster than the Discrete Fourier Transform, and its parallelisation of the additive operations shows linear speedup.

I. Introduction

Radio astronomy scientists study objects in the sky by collecting data with radio telescopes arrays like ASKAP—the Australian Square Kilometre Array Pathfinder. Such antennas collect radio signals that are transformed into images through a two step process: in the *observation* stage they measure radio signals, and in the *processing* stage they convert the signals into images by applying the 2D Discrete Fourier Transform (2D-DFT).

Recently, scientists started to get interested in near-real-time applications: they want to observe the results of the image as soon as possible, even while still measuring the radio signals, and trade image quality with responsiveness [1] using sparse regularization methods to estimate the final image. A possible way to tackle the problem of continuously generating such images is to exploit distributed stream processing frameworks, which have been developed to process data in a continuous fashion. One drawback of adapting the processing stage to a streaming setting is the complexity of the 2D-DFT, which is $O(N^2 \log N)$ for an image of size $N \times N$. The 2D-DFT is executed for every new measured radio signal, making it a bottleneck of the continuous processing stage. Therefore, in this article we aim at optimizing the 2D-DFT in a scenario where radio signals are processed sequentially on top of a distributed stream processing framework. This is the first study that focuses on how to reduce the complexity when the 2D-DFT is computed after each new stream update.

As the main result of this study, we show that, in the context of 2D-DFT computation for a single point, the updates that must be applied to the image can be computed in linear time. Our algorithm, the Single Point Incremental Fourier Transform (`SPIFT`), exploits twiddle factors to introduce redundancy in the computation. `SPIFT` proposes circular shifts to leverage this redundancy and reduce from a quadratic to a linear number of multiplications.

In order to update the image, the 2D-DFT of a single point must be added to the 2D-DFT of the signals observed in the past. The cost of the additive operations can be curbed by exploiting the embarrassingly parallel nature of the matrix addition. We exploit modern stream processing engine architectures to slice the image matrix into sub-matrices that are updated in parallel. Since there is no dependency or need for communication between these parallel tasks, the parallelization of the additive operation exhibits a perfect scalability with a linear speedup.

Our experiments show that the single point Fourier transform with circular shifts is 4-12 times faster than the `direct DFT`, which does not use shifts. Also, we found that the throughput of `SPIFT` increases linearly with the increase of the parallelism. Our algorithm is competitive with hardware-optimized libraries and outperforms them for a parallelism degree greater than or equal to 8.

Summarising, the contributions of this paper are as follows:

1) We propose the *single point Fourier transform (`SPFT`)*, which uses circular shifts to compute the 2D-Fourier transform of a single point with a linear number of complex multiplication operations.
2) For a stream setting, we propose the *single point incremental Fourier transform (`SPIFT`)* based on `SPFT`. `SPIFT` incrementally computes the 2D-Fourier transform of the entire stream as observations stream in. `SPIFT` uses less floating point operations than the fast Fourier transform `(FFT)` and the `direct DFT` to compute the 2D-Fourier transform of the stream for each point.
3) We implement and evaluate the complete streaming pipeline with the `SPIFT` algorithm in Apache Flink.
4) We experimentally study the performance of `SPIFT` by comparing it with `FFT` and `direct DFT` for various

978-1-7281-9184-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDE51399.2021.00079

grid sizes and parallelization degrees on real-world and synthetic observational data.

The rest of the article is organized as follows. Section II reviews the algorithms for computing the Fourier transform over streaming data. We introduce the application scenario and the problem statement in Section III. We describe the basics of the Fourier transform in Section IV. We present our main ideas—the circular shifts of twiddle factors and the incremental image computation—in Sections V and VI. We then propose a streaming pipeline and discuss its implementation in Apache Flink in Section VII. We evaluate our approach in Section VIII. We close with conclusions in Section IX. Table I summarizes the notations and terminologies used in the paper.

TABLE I
NOTATIONS AND TERMINOLOGY.

Notation	Description
$\mathbf{A}_t$, $\mathbf{B}_t$, $\mathbf{C}_t$, ...	a time-indexed 2D-grid or matrix
$\mathbf{A}_{t,k}$, $\mathbf{B}_{t,k}$, $\mathbf{C}_{t,k}$, ...	a time-indexed 2D sub-grid or sub-matrix belonging to partition k.
$\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$, ...	a 1D-vector or 1D-tuple
$\mathbf{x}_t$, $\mathbf{y}_t$, $\mathbf{z}_t$, ...	a time-indexed 1D-vector or 1D-tuple
d, e, f, ...	scalar values
$\mathbf{A}$[j, *]	j^{th} row of 2D-grid or matrix $\mathbf{A}$
$\mathbf{A}$[*, k]	k^{th} column of 2D-grid or matrix $\mathbf{A}$
$\mathbf{L}^{u_t,v_t}$	a twiddle factor matrix for coordinates u_t,v_t
$\mathbf{s_t}$	a record in the visibility stream
$\mathcal{J}$, $\mathcal{K}$, $\mathcal{L}$, ...	a function
vis_t	a complex visibility value in the stream at time t
$S(t)$	a stream of all the visibility records till time t
$S^{u,v}(t)$	all the visibility values vis with same (u, v) coordinates from the stream $S(t)$
N	the number of rows or columns of a 2D-grid

II. Related Work

In recent years, distributed stream processing has emerged as the paradigm to process huge volumes of data with high data rates. Several initiatives have led to a broad range of stream processing engines that process huge amounts of data reactively and continuously [2], [3], [4], [5], [6].

A stream computing approach for radio astronomy imaging was first discussed for the ASKAP Central Processor [7]. The study focuses on the convolutional gridding algorithm, i.e., a preliminary step in the radio imaging pipeline to map visibilities to a regular and finite sized 2D-grid. The authors use $System\ S$ to grid a vast number of frequency channels in parallel, overcoming memory limitations and decreasing the processing time. Mahmoud et al. [8] use IBM InfoSphere Streams [9] to compute the power spectral density through a spectrometer pipeline that takes the 1D-FFT of radio astronomy signals, showing that the streaming approach is a viable solution for a real-time spectrometer pipeline. It significantly increases the throughput and can be further improved by using hardware accelerators such as GPUs and FPGAs.

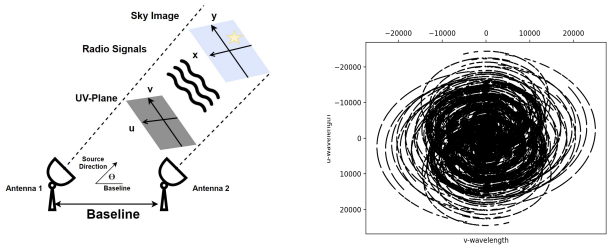
Based upon the input data, existing methods for the Fourier transformation can be categorized into `finite-size data` and `record-wise data`. To compute the Fourier transform of finite-size data, i.e., a vector $\mathbf{a} \in \mathbb{R}^N$ or a matrix $\mathbf{A} \in \mathbb{R}^{M \times N}$, the Cooley-Tukey `FFT` [10] is the state-of-the-art algorithm. `FFT` optimizations either strive to reduce the total number of additions and multiplications [11], [12] or exploit the parallelism offered by hardware accelerators. In this paper we reduce the number of operations for computing a Single Point Fourier transform.

The Fourier transform over a data stream is computed by using a finite-size sliding window over the records. The window selects a finite-size fragment of records from the stream. The idea is to efficiently compute the Fourier transform of each window by reusing the Fourier transform of the previous window. Such techniques [13], [14] reduce the complexity by not recalculating all Fourier coefficients for every window. The sliding discrete Fourier transform [15], [16] further reduces the computational complexity by using the DFT shift theorem. It is used to efficiently compute the spectral components of a sequence that is shifted in the time-domain. However, the finite precision of complex multiplicative coefficients involved in DFT computations of sliding DFT induces error in the output. The error is accumulated with each new record, which compromises accuracy and makes the approach unstable. Hence, [17] proposes a new sliding DFT algorithm which guarantees accuracy and stability at the expense of computational complexity. [18] presents a sliding Fourier transform that ensures stability as well as accuracy with the least computational complexity of all algorithms. This technique is used [19] for botnet detection in real time. When a window slides multiple time units at time, then [20], [21] propose to compute the sliding Fourier transform. The 2D-sliding Fourier transform is introduced by [22]. Hence, the Fourier transform of a data stream is computed by dividing the stream into smaller finite-size input sequences.

In case of radio astronomy stream, the Fourier transform of each record contributes to the sky image. Hence, using a finite-size sliding window on a data stream only produces partial and overlapping sky images. In this paper, we, therefore, focus on an incremental technique that computes Fourier transform of each record and then adds it up to Fourier transform of all previous records to give the sky image of all processed records of stream. The proposed algorithm for computing Fourier transform of a record requires linear number of complex multiplication operations.

III. Application Scenario and Problem Definition

A radio interferometer does not produce sky images directly. Instead, antennas sense the electric field of radio signals and produce a voltage, termed *visibility*. Each pair of antennas measures the visibility in the so-called *UV-plane* that is perpendicular to the direction of the observed source, as shown in Fig. 1(a). A visibility encodes amplitude and phase information of a radio signal and is a single Fourier component of the sky brightness measured at coordinates *u,v* in the UV-plane. The distribution and density of the visibilities across the UV-plane is the UV-coverage. A dense UV-coverage is necessary to produce high-quality sky images [23].

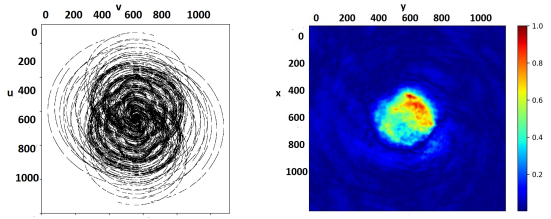


(a) Interferometry (b) UV Plane

Fig. 1. A two antenna interferometer observing the source.

The (u, v) coordinates depend upon the length of the projected vector between two antennas known as *baseline*, and the *angle* between the baseline and the observed source, shown as θ in Fig. 1(a). The angle and the baseline change due to the Earth rotation, so at a given point in time a visibility value is generated at different (u, v) coordinates. The visibilities are observed at real-valued coordinates in the UV-plane as shown in Fig. 1(b).

In order to compute the Fourier transform, the visibilities in the UV-plane must be placed at discrete grid points. The *UV-plane* is discretized to regularly-spaced and finite-sized grid, called *visibility grid* as shown in Fig. 2(a). This discretization is termed *gridding* in the radio astronomy literature [24]. To generate the sky image the Fourier transform is applied to the visibility grid, which yields the result in Fig. 2(b).



(a) Visibility Grid (b) Sky Image

Fig. 2. Radio Imaging.

A visibility vis_t is a complex value measured at coordinate u, v and time t. We assume an already gridded set of continuously arriving observations, and we denote with $S(t) = (\mathbf{s_1}, \mathbf{s_2}, ..., \mathbf{s_t})$ the time indexed stream with all visibility records up to time t. An element $\mathbf{s_t}$ is a triple:

$$\mathbf{s_t} = \langle u_t, v_t, vis_t \rangle$$

where $u_t, v_t \in \mathbb{R}$, $vis_t \in \mathbb{C}$ and $0 \leq u_t, v_t < N$. We refer to all visibilities with the same (u, v) coordinate up to time t as $S^{u,v}(t)$, i.e., $S^{u,v}(t) = \{vis \mid \langle u, u, vis \rangle \in S(t)\}$. A visibility grid $\mathbf{V_t}$ is a time-varying $N \times N$ matrix that is updated as new visibility records arrive. We write $\mathbf{V_t}[u, v]$ to refer to the value of the cell with coordinates (u, v). $\mathbf{V_t}$ at time t can be computed incrementally from the visibility grid at time $t-1$:

$$\mathbf{V_t}[u,v] = \begin{cases} \mathbf{V_{t-1}}[u,v] & \text{if } (u,v) \neq (u_t, v_t) \\ \mathbf{V_{t-1}}[u_t, v_t] + vis_t & \text{otherwise} \end{cases} \quad (1)$$

for $0 \leq u, v < N$.

Note that many observations may fall into the same cell. In this case the visibility value of the cell is the sum of all visibilities that fall into this cell:

$$\mathbf{V_t}[u,v] = \sum_{vis \in S^{u,v}(t)} vis \quad (2)$$

Example 1: As a running example we use a 4×4 visibility grid and the following stream of visibilities:

$$\mathbf{s_1} = \langle 2, 1, 1+2i \rangle, \mathbf{s_2} = \langle 3, 2, -2+3i \rangle,$$
$$\mathbf{s_3} = \langle 2, 1, 3+4i \rangle, \ldots$$

Fig. 3 shows the visibility grids $\mathbf{V_2}$ and $\mathbf{V_3}$ for stream $S(\infty) = (\mathbf{s_1}, \mathbf{s_2}, \mathbf{s_3}, ...)$. For $t = 3$ we have $S^{2,1}(3) = (vis_1, vis_3)$. Thus, $\mathbf{V_3}[2,1] = (1+2i) + (3+4i) = 4+6i$.

u \ v	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	1+2i	0	0
3	0	0	-2+3i	0

(a) $\mathbf{V_2}$

u \ v	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	4+6i	0	0
3	0	0	-2+3i	0

(b) $\mathbf{V_3}$

Fig. 3. Visibility grid $\mathbf{V_2}$ at time $t = 2$ and $\mathbf{V_3}$ at time $t = 3$.

The sky image $\mathbf{I_t}$ is computed by applying the inverse Fourier transform $\boldsymbol{\mathcal{F}}$ to grid $\mathbf{V_t}$:

$$\mathbf{I_t} = \boldsymbol{\mathcal{F}}(\mathbf{V_t})$$

Problem Definition: Given a record $\mathbf{s_t}$ at time t in the stream, compute the image $\mathbf{I_t}$ with a linear number of complex multiplication operations and linearly reduce the time complexity with the degree of parallelism.

IV. Background

The direct computation of the Discrete Fourier Transform (DFT) is performed by computing the DFT for each cell of a 1D or 2D input grid. We write $\boldsymbol{\mathcal{F}}_{dft}(\mathbf{V_t})$ for the DFT operation applied to grid $\mathbf{V_t}$ and it is computed as [25]:

$$\mathbf{I_t} = \boldsymbol{\mathcal{F}}_{dft}(\mathbf{V_t}) = \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} \mathbf{V_t}[u,v] \cdot \mathbf{L^{u,v}} \quad (3)$$

The term $\mathbf{L}^{u,v}$ denotes an $N \times N$ matrix:

$$\mathbf{L^{u,v}} = \begin{bmatrix} l_{0,0} & l_{0,1} & \cdots & l_{0,N-1} \\ l_{1,0} & l_{1,1} & \cdots & l_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ l_{N-1,0} & l_{N-1,1} & \cdots & l_{N-1,N-1} \end{bmatrix}$$

that consists of complex numbers known as *twiddle factors* [26]. Twiddle factors are defined as:

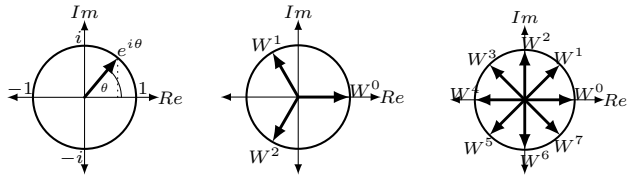
$$l_{j,k} = (e^{\frac{i2\pi}{N}})^{u \cdot j + v \cdot k}$$

Note that the values of matrix $\mathbf{L^{u,v}}$ depend on the value of (u, v). Thus, for each cell, we have a different $\mathbf{L^{u,v}}$ matrix.

Twiddle Factors: Let $W = e^{\frac{i2\pi}{N}}$. For any positive integer N, the twiddle factors are given by Euler's formula ($e^{i\theta} = cos(\theta) + isin(\theta)$) [27], i.e.:

$$W^m = e^{i\frac{2\pi m}{N}} = cos(\frac{2\pi m}{N}) + isin(\frac{2\pi m}{N}) \tag{4}$$

Thus, a twiddle factor is a complex number on the unit circle with $cos(\theta)$ as its real component and $sin(\theta)$ as its imaginary component. The N distinct twiddle factors are the N^{th} roots of unity and they are evenly spread across the unit circle as illustrated in Fig. 4(b) for $N = 3$ and in Fig. 4(c) for $N = 8$.



(a) Euler's formula (b) cube roots of unity (c) 8^{th} roots of unity

Fig. 4. Illustration of the complex-valued roots of unity.

The twiddle factors W^m are periodic because of the periodicity of sine and cosine functions, i.e., $W^{m+N} = W^m$ [28]. From the periodicity of the twiddle factors we get $W^{m\%N} = W^m$. Thus, for each twiddle factor we have:

$$l_{j,k} = W^{u \cdot j + v \cdot k} = W^{(u \cdot j + v \cdot k)\%N} \tag{5}$$

The periodicity of the twiddle factors restricts the number of possible values for the term $W^{(u \cdot j + v \cdot k)\%N}$ to N values: $\{W^0, W^1, ..., W^{N-1}\}$. Hence, the N^2 cells of the twiddle factor matrix $\mathbf{L}^{u,v}$ consists of at most N distinct twiddle factors. In Sec. V we prove that the computations involving twiddle factors can be reused in a systematic way to reduce the runtime complexity.

Incremental Fourier Transform: The visibility record $\mathbf{s_t}$ at time t updates a single cell of the visibility grid $\mathbf{V_t}$ as shown in (1). The Fourier transform of an updated cell and all the other cells can be written by replacing $\mathbf{V_t}[u, v]$ in (3) with the definition in (1). At all points where the coordinates (u, v) are different from (u_t, v_t), the visibility grid is the same as the one at the previous time instant; at point (u_t, v_t) the value is updated by adding vis_t:

$$\mathbf{I_t} = (\sum_{\substack{u=0 \\ u \neq u_t}}^{N-1} \sum_{\substack{v=0 \\ v \neq v_t}}^{N-1} \mathbf{V_{t-1}}[u, v] \cdot \mathbf{L^{u,v}}) + (\mathbf{V_{t-1}}[u_t, v_t] + vis_t) \cdot \mathbf{L^{u_t,v_t}}$$

The linearity property of the Fourier transform, i.e., $\boldsymbol{\mathcal{F}}(x + y) = \boldsymbol{\mathcal{F}}(x) + \boldsymbol{\mathcal{F}}(y)$ [29], can be used to rearrange the terms to get the incremental relation:

$$\mathbf{I_t} = \mathbf{I_{t-1}} + vis_t \cdot \mathbf{L^{u_t,v_t}} \tag{6}$$

Note that (6) no longer requires to maintain the visibility grid $\mathbf{V_t}$. For each new stream item $\mathbf{s_t}$, we compute $vis_t \cdot \mathbf{L^{u_t,v_t}}$ without updating grid $\mathbf{V_{t-1}}$. We call $vis_t \cdot \mathbf{L^{u,v}}$ the *Single Point Fourier Transform*. This is a scalar multiplication of a complex visibility value with matrix $\mathbf{L^{u,v}}$ and it requires N^2 complex multiplications. After computing the single point Fourier transform, the naive incremental update to image $\mathbf{I_{t-1}}$ requires N^2 complex addition operations. We address both issues in this paper. For the multiplications we introduce a circular shifting, while for the additions we divide the matrix into slices that can be processed independently and do not have to be combined.

These are the central elements for the incremental computation of Discrete Fourier transforms and sets the stage for our improvements of this basic approach in the forms of an observation about the twiddle factors in Section V, adaptations of the incremental approach and algorithmic considerations in Section VI, and the streaming pipeline in Section VII.

V. Circular shifts in the twiddle factor matrix

According to (5), row $\mathbf{L}^{u_t,v_t}[r, *]$ of matrix $\mathbf{L}^{u_t,v_t}$ is given as:

$$\mathbf{L}^{u_t,v_t}[r, *] = [W^{(r \cdot u_t + 0 \cdot v_t)\%N}, ..., W^{(r \cdot u_t + (N-1) v_t)\%N}] \tag{7}$$

Lemma 5.1: Consider an $N \times N$ twiddle factor matrix $\mathbf{L}^{u_t,v_t}$. Row r is a *p-circular shift* of row 0 if element 0 of row r exists at position p in row 0, i.e.:

$$\begin{aligned} \forall r, p, j (\mathbf{L}^{u_t,v_t}[r, 0] = \mathbf{L}^{u_t,v_t}[0, p] \Rightarrow \\ \mathbf{L}^{u_t,v_t}[r, j] = \mathbf{L}^{u_t,v_t}[0, (p + j)\%N]) \end{aligned} \tag{8}$$

Proof: Let $\mathbf{L}^{u_t,v_t}[r, *]$ be row r and $\mathbf{L}^{u_t,v_t}$[0,*] be row 0 of the twiddle factor matrix $\mathbf{L}^{u_t,v_t}$:

$$\mathbf{L}^{u_t,v_t}[0, *] = [W^{(0 \cdot v_t)\%N}, ..., W^{((N-1) \cdot v_t)\%N}]$$
$$\mathbf{L}^{u_t,v_t}[r, *] = [W^{(r \cdot u_t + 0 v_t)\%N}, ..., W^{(r \cdot u_t + (N-1) \cdot v_t)\%N}]$$

From the precondition $\mathbf{L}^{u_t,v_t}[r, 0] = \mathbf{L}^{u_t,v_t}[0, p]$ and (7) we directly get $W^{r \cdot u_t \%N} = W^{(p \cdot v_t)\%N}$. We have to show that for $0 \leq j < N$ the elements $\mathbf{L}^{u_t,v_t}[s, j]$ and $\mathbf{L}^{u_t,v_t}[0, (p+j)\%N]$ are equal:

$$\begin{aligned} \mathbf{L}^{u_t,v_t}[r, j] &= \mathbf{L}^{u_t,v_t}[0, (p + j)\%N] \\ W^{(r \cdot u_t + j \cdot v_t)\%N} &= W^{((p+j)v_t)\%N} \\ &= W^{(p \cdot v_t + j \cdot v_t)\%N} \\ &= W^{((p \cdot v_t)\%N + (j \cdot v_t)\%N)\%N} \\ &= W^{(p \cdot v_t)\%N} \cdot W^{(j \cdot v_t)\%N} \\ &= W^{(r \cdot u_t)\%N} \cdot W^{(j \cdot v_t)\%N} \\ &= W^{(r \cdot u_t + j \cdot v_t)\%N} \end{aligned}$$

Thus, if element 0 of row r exists in row 0, then row r is a circular shift of row 0. ∎

The same reasoning holds for columns. Any two consecutive elements in a column differ by W^{u_t}. The roles of u_t and v_t are switched in this case and it follows that column $\mathbf{L}^{u_t,v_t}[*,c]$ is a p-circular shift of column 0 if element 0 of column c exists in column 0.

Example 2: Consider row 0 and row 1 of the 8×8 twiddle factor matrix $\mathbf{L}^{2,3}$ in Table II:

$$\mathbf{L}^{2,3}[0,*] = [W^0, W^3, W^6, W^1, W^4, W^7, W^2, W^5]$$
$$\mathbf{L}^{2,3}[1,*] = [W^2, W^5, W^0, W^3, W^6, W^1, W^4, W^7]$$

Element 0 of row $\mathbf{L}^{2,3}[1,*]$ exists at position $p = 6$ of row 0, i.e., $\mathbf{L}^{2,3}[1,0] = \mathbf{L}^{2,3}[0,6]$. Thus, row 1 is a 6-circular shift of row 0, i.e., elements $\mathbf{L}^{2,3}[1,j]$ and $\mathbf{L}^{2,3}[0,(6+j)\%8]$ are equal for $0 \leq j < 8$.

TABLE II
THE 8×8 TWIDDLE FACTOR MATRIX $\mathbf{L}^{2,3}$.

r \ c	0	1	2	3	4	5	6	7
0	W^0	W^3	W^6	W^1	W^4	W^7	W^2	W^5
1	W^2	W^5	W^0	W^3	W^6	W^1	W^4	W^7
2	W^4	W^7	W^2	W^5	W^0	W^3	W^6	W^1
3	W^6	W^1	W^4	W^7	W^2	W^5	W^0	W^3
4	W^0	W^3	W^6	W^1	W^4	W^7	W^2	W^5
5	W^2	W^5	W^0	W^3	W^6	W^1	W^4	W^7
6	W^4	W^7	W^2	W^5	W^0	W^3	W^6	W^1
7	W^6	W^1	W^4	W^7	W^2	W^5	W^0	W^3

Lemma 5.2: Consider an $N \times N$ twiddle factor matrix $\mathbf{L}^{u_t,v_t}$. If a twiddle factor with exponent u_t modulo N, i.e., $W^{u_t\%N}$, exists in row 0 at index p, then the exponent in row 0 at position $(p \cdot r)\%N$ is equal to $W^{(r \cdot u_t)\%N}$:

$$\forall p, r(\mathbf{L}^{u_t,v_t}[0,p] = W^{u_t\%N} \Rightarrow \mathbf{L}^{u_t,v_t}[0,(p \cdot r)\%N] = W^{(r \cdot u_t)\%N}) \quad (9)$$

Proof: The j^{th} element of row 0 is $\mathbf{L}^{u_t,v_t}[0,j] = W^{(j \cdot v_t)\%N}$. We show that, since W^{u_t} exists at index p in row 0, i.e., $W^{u_t\%N} = W^{p \cdot v_t\%N}$, the twiddle factor $W^{r \cdot u_t\%N}$ exists at position $(p \cdot r)\%N$ in row 0:

$$\begin{aligned}\mathbf{L}^{u_t,v_t}[0,(p \cdot r)\%N] &= W^{(p \cdot r \cdot v_t)\%N} \\ &= W^{p\%N \cdot r\%N \cdot v_t\%N} \\ &= (W^{p\%N \cdot v_t\%N})^{r\%N} \\ &= (W^{(p \cdot v_t)\%N})^{r\%N} \\ &= (W^{u_t\%N})^{r\%N} \\ &= W^{r \cdot u_t\%N}\end{aligned}$$

∎

Similarly, if a twiddle factor with exponent v_t, i.e., W^{v_t}, exists in column 0 then the twiddle factor $W^{(c \cdot v_t)\%N}$ exists in column 0.

Example 3: Consider the 8×8 twiddle factor matrix $\mathbf{L}^{2,3}$ in Table II. Row 0 consists of the twiddle factors with exponents equal to multiples of 3 module 8. Column 0 consists of the twiddle factors with exponents equal to multiples of 2 modulo 8. Since, $W^{u_t\%N} = W^2$ exists at $p = 4$, all its multiples $W^{(r \cdot u_t)\%N}$ exist in row 0 and their index can be computed using (9). Since $W^{v_t} = W^3$ does not exist in column 0, it is not possible to compute the index of all its multiples $W^{(c \cdot 3)\%N}$ in column 0. For example, W^1, W^5 and W^7 do not exist in column 0 of Table II.

A. Row shiftable matrices

An $N \times N$ twiddle factor matrix $\mathbf{L}^{u_t,v_t}$ is row shiftable if each row is a circular shift of row 0.

Lemma 5.3: Consider an $N \times N$ twiddle factor matrix $\mathbf{L}^{u_t,v_t}$. Let N be a power of 2 (i.e., $N = 2^m, m > 0$) or N be a prime number. The twiddle factor matrix is *row shiftable* if $gcd(u_t, N) \geq gcd(v_t, N)$, where gcd is the *greatest common divisor*.

Proof: For a twiddle factor matrix $\mathbf{L}^{u_t,v_t}$, row 0 consists of twiddle factors with exponents equal to multiples of v_t modulo N:

$$\mathbf{L}^{u_t,v_t}[0,*] = [W^{(0 \cdot v_t)\%N}, ..., W^{((N-1) \cdot v_t)\%N}]$$

$gcd(v_t, N) = 1$ if row 0 consists of all N twiddle factors, i.e., $\mathbf{L}^{u_t,v_t}[0,*] = \{W^0, W^1, ..., W^{N-1}\}$. In this case, W^{u_t} exists in row 0 because $0 \leq u_t < N$. According to Lemma 5.2, all multiples of W^{u_t} also exist in the row 0, which are the elements of column 0 or element 0 of every row. Hence, each row is a circular shift of row 0 if $gcd(v_t, N) = 1$. Note that $gcd(v_t, N)$ is 1 if N is a prime number or v_t is an odd integer but N is a power of 2.

$gcd(v_t, N) > 1$ if both N and v_t are even integers. In this case, row 0 consists of twiddle factors with exponents that are equal to multiples of $gcd(v_t, N)$. Thus, the lower $gcd(v_t, N)$, the higher the number of twiddle factors in row 0. The $gcd(v_t, N)$ is 2 if v_t is not a power of 2. When $gcd(v_t, N)$ is 2, row 0 consists of the twiddle factors with the exponents as multiples of 2, which means all the even exponents from 0 to N-2, i.e., $\mathbf{L}^{u_t,v_t}[*,0] = \{W^0, W^2, ..., W^{N-2}\}$. Then, for any even u_t when $gcd(v_t, N)$ equals 2, W^{u_t} exists in row 0. Moreover, according to Lemma 5.2 all its multiples also exist in row 0, such that every row is a circular shift of row 0. However, when v_t is a power of 2, then $gcd(v_t, N) \geq 2$, and the row 0 consists of less twiddle factors with not all the even exponents. In such a case, W^{u_t} only exists in row 0 if the $gcd(u_t, N) \geq gcd(v_t, N)$. ∎

B. Column shiftable matrices

Similar to row shiftable matrices, the twiddle factor matrix is *column shiftable* if each column is a circular shift of column 0. Specifically, the twiddle factor matrix is *column shiftable* if $gcd(v_t, N) \geq gcd(u_t, N)$.

Example 4: Consider Table II with the 8×8 twiddle factor matrix $\mathbf{L}^{2,3}$ for coordinate $(u_t, v_t) = (2, 3)$. From $gcd(2, 8) > gcd(3, 8)$ and Lemma 5.3 it follows that $\mathbf{L}^{2,3}$ is *row shiftable*. Since $gcd(v_t, N) \not\geq gcd(u_t, N)$, the twiddle factor matrix $\mathbf{L}^{2,3}$ is not *column shiftable*.

C. Shiftable matrices

The twiddle factor matrix is *row* and *column shiftable* if u_t and v_t are odd integers and N be a power of 2. In this case, $gcd(u_t, N) = gcd(v_t, N) = 1$ and column 0 as well as row 0 include all the twiddle factors. If $gcd(u_t, N) < gcd(v_t, N)$, then the twiddle factor matrix is not *row shiftable*. If $gcd(v_t, N) < gcd(u_t, N)$, then the twiddle factor matrix is not *column shiftable*. Note that $gcd(u_t, N) > gcd(v_t, N)$, $gcd(u_t, N) = gcd(v_t, N)$, or $gcd(u_t, N) < gcd(v_t, N)$. Hence, if N is a prime number or a power of two, we conclude that for any coordinate u_t and v_t, either a row or a column shift exists for an $N \times N$ twiddle factor matrix $\mathbf{L}^{u_t,v_t}$.

D. Restriction of N to being power of 2 or prime

A circular shift is guaranteed to exist if the size N of 2D matrix is either prime or a power of 2. This is not an uncommon assumption in the application area. Looking at state-of-the-art libraries, GPUFFTW (a high performance FFT library for GPUs) and Intel IPP (specialized for multimedia and signal processing) are restricted to power of two transforms. Other libraries, like Intel MKL and FFTW3, accept any value of N as long as it is composite, i.e., expressed as a multiple of small prime numbers 2, 3, 5, 7, 11, 13. However, FFTW3 reports that transforms for sizes of 2 are much faster.

VI. The Single Point Incremental Fourier Transform Algorithm

This section introduces the Single Point Incremental Fourier Transform (SPIFT) Algorithm based on circular shifts.

A. Setting the stage: Single Point Fourier Transform (SPFT)

The Single Point Fourier Transform $vis_t \cdot \mathbf{L}^{u_t,v_t}$ of a single record $\mathbf{s_t}$ can be computed efficiently by shifting a row of the twiddle factor matrix $\mathbf{L}^{u_t,v_t}$. First, row 0 of the twiddle factor matrix is computed:

$$\mathbf{L}^{u_t,v_t}[0,*] = [W^{(0\cdot v_t)\%N}, ..., W^{((N-1)v_t)\%N}]$$

Next, shift index p is computed such that $W^{u_t} = \mathbf{L}^{u_t,v_t}[0,p]$. Finally, for each row r of a twiddle factor matrix $vis_t \cdot \mathbf{L}^{u_t,v_t}$ is computed as a circular shift of row 0 using (8) and (9):

$$vis_t \cdot \mathbf{L}^{u_t,v_t}[r,j] = vis_t \cdot \mathbf{L}^{u_t,v_t}[0,(p \cdot r + j)\%N] \quad (10)$$

At time instant t, image $\mathbf{I_t}$ can be computed using (6). Since each row or column of $vis_t \cdot \mathbf{L}^{u_t,v_t}$ is a circular shift of row 0 or column 0, the computation of $\mathbf{L}^{u_t,v_t}$ can be avoided, and the incremental update is applied directly to image $\mathbf{I_{t-1}}$ using $\mathbf{L}^{u_t,v_t}[0,*]$ or $\mathbf{L}^{u_t,v_t}[*,0]$. For a row shiftable matrix $\mathbf{I_t}$ can be computed using (6) and (10) for all $k = 0, ..., N-1$ as:

$$\mathbf{I_t}[x,k] = \mathbf{I_{t-1}}[x,k] + vis_t \cdot \mathbf{L}^{u_t,v_t}[0,(p \cdot x + k)\%N] \quad (11)$$

If the matrix $\mathbf{L}^{u_t,v_t}$ is *row shiftable*, each row of $\mathbf{I_{t-1}}$ is updated directly using a circular shift of row 0 of the twiddle factor matrix. Similarly, each column of $\mathbf{I_{t-1}}$ will be updated directly using column 0 of the twiddle factor matrix if the matrix is *column shiftable*.

The vector $vis_t \cdot \mathbf{L}^{u_t,v_t}[0,*]$ or $vis_t \cdot \mathbf{L}^{u_t,v_t}[*,0]$ is computed only once and requires N complex multiplication operations.

B. Key-Based Partitioning

We use key-based partitioning to distribute image $\mathbf{I_t}$ to parallel task instances. Each parallel task instance updates a slice of size $N \times \frac{N}{d}$, where d is the number of parallel task instances.

If the twiddle factor matrix is *row shiftable*, then row j of sub grid $\mathbf{I_{t,key}}$ can be computed as:

$$\mathbf{I_{t,key}}[j,k] = \mathbf{I_{t-1,key}}[j,k] + vis_t \cdot \mathbf{L}^{u_t,v_t}[0,(p(j + (key \times rows)) + k)\%N] \quad (12)$$

Here $rows = N/d, j = 0, ..., rows - 1$ and $k = 0, ..., N-1$. Similarly, if the twiddle factor matrix is *column shiftable*, then column k of sub grid $\mathbf{I_{t,key}}$ can be computed for all $j = 0, ..., rows - 1$ as:

$$\mathbf{I_{t,key}}[j,k] = \mathbf{I_{t-1,key}}[j,k] + vis_t \cdot \mathbf{L}^{u_t,v_t}[((key \times rows) + j + (p \times k))\%N, 0] \quad (13)$$

C. The SPIFT Algorithm

The SPIFT algorithm in Algorithm 1 first determines if the twiddle factor matrix is *row shiftable* or *column shiftable* for coordinates (u_t, v_t). As shown earlier, if the condition for a row shift is not satisfied, then the condition for a column shift is satisfied (line **1**). The shift type is used to compute the shift index (p) and the shift vector ($\mathbf{q}$). If the matrix is *column shiftable*, the shift vector ($\mathbf{q}$) represents the first column (i.e., $vis_t \cdot \mathbf{L}^{u_t,v_t}[*,0]$) of the twiddle factor matrix given in (10). It is computed by multiplying the complex value vis_t with the twiddle factors such that the exponents of the twiddle factors are modulo N of multiples of the coordinate u_t (line **5**). If the matrix is *row shiftable*, the shift vector ($\mathbf{q}$) represents the first row (i.e., $vis_t \cdot \mathbf{L}^{u_t,v_t}[0,*]$) of the twiddle factor matrix given in (10). It is computed by multiplying the complex value vis_t with the twiddle factors such that the exponents of twiddle factors are modulo N of multiples of the coordinate v_t (line **7**). This computation requires $O(N)$ complex multiplications and are the only complex multiplication operations in our algorithm. Finally, Incremental Update is performed using shift type, shift index and computed vector.

Algorithm 1: SPIFT (u_t, v_t, vis_t, N, key, d)

```
1  isCS ← (v_t = 0) or
          (u_t%2 = 1 and v_t%2 = 0) or
          (v_t%2 = 0 and gcd(u_t, N) < gcd(v_t, N));
2  p ← ShiftIndex(u_t, v_t, isCS);
3  for k ← 0 to N do
4      if (isCS) then
5          q[k] ← vis_t · W^{k·u_t%N}
6      else
7          q[k] ← vis_t · W^{k·v_t%N}
8  I_{t,key} ← IncUpdate(I_{t-1,key}, q, isCS, p, N, key, d);
9  return I_{t,key}
```

ShiftIndex: After computing the type of shift, the shift index is computed using Algorithm 2. The shift index p is determined by matching the coordinate u_t with the modulo of multiple of coordinate v_t and vice versa. The lines **3-4** in Algorithm 2 find the shift index in case the matrix is *column*

shiftable whereas lines **6-7** determine the shift index in case the matrix is *row shiftable*.

Algorithm 2: ShiftIndex(u_t, v_t, isCS)

```
1  if (u_t = 0 or v_t = 0) then return 0;
2  if (isCS) then
3  |   for j ← 0 to N do
4  |   |   if (v_t = j u_t %N) then return j;
5  else
6  |   for k ← 0 to N do
7  |   |   if (u_t = k v_t %N) then return k;
```

IncrementalUpdate: Algorithm 3 computes an image slice $\mathbf{I_{t,key}}$. It directly updates an image slice $\mathbf{I_{t-1,key}}$ by taking as input the partition key, type of shift, the shift index and the vector. If the matrix is column shiftable, the incremental transform first computes the starting index $startidx$ of the shift vector $\mathbf{q}$ according to (13), for the column k of image slice in line **4**. Then it iterates over each row j in line **6** and computes its index. Finally, in line **7**, the algorithm updates one cell of the previous image slice. Similarly, lines **9-13** compute the incremental Fourier transform for row shiftable matrices.

Algorithm 3: IncUpdate($\mathbf{I_{t-1,key}}, \mathbf{q}, isCS, p, N, key, d$)

```
1  rows ← N/d ;
2  if (isCS) then
3  |   for k ← 0 to N do
4  |   |   startidx ← ((key × rows) + (p × k))%N ;
5  |   |   for j ← 0 to rows do
6  |   |   |   idx ← (startIdx + j)%N ;
7  |   |   |   I_{t,key}[j,k] ← I_{t-1,key}[j,k] + q[idx]
8  else
9  |   for j ← 0 to rows do
10 |   |   startidx ← (p(j + (key × rows)))%N ;
11 |   |   for k ← 0 to N do
12 |   |   |   idx ← (startIdx + k)%N ;
13 |   |   |   I_{t,key}[j,k] ← I_{t-1,key}[j,k] + q[idx]
14 return I_{t,key}
```

Our implementation does not actually shift the elements of vector $vis_t \cdot \mathbf{L}^{u_t,v_t}[0,*]$ or $vis_t \cdot \mathbf{L}^{u_t,v_t}[*,0]$, but manages the elements in a circular array. That is why we compute a starting index $startIdx$ of the vector $vis_t \cdot \mathbf{L}^{u_t,v_t}[0,*]$ or $vis_t \cdot \mathbf{L}^{u_t,v_t}[*,0]$ to update the j^{th} row or k^{th} column of $\mathbf{I_{t-1,key}}$ respectively.

Example 5: The DFT of a record $\mathbf{s_1} = \langle 2, 1, 1{+}2i\rangle$ in Example 1 is computed using SPIFT. First, SPIFT determines the shift type for the coordinates (2,1). The twiddle factor matrix $\mathbf{L}^{2,1}$ is *row shiftable* because $gcd(2,4) > gcd(1,4)$. Algorithm 2 returns shift index 2 because condition $u_t = kv_t\%N$ is true for $k = 2$, which means that twiddle factor W^2 exists at index 2 in row 0. The shift vector $q = [(1{+}2i)\cdot W^0, (1{+}2i)\cdot W^1, (1{+}2i)\cdot W^2, (1{+}2i)\cdot W^3] = [1{+}2i, -2{+}i, -1{-}2i, 2{-}i]$ is computed by multiplying the complex value $1+2i$ with the twiddle factors. Algorithm 3 computes $\mathbf{I_1}$ (Fig. 5) by adding the shift vector $\mathbf{q}$ to each row of the image $\mathbf{I_0}$.

D. Complexity Analysis of SPIFT, FFT and direct DFT

The asymptotic complexity of the single point Fourier transform (i.e., the updates that must be applied to the image) using circular shifts is $O(N)$, as it requires N complex multiplication operations as discussed in Section VI-A. The asymptotic complexity of the entire SPIFT algorithm is $O(N^2 + N) = O(N^2)$. The direct DFT approach in (3) requires $2N^2$ complex multiplication operations followed by N^2 complex addition operations, with an overall complexity of $O(N^2)$ for a single point Fourier transform. 2D-FFT has an asymptotic complexity of $O(N^2 \log N)$ for computing the Fourier transform for a single point update. Our experiments show that there are significant runtime differences between SPIFT and direct DFT. The asymptotic notation is able to show the differences for the multiplications but not the additions.

u \ v	0	1	2	3
0	1+2i	-2+i	-1-2i	2-i
1	-1-2i	2-i	1+2i	-2+i
2	1+2i	-2+i	-1-2i	2-i
3	-1-2i	2-i	1+2i	-2+i

Fig. 5. Image $\mathbf{I_1}$ at time $t = 1$

VII. The SPIFT Streaming Pipeline

In this section we discuss the streaming pipeline and the data flow for the SPIFT algorithm.

A typical streaming pipeline consists of input sources, output sinks and the transformations applied to the stream items. The transformations are performed by using specific transformation operators with a user-defined or built-in transformation logic. The output of an operator serves as input for the next transformation operator in the pipeline. Depending upon the use case the transformation operator must remember partial results for the records arriving later in the stream. Results or data that is remembered for future computation is called State and transformation on such data is called *stateful transformations* in stream processing.

Fig. 6 shows the streaming pipeline that computes the incremental Fourier transform using SPIFT. The pipeline is divided into three stages, as described below.

Computing ShiftType, ShiftIndex and Partitioning: The pipeline first reads the source stream of visibility records of the form $\mathbf{s_t}$ and processes each record independently to incrementally compute the sub grids. The first stage includes three stateless transformations and a partition operator. The first transformation operator enriches a stream item $\mathbf{s_t}$ with a bit indicating whether the twiddle factor matrix for this observation is *row shiftable* or *column shiftable*. The second transformation operator transforms the data element $\langle \mathbf{s_t}, 0/1\rangle$ by appending the shift index p, which is determined with Algorithm 2. The third operator transforms the data element $\langle \mathbf{s_t}, 0/1, p, \rangle$ into d elements by making d copies, where d is the degree of parallelism. A unique key is prepended to each copy, i.e., $\langle \mathbf{s_t}, 0/1, p\rangle \Rightarrow [\langle 0, \mathbf{s_t}, 0/1, p\rangle, \langle 1, \mathbf{s_t}, 0/1, p\rangle, .., \langle d -$

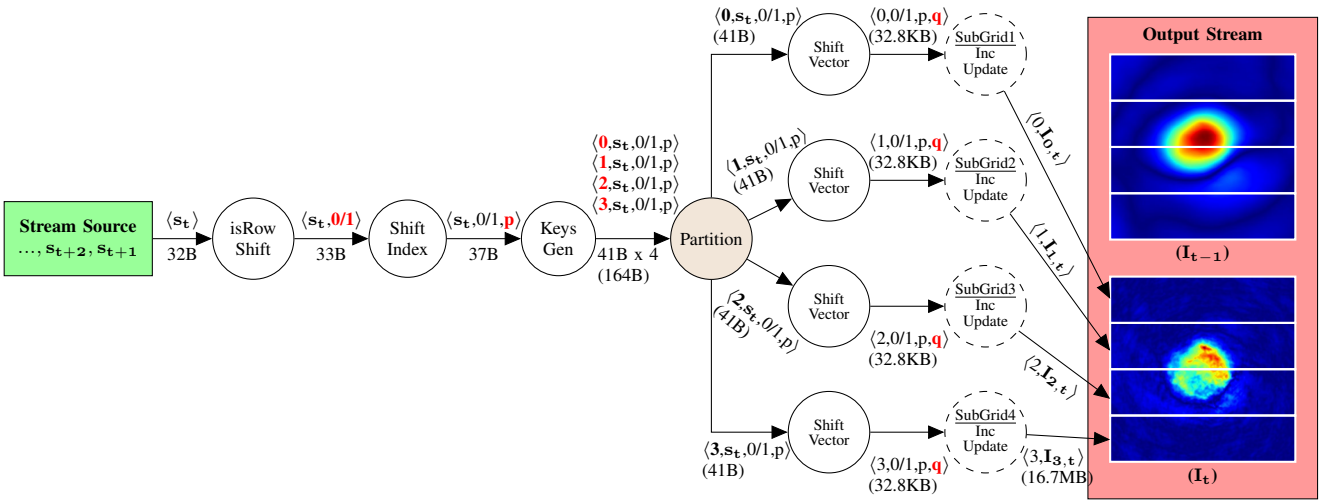


Fig. 6. The SPIFT Streaming Pipeline with data flow and data size between operators for a 2048×2048 grid. Transformation operators are shown as circles. A solid line is used for stateless transformation operators. A dashed line is used for stateful transformation operators; the underlined text denotes the *State*.

$1, \mathbf{s_t}, 0/1, p\rangle]$. The fourth operator is a partition operator, which uses hash-based partition to group the records with the same key. In the SPIFT pipeline, the number of unique keys for each update $\mathbf{s_t}$ should be equal to the parallelism degree of the incremental update operator to ensure that each image slice is updated independently.

Shift Vector Computation and Parallel Incremental Addition on Sub Grids: The second stage computes a shift vector, and performs an incremental addition. Both the shift vector computation and incremental addition are executed in parallel in the SPIFT pipeline. At time t, the subgrid $\mathbf{I_{key,t}}$ represents the Fourier transform computed from time 0 to t. First, a stateless transformation computes a shift vector by transforming data element $\langle key, \mathbf{s_t}, 0/1, p\rangle$ to $\langle key, 0/1, p, \mathbf{q}\rangle$. The second opreator is a stateful transformation that stores the subgrid $\mathbf{I_{key,t}}$ as its state. Each parallel instance of such stateful transformation takes $\langle key, 0/1, p, \mathbf{q}\rangle$ as input, updates the corresponding subgrid $\mathbf{I_{key,t}}$ by performing the incremental additions using Algorithm 3, and stores the result for subsequent updates.

Although the shift vector $\mathbf{q}$ is the same for each incremental update, it is computed by every parallel operator. The reason is the key-based partitioning in the pipeline, which creates d copies of $\langle \mathbf{s_t}, 0/1, p\rangle$ as illustrated for $d = 4$ in Fig. 6. If one computes the shift vector before the partitioning, then it creates d copies of the shift vector q. This is a bottleneck that may reduce the throughput. Our experiments for grid size 2048×2048 show that the throughput of the pipeline increased until parallelism degree 16 and drops by 200% for parallelism degree 32. At the end, the throughput for a parallelism of 256 was even less than for parallelism degree 1. With the SPIFT pipeline in Fig. 6 the throughput increases linearly with the number of parallel threads.

Sub Grid Image Stream: After an incremental update, each parallel instance produces an updated sub grid $\mathbf{I_{t,key}}$ along with its partition key. In the third and last stage, the pipeline produces a stream of sub grids that can be consumed by an end user.

SPIFT implementation in Apache Flink: We implemented the SPIFT streaming pipeline in Apache Flink [4], which is a platform for distributed stream processing. We selected Apache Flink as it is one of the most widely adopted streaming platform.

The SPIFT streaming pipeline code written in Java with lambda expressions for Apache Flink is shown in Listing 1. The `map`, `flatMap` are the transformation operators. The elements on the left side of an arrow represent the input whereas the transformation/output is represented on the right side of an arrow. The `keyBy` operator is the partition operator and parameter 0 indicates that the partition was performed on the basis of first field of the input. Function `IncUpdate` uses the `AggregatingState` provided by Apache Flink, to store sub grid $\mathbf{I_{key}}$ as state. The `setParallelism(d)` operator runs d parallel instances of the transformation operator. The `addSink` operator is used to specify the output sink.

```
S.map(s->Tuple2.of(s,isColumnShift(s)))
 .map((s,isCS)->
     Tuple3.of(s,isCS,ShiftIndex(s,isCS)))
 .flatMap(s,isCS,p->KeysGen(s,isCS,p))
 .keyBy(0)
 .map((key,s,isCS,p)->
    Tuple4.of(key,isCS,p,ComputeVector(s,isCS))
 .setParallelism(d)
 .flatMap((key,isCS,p,q)->IncUpdate(key,isCS,p,q))
 .setParallelism(d)
 .addSink(key,I_{key,t})
```

Listing 1. SPIFT streaming pipeline Java code with lambda expressions for Apache Flink

Transformation operators that are equivalent to the ones used in Listing 1, i.e., `map`, `flatMap`, `keyBy`, `setParallelism`, and `addSink` are available in other distributed stream processing frameworks such as Apache Spark

[2] and Apache Storm. Hence, the SPIFT streaming pipeline in Fig. 6 can easily be implemented in other frameworks.

VIII. Performance Evaluation

This section evaluates the benefits of circular shifting and compares SPIFT with baseline algorithms as well as state-of-the-art hardware optimized libraries.

A. Experimental Setup

Metrics: We are interested in evaluating: (1) the time performance of using shifts for computing a single point Fourier transform, (2) the time performance of SPIFT compared to the direct 2D discrete Fourier transform, the Cooley-Tukey 2D fast Fourier transform, and hardware optimized FFT libraries, (3) the scalability of SPIFT, (4) the average throughput per second, and (5) the accuracy of SPIFT. We vary the grid size and the number of threads.

Datasets: As a real world dataset we use observations taken on 2019-11-24 for the project *Evolutionary Map of the Universe (EMU)* [30]. We use the dataset of beam 20.[1] We set grid sizes between 512 and 8192, which are typical values used in radio astronomy. We vary the number of threads between 1 and 256. We also consider a synthetic data stream with uniform distributions.[2] The script generates each stream item by sampling four values from uniform distributions: u, v are sampled from $\mathcal{U}(0, N)$ and vis_{real}, vis_{img} are sampled from $\mathcal{U}(-10000, 10000)$, where vis_{real} and vis_{img} are the real and imaginary components of the visibility. The experiments show that the performance of SPIFT and DFT and FFT algorithms is independent of the datasets and we usually report the numbers for the EMU dataset.

Algorithms: We compare SPIFT with the direct DFT computation for a single point, as well as with an in-place implementation of a radix-2 FFT algorithm. We also compare the performance of SPIFT with state-of-the-art hardware optimized libraries: FFTW3 [31] and Intel MKL [3]. The parameters used for computing the transform with FFTW3 and Intel MKL were: Backward Transform, Complex-to-Complex, Out-of-place, and Double precision. We also compare our parallel implementation of SPIFT with the parallel implementations of the FFTW3 and Intel MKL libraries.

Hardware: We use a machine with two Intel E5-2680 v2 2.80GHz processors and 128 GB RAM. Each processor has 10 cores with two threads per core. To evaluate the scalability of our algorithm, we used a cluster of eight such machines connected through a Gigabit-Ethernet network. Apache Flink chooses one node as job manager to serve as master node for distributing and scheduling tasks among slave nodes and the remaining nodes as task executors to provide task slots for parallel processing. We allocated 6GB memory to the job manager JVM and 12 GB to each of the task executor JVM.

[1]https://gitlab.ifi.uzh.ch/Saad/spift/, file scienceData_SB10635_EMU_2205-51.beam20_averaged_cal.ms

[2]https://gitlab.ifi.uzh.ch/Saad/spift/, file pyGen.py

[3]https://software.intel.com/mkl

B. Runtime Gain from Circular Shifting

This experiment investigates the time performance of the single point Fourier transform ($vis_t \cdot \mathbf{L}^{u_t,v_t}$) with and without shifts. Fig. 7 shows that our approach with shifts performs better as the grid size increases: it is approximately 5 times faster for smaller grid sizes (i.e., 512, 1024 and 2048), 6 times faster when the grid size is 4096, and 12 times faster for grid size 8192. The experiment confirms the analytical analysis: the single point Fourier transform requires N complex multiplication operations when using shifts and a quadratic number of complex multiplication operations when shifts are not used.

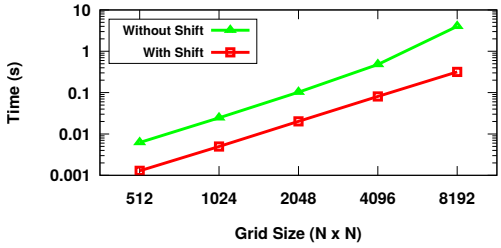


Fig. 7. Time Performance of Single Point Fourier Transform computed with and without shifts.

C. Runtime Comparison of SPIFT, direct DFT, and FFT

In this experiment, we study the time performance of SPIFT by comparing it with FFT and DFT, as well as with hardware-optimized libraries that implement FFT. Fig. 8 shows the time it takes for the algorithms to compute the Fourier transform of a 2D-grid after a new record has arrived in the stream. Analytically, the total number of floating operations for a single point is $10N^2 \log N$ for FFT, $14N^2$ for direct DFT and $2N^2 + 2N$ for SPIFT.

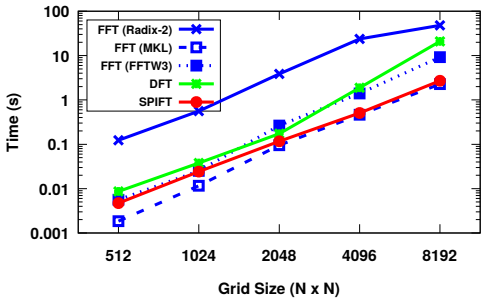


Fig. 8. Runtime Comparison of Single Point Incremental with Direct Discrete and Fast Fourier transform and hardware optimized FFTW3 and Intel MKL.

We measure the performance of the five algorithms when the grid size varies from 512×512 to 8192×8192. SPIFT performs 1.8 to 7.7 times faster than the direct DFT computation of a single point. The performance gain of SPIFT over DFT increases as the grid size increases. SPIFT performs 17 to 46 times faster than FFT. Fig. 8 shows that SPIFT is competitive with mature libraries even without advanced optimizations. These optimizations are orthogonal to the circular

shifting we propose in this paper. The differences between the solid and the dotted/dashed blue lines i.e. FFT with Radix-2 vs. FFT-MKL/FFTW3 in Fig. 8 illustrate the potential of applying optimizations that have been developed for mature FFT libraries. We think that some of these optimizations could be applied to SPIFT resulting in an even better performance.

D. Break Down of the Runtime of SPIFT

This experiment investigates the main steps of the SPIFT algorithm: the vector computation and the incremental addition and highlight the possibility of optimizing the latter with paralellization. As shown in Fig. 9, the shift vector computation takes only about 1% of the total time for computing SPIFT. This operation includes the time for finding the shift type, shift index, and N complex multiplication operations of twiddle factors with the complex value. The incremental addition, which takes up the bulk of the time, is trivially parallelizable and can be split up without overhead. The image is divided into slices, which are processed independently. As Fig. 9 highlights for a parallelism degree of $d = 256$ processes, the time for incremental addition can be reduced to roughly the time required for shift vector computation.

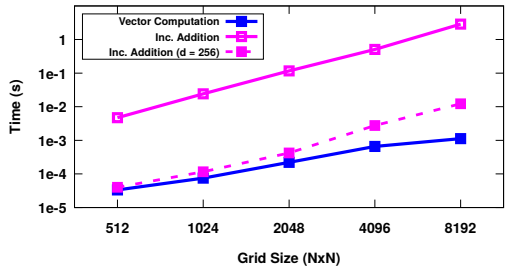


Fig. 9. Runtime of Shift Vector Computation and Incremental Addition operation (without and with a parallization degree of $d = 256$) in the SPIFT algorithm.

Both operations are affected by the grid size, which explains why the execution time increases when the grid size increases. The incremental addition with a quadratic complexity is a bottleneck in a streaming environment. As discussed in Section VIII-E, we overcome this bottleneck by performing additions in parallel, which we study in the next experiment.

E. Scalability of SPIFT

In this experiment we compare SPIFT with parallel implementations of FFT of modern libraries. For the hardware-optimized libraries, due to the non-trivial challenge required of setting up an environment for using MPI routines in a distributed setting, we only perform experiments with the multi-threaded routines on SMP. Hence, we perform FFTW3 and Intel MKL experiments on a single machine from the cluster, as detailed in Section VIII-A. For SPIFT, which runs on top of Apache Flink, two machines were used, one as a job manager and the other as a task executor. Since, each machine has 40 cores, we consider parallelism degree up to 32.

Fig. 10 shows the time performance of SPIFT when the parallelism ranges between 2 and 32 for different grid sizes. The graphs confirm that computing the additive operations through parallelism reduces the time complexity. The multi-threaded SPIFT outperforms parallel FFTW3 for the considered parallelism degrees. The performance gain for SPIFT increases with an increase of the parallelism because SPIFT does not have any parallel overhead since the incremental additions are performed independently over the disjoint slices/-subgrids with no communication required between tasks. The performance gain is 1.25 to 4.95 times with an increase of the parallelism for grid sizes 512×512 and 1024×1024. The performance gain increases from 3 to 31 times, 2 to 21 times, and 4 to 10 times for grid size 2048×2048, 4096×4096, and 8192×8192 respectively. SPIFT outperforms Intel MKL up to 2 to 3 times for parallelism degree greater or equal to 8 except for grid size 8192×8192. SPIFT only outperforms Intel MKL at parallelism degree 32 for grid size 8192×8192. The parallel multi-dimensional FFT becomes relatively slower with an increase of the parallelism, because it is computed by taking 1D-Fourier transforms across rows or columns and by dividing them equally among processors. Although, the rows or columns are transformed by different processors, they are still interleaved in memory. This causes memory contention, which increases with the increase in parallelism and results in a decrease of the performance. [4]

F. Throughput

In this experiment, we evaluate the throughput of the streaming pipeline. For this experiment, we executed the streaming pipeline in a cluster of eight machines, as detailed in Section VIII-A. Fig. 11 shows the throughput, i.e., the number of single point updates processed per second by SPIFT. The throughput in a streaming pipeline is dictated by the slowest task in the pipeline. The slowest task in the SPIFT streaming pipeline are the quadratic number of additions. By partitioning the image and computing the additions in parallel we get a linear speedup with the degree of parallelization. As the execution time decreases, more updates can be processed leading to an increase of the throughput as the degree of parallelism increases. This also confirms the advantage of exploiting parallelization for SPIFT.

In an interferometer array, the pairs of antennas produce a single visibility value every constant interval of time known as integration time. The ASKAP array has 36 antennas with integration time of 5 secs. The 36 antennas forms 630 unique pairs, with the addition of 36 self-pairs, resulting in 666 visibility values per beam per frequency channel [32] every 5 seconds. In our setting, SPIFT can handle a 512×512 grid for this update rate with parallelism degree 1. Increasing the parallelism, SPIFT can handle grid sizes of up to 4096. Dealing with grid size 8192 requires a parallelism degree larger than 256.

[4] http://www.fftw.org/parallel/parallel-fftw.html

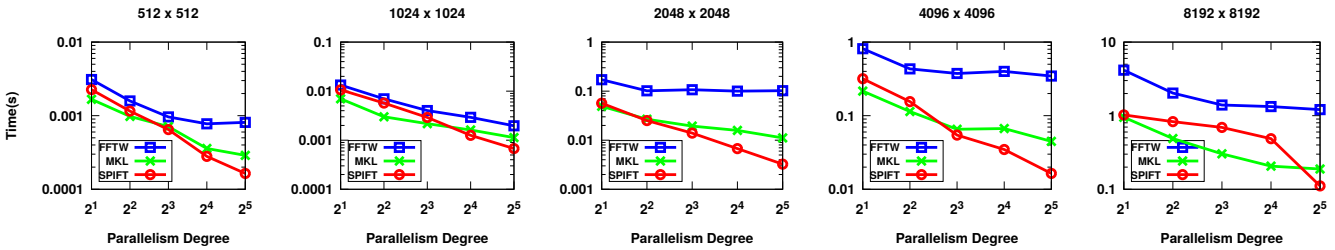


Fig. 10. Runtime comparison of a parallel SPIFT, parallel FFTW3 and parallel Intel MKL performed on a SMP hardware.

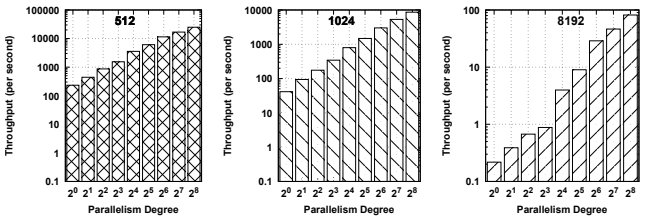


Fig. 11. Average Throughput of SPIFT with Apache Flink.

G. Varying Update Rate effect on the Throughput

In this experiment, we computed the throughput for all considered grid sizes with parallelism degree 8. We study the throughput over update rates 100 and 1000. We controlled the update rate using the ThrottledIterator provided by Apache Flink over the input source. The graph in Fig. 12 shows throughput for an average input rate of 100 and 1000 updates per second. The graph also shows the maximum attainable throughput of the SPIFT pipeline with the consumption rate decided by Apache Flink without our control (as reported in Section VIII-F). We see that the throughput is equal to the update rate, whenever the update rate is less than the maximum throughput. If the update rate is higher than the maximum achievable throughput, Apache Flink throttles the throughput so that the update rate does not surpass the maximum attainable throughput. For example, the maximum attainable throughput for grid 512×512 is 3571, hence, with an update rate of 100 and 1000 records the average throughput remains 100 and 1000 records per second. In contrast, the maximum attainable throughput for grid size 1024×1024 is 795. Thus, the input rate of 1000 is throttled to 795.

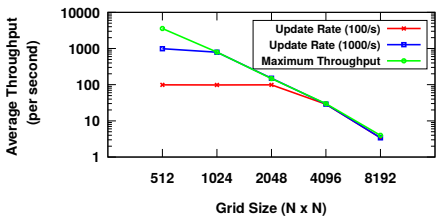


Fig. 12. Throughput of SPIFT at different update rates.

The relation between input rate and throughput is not surprising. In a streaming pipeline, the maximum attained throughput and the rate at which pipeline reads from the input source are dictated by the slowest task in the pipeline, which is the incremental addition in the SPIFT pipeline. Backpressure occurs at the key map operator, because the task partition operator produces data faster than the downstream shift vector operator can consume. Apache Flink deals with backpressure by propagating it upstream to one of its tasks, eventually slowing down the emission rate of the source.

H. Accuracy of SPIFT

This experiment assesses the accuracy of incremental Fourier transform. We consider the EMU and the synthetic datasets with 100K updates, and we compare the Fourier transform of each dataset computed with SPIFT and the one computed with a non-incremental FFT of the FFTW3 library. We evaluate the accuracy by measuring the root mean square error of both transforms and report the results in Fig. 13. The root mean square errors are of the order of 10^{-6} for the EMU dataset and 10^{-10} for the synthetic dataset. A low root mean square error suggests that incremental the Fourier transform is comparably as accurate as the non-incremental one. The graph also shows that SPIFT remains accurate as well as stable even with the increase in transform size.

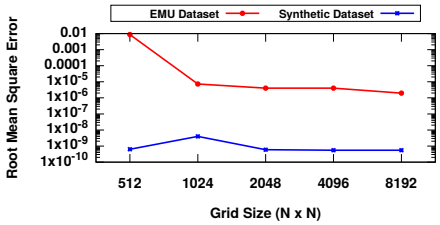


Fig. 13. Root Mean Square errors of SPIFT and FFTW3.

IX. Conclusion

In this paper, we have proposed an incremental algorithm for computing Fourier transform with each record in the stream. The proposed algorithm requires the least number of floating point operations for computing a Fourier transform of a single point when compared to state of the art algorithms. The main technique of our algorithm is to reduce the complex multiplication operations by reusing the computations of twiddle

factors for a 2D-grid of size $N \times N$, where N is power of two or N is prime. Our evaluations showed that the number of floating point operations increases rapidly when grid size increases for `FFT` and `direct DFT`; more than for our proposed `SPIFT` algorithm. We have reduced the number of complex multiplication operations but the time complexity of additive operations is reduced by adding parallelism. `SPIFT` is scalable by increasing parallelism and is ideal for larger grid sizes.

We developed this research in the radio astronomy context, but `SPIFT` may find application in other fields where data is collected continuously and the streaming requirements emerge. For example, in medicine, real-time MRI seems to work similarly to the radio astronomy process described in this article. In our future work, we will better study such problems to properly understand if `SPIFT` can be applied out of the box. We will also study the deployment of `SPIFT` on GPUs.

Acknowledgments: There are a number of people and organization we would like to thank: the Swiss National Science Foundation for the partial support under contract number #407550_167177; Bärbel Koribalski, Keith Bannister and Wasim Raja from the Australia Telescope National Facility (ATNF) for the helpful insights while studying the radio imaging domain; Nicolas Spielmann who has implemented and evaluated the single point Fourier transform with circular shifts in his Bachelor's thesis [33]; and the anonymous reviewers for their constructive and valuable comments.

REFERENCES

[1] X. Cai, L. Pratley, and J. D. McEwen, "Online radio interferometric imaging: assimilating and discarding visibilities on arrival," *Monthly Notices of the Royal Astronomical Society*, vol. 485, no. 4, pp. 4559–4572, 03 2019.

[2] Z. Matei, C. Mosharaf, F. Michael J., S. Scott, and S. Ion, "Spark: cluster computing with working sets," *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010.

[3] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1792–1803, Aug. 2015.

[4] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink : Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[5] K. Jay, N. Neha, and R. Jun, "Kafka: A distributed messaging system for log processing," *In Proceedings of the NetDB.1–7*, 2011.

[6] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: Stateful scalable stream processing at linkedin," *Proc. VLDB Endow.*, vol. 10, no. 12, pp. 1634–1645, Aug. 2017.

[7] A. Biem, B. Elmegreen, O. Verscheure, D. Turaga, H. Andrade, and T. Cornwell, "A streaming approach to radio astronomy imaging," in *2010 IEEE International Conference on Acoustics, Speech and Signal Processing*, March 2010, pp. 1654–1657.

[8] M. Mahmoud, A. Ensor, A. Biem, B. Elmegreen, and S. Gulyaev, "Data provenance and management in radio astronomy: A stream computing approach," *Studies in Computational Intelligence*, vol. 426, 12 2011.

[9] C. Ballard and I. B. M. Corporation, *IBM Infosphere Streams Harnessing Data in Motion*, ser. IBM redbooks. Vervante, 2010.

[10] W. Cochran, J. Cooley, D. Favin, H. Helms, R. Kaenel, W. Lang, G. Maling, D. Nelson, C. Rader, and P. Welch, "What is the fast fourier transform?" *IEEE Transactions on Audio and Electroacoustics*, vol. 15, no. 2, pp. 45–55, June 1967.

[11] P. Duhamel and H. Hollmann, "'split radix' fft algorithm," *Electronics Letters*, vol. 20, no. 1, pp. 14–16, 1984.

[12] S. G. Johnson and M. Frigo, "A modified split-radix fft with fewer arithmetic operations," *IEEE Transactions on Signal Processing*, vol. 55, no. 1, pp. 111–119, Jan 2007.

[13] B. Sherlock, "Windowed discrete fourier transform for shifting data," *Signal Processing*, vol. 74, no. 2, pp. 169 – 177, 1999.

[14] B. G. Sherlock and D. M. Monro, "Moving discrete fourier transform," *IEE Proceedings F - Radar and Signal Processing*, vol. 139, no. 4, pp. 279–282, Aug 1992.

[15] E. Jacobsen and R. Lyons, "The sliding dft," *IEEE Signal Processing Magazine*, vol. 20, no. 2, pp. 74–80, March 2003.

[16] E. Jacobsen and R. Lyons, "An update to the sliding dft," *IEEE Signal Processing Magazine*, vol. 21, no. 1, pp. 110–111, Jan 2004.

[17] K. Duda, "Accurate, Guaranteed Stable, Sliding Discrete Fourier Transform [DSP Tips & Tricks]," *IEEE Signal Processing Magazine*, vol. 27, no. 6, pp. 124–127, Nov 2010.

[18] C. Park, "Fast, Accurate, and Guaranteed Stable Sliding Discrete Fourier Transform [sp Tips & Tricks]," *IEEE Signal Processing Magazine*, vol. 32, no. 4, pp. 145–156, July 2015.

[19] X. Yu, X. Dong, G. Yu, Y. Qin, D. Yue, and Y. Zhao, "Botnet detection based on incremental discrete fourier transform," *JNW*, vol. 5, pp. 568–576, 2010.

[20] C. Park and S. Ko, "The Hopping Discrete Fourier Transform [sp Tips & Tricks]," *IEEE Signal Processing Magazine*, vol. 31, no. 2, pp. 135–139, March 2014.

[21] A. Srivastava and V. Karwal, "Windowed r-point update algorithm for discrete fourier transform," in *2013 International Conference on Signal Processing and Communication (ICSC)*, Dec 2013, pp. 185–190.

[22] C. Park, "2d discrete fourier transform on sliding windows," *IEEE Transactions on Image Processing*, vol. 24, no. 3, pp. 901–907, March 2015.

[23] *Synthesis Imaging in Radio Astronomy II*, ser. Astronomical Society of the Pacific Conference Series, vol. 180, Jan. 1999.

[24] A. Thompson, J. Moran, and G. Swenson, Jr, *Interferometry and Synthesis in Radio Astronomy*, 01 1991, vol. -1.

[25] W. Burger and M. J. Burge, *Digital Image Processing: An Algorithmic Introduction Using Java*, 2nd ed. Springer Publishing Company, Incorporated, 2016.

[26] W. M. Gentleman and G. Sande, "Fast fourier transforms: For fun and profit," in *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, ser. AFIPS '66 (Fall). New York, NY, USA: ACM, 1966, pp. 563–578.

[27] J. Stillwell, *Mathematics and its history*. Springer, 2010.

[28] M. R. Spiegel, *Theory and Problems of Complex Variables (SI (Metric) Edition)*. Schaum's Outline Series, 1981.

[29] E. O. Brigham, *The Fast Fourier Transform and Its Applications*. USA: Prentice-Hall, Inc., 1988.

[30] R. P. Norris, A. M. Hopkins, J. Afonso, S. Brown, J. J. Condon, L. Dunne, I. Feain, R. Hollow, M. Jarvis, M. Johnston-Hollitt, and et al., "Emu: Evolutionary map of the universe," *Publications of the Astronomical Society of Australia*, vol. 28, no. 3, p. 215–248, 2011.

[31] M. Frigo and S. G. Johnson, "Fftw: an adaptive software architecture for the fft," in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*, vol. 3, May 1998, pp. 1381–1384 vol.3.

[32] T. Cornwell and B. Humphreys, "Askap science processing, askap-sw-0020," 2016. [Online]. Available: https://www.atnf.csiro.au/projects/askap/ASKAP-SW-0020.pdf

[33] N. Spielmann, "Implementation of Single-Point Discrete Fourier Transform on two dimensional data," B.Sc. thesis, University of Zurich, 2020.