Department of Informatics, University of Zürich

BSc Thesis

Implementation and Performance Evaluation of multi dimensional Fast Fourier Transform Algorithms on Streaming Data using Apache Flink

Claudio Brasser

Matrikelnummer: 14-921-746

Email: `claudio.brasser@uzh.ch`

August 19, 2018

supervised by Prof. Dr. Michael Böhlen

University of Zurich^{UZH}

Department of Informatics

Acknowledgments

Firstly, I'd like to express my sincerest gratitude to my supervisor Muhammad Saad for providing invaluable feedback and support in the process of writing my thesis. I would also like to thank Prof. Dr. Michael Böhlen for the opportunity to write my thesis at the Database technology research group at the University of Zurich.

Abstract

The discrete Fourier Transform (DFT) is an algorithm that can be used to obtain the frequency components of a function dependent of time represented as a set of data samples. It is the foundation of various modern applications. The fast Fourier Transform (FFT) is a faster version of this algorithm, which improves the runtime complexity from $O(n^2)$ to $O(n*log(n))$, thus making the algorithm far more accessible for practical use.
In this thesis, different versions of the FFT algorithm are implemented in order to evaluate their performance in a streaming environment. For this purpose, we built a streaming pipeline for the sampled input data. The dataset origins from the domain of radio astronomy where the Fourier Transform is a commonly used algorithm in the process of computing images from raw data. Cooley-Tukey FFT algorithms are a subset of FFT algorithms that use a divide-and-conquer approach to improve the runtime complexity. The discussed variations of the Cooley-Tukey FFT algorithm are Radix-2, Radix-4, and Split Radix. For each algorithm its explicit definition in the pseudo-code language, the corresponding mathematical equations, as well as a butterfly flow diagram is presented. We conducted experiments in order to evaluate each algorithm's performance both isolated, as well as embedded in the streaming pipeline. We show that Split-Radix has the lowest amount of complex operations from all of the discussed algorithms and thus conclude that it should grant the best performance. In an isolated environment, this hypothesis can be supported by the gathered data over all experiments. The difference in performance between Split-Radix and Radix-4 converged as the input size was increased and evaluated to only approximately 1% at an input size of 1024. The same conclusion can be drawn from the second experiment, where each of the algorithms has been embedded into the pipeline and the complete dataset has been processed. The Split-Radix FFT algorithm performed best across all algorithms and parameter settings followed by Radix-4 and Radix-2.

Zusammenfassung

Die diskrete Fourier Transformation ist ein Algorithmus der benutzt werden kann, um die Frequenz-Komponenten eines Signales zu erhalten, welches als eine Menge von Datenpunkten dargestellt wird und die Basis vieler moderner Applikationen ist. Unter der schnellen Fourier Transformation (FFT) versteht sich einen Algorithmus, welcher die Laufzeit Komplexität des Prozesses von $O(n^2)$ zu $O(n * log(n))$ verbessert, was den Algorithmus für den praktischen Gebrauch zugänglich macht.

In dieser Arbeit werden verschiedene Versionen eines FFT Algorithmus implementiert, um deren Laufzeit auf streaming Daten zu evaluieren. Zu diesem Zweck haben wir eine streaming Applikation für ASKAP Calibrated Visibilities Daten entwickelt. Die Daten stammen aus dem Bereich der Radio-Astronomie, wo die Fourier Transformation oft verwendet wird, um Bilder aus Rohdaten zu berechnen. Cooley-Tukey FFT Algorithmen sind eine Teilgruppe von FFT Algorithmen, welche ein 'divide-and-conquer' Verfahren benutzen und somit eine verbesserte Laufzeitkomplexität erreichen. In dieser Arbeit werden Radix-2, Radix-4 und Split-Radix Implementationen untersucht. Für jeden dieser Algorithmen wird jeweils dessen explizite Definition in Form von Pseudocode, die zugehörige mathematische Herleitung und ein Schmetterlings-Diagramm präsentiert. Wir haben ein Experiment durchgeführt, um die Laufzeit der Algorithmen sowohl in einer isolierten Umgebung als auch eingebunden in die Streaming Umgebung zu evaluieren. Es wird aufgezeigt, dass Split-Radix am wenigsten komplexe Operationen von den drei berücksichtigten Algorithmen hat und schliessen daraus, dass diese Version die beste Leistung liefern sollte. In einer isolierten Umgebung kann diese Hypothese von den gesammelten Daten unterstützt werden, obwohl der Unterschied zwischen Radix-4 und Split-Radix marginal ist. Im zweiten Experiment wurde jeder Algorithmus in die streaming Applikation eingebunden um einen Datensatz zu verarbeiten. Auch in diesem Experiment stellte sich der Split-Radix Algorithmus als die effizienteste unserer Implementationen heraus, unabhängig von anderen Parametern der Applikation.

Contents

List of Figures

List of Tables

1 Introduction

Data streaming platforms have become a central part of various technologies that require real-time processing of continuously generated data. Applications that continuously generate large amounts of data may benefit from a streaming environment since the generated data can be processed in real time or with a small delay. Apache Flink is one such framework that provides functionality for stream processing.

One domain that can benefit from such an environment is Radio Astronomy. The Australian Square Kilometer Array (ASKAP) is a radio telescope made up of 26 antennas that function together as a single instrument. According to the Pawsey supercomputing center, which hosts the generated data, the antennas produce 5.2 terabytes of data per second. This data is processed locally and then streamed to the storage hoster at 956 gigabytes for every 12 hours [2]. In this domain, the Fourier Transform is often used in the process of extracting images from radio astronomy data.

The Fourier Transform describes phase and amplitude for all sinusoid that correspond to a frequency [4]. For discretely sampled signals, a discrete version of the Fourier Transform (DFT) has to be used. The fast Fourier Transform (FFT) labels a computationally faster version of the traditional DFT, that yields the same results. The goal of this thesis is to evaluate the performance of three different FFT algorithms on streaming data. The dataset was generated by the aforementioned ASKAP radio telescope and is stored locally on the same computer that is used for the experiments. In order to embed the algorithms into a streaming environment, we built a pipeline that generates a data stream from the dataset, applies the FFT algorithms and measures their respective performance on inputs of different dimensions. Furthermore we discuss the performance of the other parts of the pipeline that are necessary to convert the stream of records into a format that is applicable to the FFTs.

2 The Fourier Transform

The Fourier Transform has a wide range of applications in digital data and signal processing, including but not limited to electro acoustic music and audio signal processing, image processing, and pattern recognition [12]. The Fourier transform itself has been used for a long time for retrieving the frequency components that contribute to and make up a waveform. The Fourier Transform and its reverse counterpart for a continuous signal are shown in Equation (2.1)

$$\begin{aligned} X(f) &= \int_{-\infty}^{\infty} x(t)e^{-i2\pi ft}dt \\ X(f) &= \int_{-\infty}^{\infty} x(f)e^{i2\pi ft}df \end{aligned} \tag{2.1}$$

with $-\infty < f < \infty$,$-\infty < t < \infty$, and $i = \sqrt{-1}$. Here, the $X(f)$ is the Fourier transformed output, whereas x corresponds to the input signal in the time- or frequency-domain. This formula is of limited use for computer systems because it is defined in a continuous domain for both in- and output. If we want to analyze a sampled signal on a computer system, the *discrete* Fourier Transform has to be used, for which the signal has to be transformed to a discrete domain [3].

2.1 The Discrete Fourier Transform

The discrete Fourier Transform converts a sequence of equally-spaced samples of a function into a same-length sequence of equally-spaced samples of the discrete-time Fourier transform (DTFT), which is a complex-valued function of frequency.
According to James W. Cooley and John W. Tukey [6], the DFT is defined as

$$X(j) = \sum_{k=0}^{N-1} x(k) * W_N^{jk}, j = 0, 1, ..., N-1 \tag{2.2}$$

where $x(k)$ are the complex Fourier coefficients and W_N is the principal Nth root of unity.

$$W_N = e^{-2\pi i/N} = \cos(-2\pi/N) + i\sin(-2\pi/N) \tag{2.3}$$

The Nth root of unity is a complex number c, that yields 1 when raised by N.
For $W_N = e^{-2\pi i/N}$ this can be shown with the *Euler's identity*:

$$\begin{aligned} W_N^N &= e^{-2\pi i/N*N} = e^{-2\pi i} \\ &= \cos(-2\pi) + i*\sin(-2\pi) \\ &= 1 \end{aligned} \tag{2.4}$$

Algorithm 1 depicts the computation of the DFT for a finite, complex input sequence of arbitrary length x. Here the letter i refers to the complex number $i = \sqrt{-1}$.

Algorithm 1 dft(x)

```
$N \leftarrow x.length$
$y \leftarrow Complex[N]$
**for** k=0 to N **do**
    $sum \leftarrow 0$
    **for** j=0 to N **do**
        $w \leftarrow e^{(-2i\pi/N)jk}$
        $sum+ = x[j] * w$
    **end for**
    $y[k] \leftarrow sum$
**end for**
**return** $y$
```

As one can notice, this approach to calculate the discrete Fourier Transform requires $O(n^2)$ complex multiplications and additions as well as trigonometrical computations. The latter can be avoided by pre-computing the sine and cosine values for all necessary values of W. This can be done by a so called twiddle table, to which we will be referring later in the context of the FFT. In order to compute the DFT for a two dimensional input like a matrix, we first need to transform all rows of the input and then all the columns with the aforementioned DFT procedure. This can be emulated in code by transposing the input matrix after transforming the rows and then again transforming the rows of the transposed matrix. With $dft()$ denoting the function for a one dimensional DFT in Algorithm 1 and $transpose()$ being an assumed utility function for transposing a matrix, this is shown in Algorithm 2. Here, x is a two dimensional array (matrix) of complex numbers. We assume that we can access the i-th row of the matrix with $x[i]$ and that we can get the number of rows of a matrix via the $.rows$ property. A two dimensional FFT works exactly the same, the only difference being the algorithm used for computing the one dimensional FFTs in the process. Thus the main aspect of interest in this thesis are the one dimensional FFT implementations because that's where the performance differences origin.

Algorithm 2 dft2d(x)

```
$N \leftarrow x.rows$
**for** i=0 to N **do**
    $dft(x[i])$
**end for**
$transpose(x)$
$N \leftarrow x.rows$
**for** i=0 to N **do**
    $dft(x[i])$
**end for**
$transpose(x)$
**return** $y$
```

2.2 The Fast Fourier Transform

The fast Fourier Transform (FFT) describes a method that allows to efficiently compute the Fourier Transform over discrete data samples [5]. In 1965 J. W. Cooley and John Tukey published their paper "An algorithm for the machine calculation of complex Fourier series" [6] that provided means on how to efficiently calculate the DFT. With their algorithm they were able to reduce the amount of complex additions and multiplications from $O(n^2)$ to $O(n * log(n))$. This can be seen as a turning point in digital signal processing as well as in certain areas of numerical analysis [8].

The Cooley-Tukey FFT algorithm falls under the category of the divide-and-conquer algorithms, which rely on (recursively) splitting a problem of size N into n_1 subproblems of size $N/n_1 = n_2$ until the solving process is trivial. The term *radix* denotes an n_1 or n_2 that is bounded [7], in our case it is constant. The second important concept used for the algorithm is the fact that the multiplicand $W_N = e^{-2\pi i/N}$ has symmetric characteristics, which allows for re-using certain computations. These properties in combination with the divide-and-conquer approach are the main performance drivers of these algorithms.
The decomposition relies on N being a composite number. If N is a power of some real positive integer r, it can be decomposed into $r^{\log_r N}$. In [6], James W. Cooley and John W. Tukey show that by this decomposition, the runtime complexity can be improved from $N * N$ to $N * \log_r N$. In this thesis, we will discuss three different FFT algorithms that use this decomposition. Generally, a Radix-r Cooley-Tukey FFT algorithm needs an input of size $N = r^{\log_r N}$ and recursively divides the input sequence into r sequences of length N/r. Both Radix-2 and Radix-4 algorithms follow this exact pattern. The Split-Radix algorithm uses the same divide-and-conquer principles, but differs in the way it divides the input sequence. A Split-Radix r/s divides the input sequence into multiple different sized sequences of size N/r and N/s. For this Thesis we implemented a Split-Radix 2/4 algorithm, which divides the input into one sequence of length $N/2$ and two sequences of length $N/4$. In further sections, when the term Split-Radix is used, it refers to this 2/4 version of the Split-Radix algorithm.
We implemented the algorithms in a recursive manner, which is closely related to the problem definition and helps translating from the mathematical formulation into a programming

language. The algorithm can also be computed non-recursively by traversing with a "breadth-first" tree traversal, as is the case in most traditional implementations [7]. In Chapter 3 we first present an overview of important concepts in streaming platforms in general, as well as an introduction into the technologies and software packages used in the context of this work. We also discuss our implemented pipeline for the ASKAP calibrated visibilities dataset including a data model, a value deserializer, and transformations which are necessary in order to feed the data to the FFT algorithms.

3 Streaming Pipeline

Data streaming platforms allow data records to continuously flow from the data source to and between applications. A stream processor is embedded in such a platform and performs operations in real-time or close-to-real-time on the stream of records it receives. There are many requirements bound to a stream processor due to possible machine or network failures in distributed systems. It is not only important to restart the process after a failure, but also to recover the state of the computations in order to continue the process without a loss of data or computation cycles. Apache Flink[1] is a stream processing framework that performs operations over bounded or unbounded data streams. When a data stream is generated in real-time it is called *unbounded*. A data stream may also be generated from a fixed-size dataset, which is then called a *bounded* stream. Apache Flink provides features like consistent and efficient checkpoints in order to guarantee recoverability and maintenance. Checkpoints store information about the streaming application's state and allow to recover from the last checkpoint after a failure. In combination with specific storage systems and stream sources Flink can guarantee that every record is only processed and written out once [1].
A basic architecture of a streaming platform consists of a data source, a stream processor, and a data sink. In our case the data is delivered from a fixed-size dataset and thus results in a bounded stream of records. Figure 3.1 depicts an abstract view of our architecture, where Apache Kafka serves both as a source as well as a sink for the stream processor built with Apache Flink. In the following sections we will provide a basic introduction into each of the technologies used for those roles.

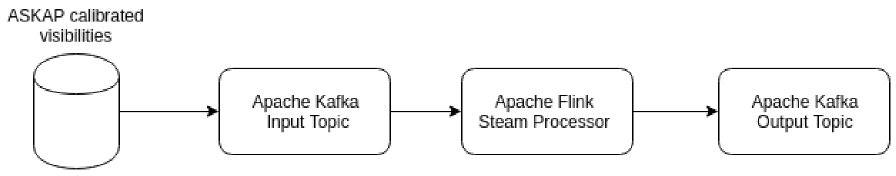

ASKAP calibrated visibilities
Apache Kafka Input Topic
Apache Flink Steam Processor
Apache Kafka Output Topic

Figure 3.1: Architecture overview

3.1 Dataset

The first part of the architecture and the source of the data stream is the dataset. In the context of this work a radio astronomy dataset is used, which is called ASKAP calibrated Visibilities. The data records are complex-valued measurements made by a large array of radio antenna and are stored by 3D continuous coordinates. Each data-point consists out of its coordinates

[1]https://flink.apache.org/

(u, v & w) and a matrix of complex numbers. Each row in the matrix represents a frequency of the input signal and the columns the values of each frequency.
As an example, for a certain set of coordinates u, v, & w a record can be modeled as shown in Figure 3.2.

```
x(u,v,w) = [
       [f1_v1, f1_v2, ..., f1_vn],
       [f2_v1, f2_v2, ..., f2_vn],
       ...
       [fm_v1, fm_v2, ..., fm_vn]
       ]
```

Figure 3.2: Structure of one record

Here *fm_vn* corresponds to the $n-th$ value of the $m-th$ frequency of the record. One data record contains roughly 48 frequencies with 4 values each, although the number of frequencies may vary for some records. The data is stored in the measurementSet (ms) format, a data format specific for radio astronomy that is based on the paradigm of a relational database. It's goal is to allow storage of radioastronomical data including corresponding meta-data while being efficient in both storage-space and data-maintenance by avoiding data redundancy [9]. When mapped onto a discrete raster, the dataset is relatively sparse, as shown in Figure 3.3, where the yellow colored points depict a coordinate that is represented in the dataset.

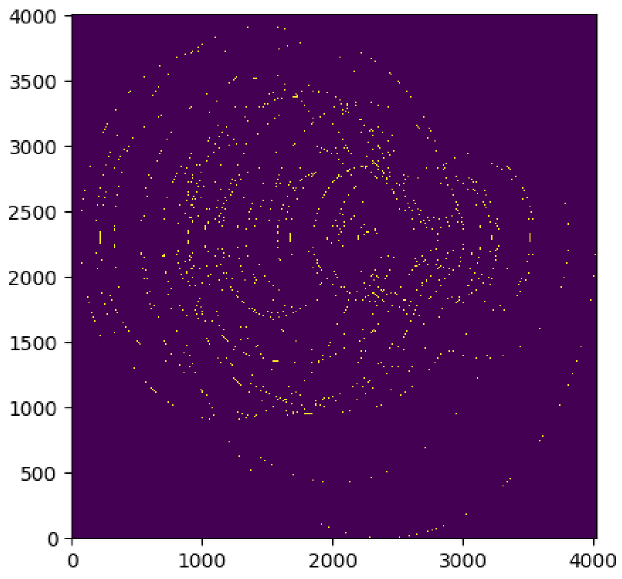


Figure 3.3: Dataset with approximately 33'000 records visualized

3.2 Data Stream Source

As mentioned in the previous section, the data stream used in our application is generated from a file in the measurementSet format, contrary to other stream processing structures which connect to a source which generates data in real-time. Thus there is need for a technology that creates a reliable data stream from the stationary data. Apache Kafka[2] is a streaming platform that provides this functionality. Apache Kafka can be run in a distributed system across multiple servers in order to grant scalability for a streaming application. In this work it is hosted on a local mini-cluster since scalability is not a concern in our context. Kafka stores data as records, which each consists of a key, a value and a timestamp. It allows to publish or subscribe to streams of records and store them in a fault-tolerant way. The storage containers in Kafka are called *topics*, of which multiple can be hosted by the same cluster. In order to publish a data stream one has to open one or more topics to which the data is streamed. In the Kafka API, this is called a *producer*. When subscribing to a topic we can receive the stream of records produced to them, while being able to choose if we want the complete stream from the beginning of the topic or from the moment of subscription, this is called a *consumer*. Figure 3.4 shows a schematic of the Kafka cluster used in our application where two topics are hosted by the same cluster. Multiple producers can publish to the same topic and the same goes for consumers, which is useful when trying to monitor the pipeline traffic through an external connection.

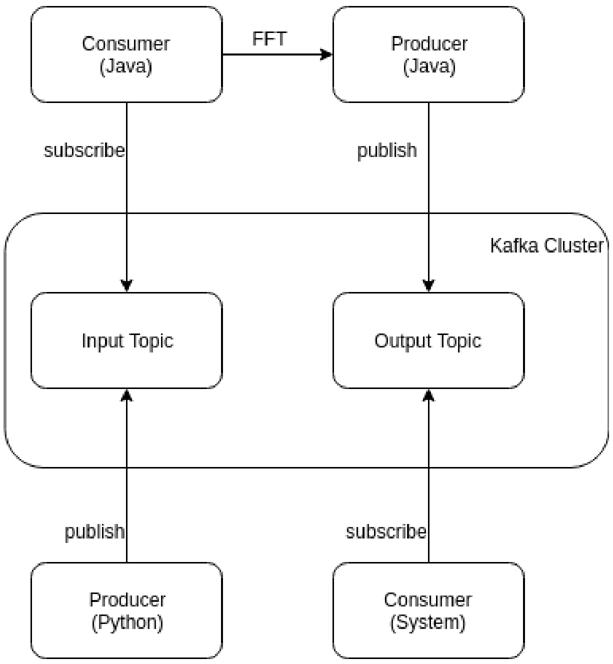


Figure 3.4: Kafka Example Usage

As depicted in Figure 3.1 we need two different topics for our pipeline, an input topic which serves as source for the stream processor and an output topic which serves as a sink.

[2]https://kafka.apache.org/

We created those topics by using a slightly customized shell script provided by the installation of Apache Kafka.

3.2.1 Kafka Producer

In order to publish to the input topic we need to create a *producer*. Kafka provides APIs for several languages that allow us to do so. In this section, we will shortly provide means on why we chose Python as the programming language to achieve this and how it has been done.
As previously mentioned, the dataset is stored in a format specific to radio astronomy, the measurementSet format. Casacore is a software package that has built-in functionality to read this format and query the records stored in a file. This can be done with very few lines of code through *python-casacore*[3], a set of Python bindings for Casacore. The library allows to read the dataset from the local file system and create an object that serves functionality very similar to a relational database. This means that we can query the table represented by the object by using standard SQL statements in order to receive the records in native Python data types. Most of the attributes can be omitted by the query, as we only need the data values and the coordinates of each tuple.

With the relevant data extracted from the measurementSet formatted database, we can use the *kafka-python* library, which is a Python client for Apache Kafka, to create a producer and start publishing the records to the input topic. The producer is instantiated through the provided API, one only needs to provide the networking credentials of the hoster of the topic. Since the cluster is hosted locally in our case, this is done via *localhost*. A function that does this is described in pseudo code in Algorithm 3, where $read\_data()$ corresponds to a function that opens the ms data file, as previously described. Since Python can not serialize complex numbers into the JSON format, we first need to convert the *"DATA"* attribute of each record to a String, which has been omitted in the pseudo-code.

Algorithm 3 producer(x)

```
$data \leftarrow read\_data()$
$producer \leftarrow KafkaProducer('localhost : port', serializer)$
$topic \leftarrow' input'$
**for** record in data **do**
    $producer.send(topic, 'data' : record["DATA"], 'coordinates' : record["UVW"])$
**end for**
```

3.2.2 Apache Flink Stream Processor

After the data is being streamed by the Kafka producer, the next part of the pipeline is the stream processor, built with Apache Flink in the Java programming language. The stream processor subscribes to the input topic through a consumer, which is instantiated through the

[3]http://casacore.github.io/python-casacore/

Java API for Apache Kafka. An incoming record can be used for computations directly as it arrives by using an implementation of *processFunction*, an interface provided by Flink. a *processFunction* is individually applied to each record. However since we need a set of records in order to compute the FFTs, a *processWindowfunction* is used in our case. The latter allows to accumulate incoming records for a certain amount of time and then do computations on the received set of records. The batch of data that arrived during the specified time (*window size*) will be later referred to as *window*. We also created a Kafka producer in the same way as the consumer and assigned the role of the sink to it. With this configuration we can collect all values that should be emitted in a container and emit them as a stream to the output topic via the producer when the window has been processed completely.
Since the data has been serialized in Python we need a custom deserializer to represent the data in Java. This is achieved by parsing the received string in such a way that it can be cast onto native Java data-types afterwards. This process is expensive due to the records having a relatively large *DATA* attribute. We found that using regular expressions in order to search and replace certain elements in the string was superior in terms of running time to iterating over the string as a char array. We created an initial data model including a deserializer to get the data into a native Java format. This data model solely serves the purpose of storing the information in Java and is altered later in a second step in order to fit the format we need for the Fourier Transforms.

This concludes the presentation of our pipeline, as all parts have been introduced. The complete Streaming pipeline modeled in an activity diagram for further clarifications can be seen in Figure 3.5. In Section 3.3 we will introduce the format of the dataset and the records contained in it. We also discuss the transformations made to the records structure in order to apply the FFTs.

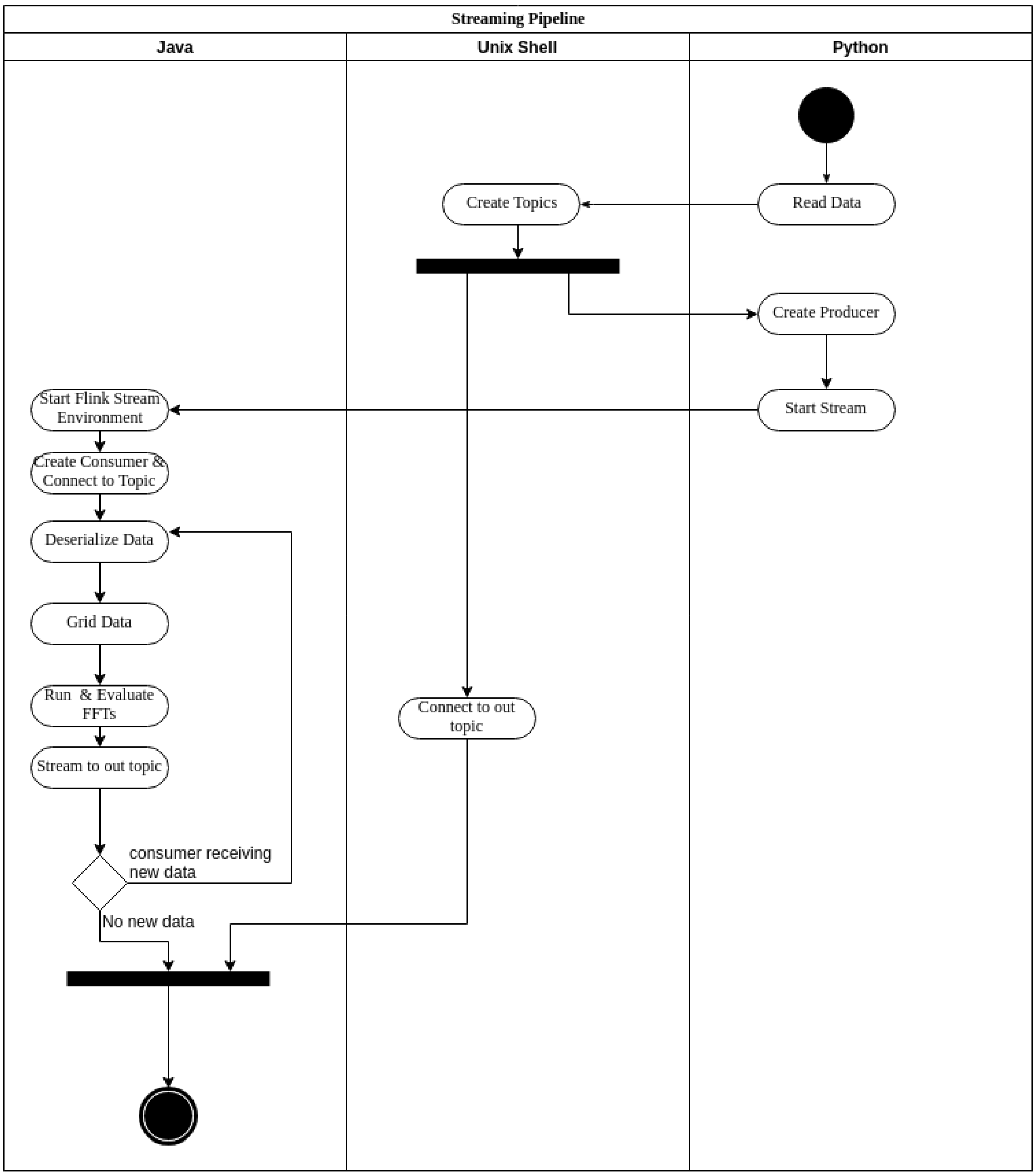


Figure 3.5: Activity Diagram

3.3 Data Model

The initial model for one record consists of a complex matrix as described in Figure 3.2 and its coordinates in the continuous domain. The coordinates in each dimension reach are approximately in the range $[-2000, 2000]$. Recall that the data of a measurement point consists of m rows (frequencies) and n columns (values). In order to test the performance of the FFT algorithms, we applied them to the *first* value of each frequency of each record. This

```
public MeasurementData(double u, double v, double w, Complex[][]
    data) {

        this.u = u;
        this.v = v;
        this.w= w;
        this.data= data;
}
```

Figure 3.6: The data model, depicted in the paradigm of a constructor in the Java programming language.

is sufficient since the goal of this Thesis is solely the performance evaluation of the FFT algorithms. Thus we can reduce the amount of information in our data model by eliminating all other values. Further we will reduce the third dimension of the data (w-coordinate) in order to create a 2 dimensional grid by deleting the w coordinate of all records. This results in a new, simplified version of the data model, as seen in Figure 3.7. Since the FFT algorithms are

```
x(u,v) =    [f1_v1, f2_v1, ..., fm_v1]
```

Figure 3.7: A reduced data model with the w dimension removed and only the first values of all frequencies

applied separately for each frequency, it is useful to have all values of that frequency stored in a separate matrix. Thus we created a second data model, here called *FrequencyMatrix*, which represents the first value of one frequency of each record mapped onto a grid of size $N = 4^r$ for some $r = 1, 2, 3, ...$ indexed by the record's coordinates. This means that every record will contribute to all frequency matrices. We previously mentioned that each record contains 48 frequencies, thus there will be 48 frequency matrices in total with a dimension of Nx N. These matrices are initialized at application startup and are available for insertion in constant time. In the further chapters, the process of calculating the discrete coordinates of a record and inserting the first value of each frequency in the corresponding frequency matrix will be referred to as *gridding*. Figure 3.8 shows an example frequency matrix $f1$ with a grid size of n. Here $f1\_v1(x, y)$ is the first value of the first frequency of the records, whose continuous coordinates are mapped onto a discrete set of coordinates (x/y).

```
f1 =    [
        [f1_v1(0,0), f1_v1(1,0), ..., f1_v1(n,0)],
        [f1_v1(0,1), f1_v1(1,1), ..., f1_v1(n,1)],
        ...
        [f1_v1(0,n), f1_v1(1,n), ..., f1_v1(n,n)]
        ]
```

Figure 3.8: The frequency-matrix data model for example frequency 1. For each coordinate this matrix contains the first value of frequency 1 of records with that coordinate.

With this final data model, the pipeline for an incoming record with the frequency matrices initialized can be graphically described in Figure 3.9. Here extracting the data and coordinates corresponds to deserializing the record from the JSON format. This procedure is invoked automatically on arrival of a record by the Apache Flink framework by registering a Java class with the necessary code. The latter three steps are applied once the *processWindowFunction* is applied.

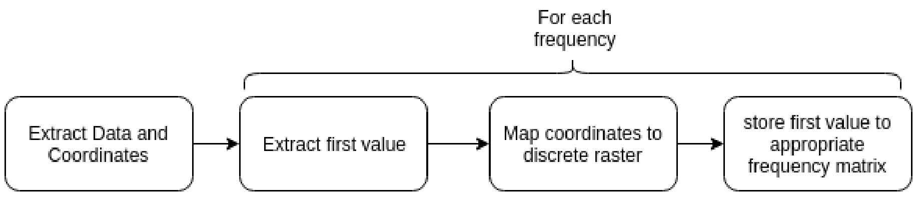

For each frequency
Extract Data and Coordinates
Extract first value
Map coordinates to discrete raster
store first value to appropriate frequency matrix

Figure 3.9: Extracting process

3.4 Complete pipeline example

To conclude this section, we will present an example for the complete process of streaming and transforming one set of records based on simplified data. We assume an example dataset where each record has 2 rows of 4 complex values and that the range of the continuous coordinates in both dimensions is $[0, 10]$. Equation (3.1) shows an example record to be processed.

$$x_{2.24,8.85} = \begin{pmatrix} 1.00 + 2.0i & -1.12 - 7.72i & -1.12 + 7.72i & 9.09 + 0.0i \\ 3.00 + 4.0i & -1.23 - 7.88i & -1.23 + 7.88i & 9.22 + 0.0i \end{pmatrix} \tag{3.1}$$

In our terms, each row is equivalent to one frequency, thus we will refer to the 2 rows as frequency 1 and 2 in this example. In our pipeline, only the first value of each frequency is used for the calculations, thus we can delete all other values of the records. This is shown in table 3.1 where each record is left with one value per frequency.

record	u	v	data
1	u_1	v_1	$f_1(u_1, v_1)$, $f_2(u_1, v_1)$
2	u_2	v_2	$f_1(u_2, v_2)$, $f_2(u_2, v_2)$
3	$\vdots$	$\vdots$	$\vdots$
4	2.24	8.85	$1.00 + 2.00i$, $3.00 + 4.00i$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 3.1: Example dataset

Each record will be serialized to the JSON format and streamed to the *input* topic of the pipeline.

Meanwhile the streaming processor initializes and subscribes to the input topic. The initialization includes preparing 2 *frequency matrices*, since each record has 2 rows. The dimensions of the *frequency matrices* correspond to the grid size we want to use for the FFT computations. In this example, we will use a grid size of 4 by 4. The streaming processor receives all streamed records in the time span that has been specified with the time window size. Each record will be automatically deserialized using a regular expression in order to remove unwanted characters in the string and extract the complex values in the correct order. This lets us instantiate our data model *MeasurementData* for each record, which is just a representation of eq. (3.1) in code. Once the time window concludes, all received records will be handed to the *processWindowFunction*. This function firstly calculates discrete coordinates in the range of our specified grid size trough a linear hash function. A hash function suitable for our example data is presented in fig. 3.10. Here, 4 corresponds to the aforementioned grid dimensions and 10 to the original coordinate range. For our record shown in eq. (3.1) this gives the coordinates $(0/3)$.

```
int hash = (int) (coordinate / 10 * 4);
```

Figure 3.10: Example hash function

With these converted coordinates, we insert the first values of frequency 1 and 2 into the corresponding frequency matrices at the calculated coordinates as shown in fig. 3.11 for frequency 1. This process will be applied to all records received in the time window.

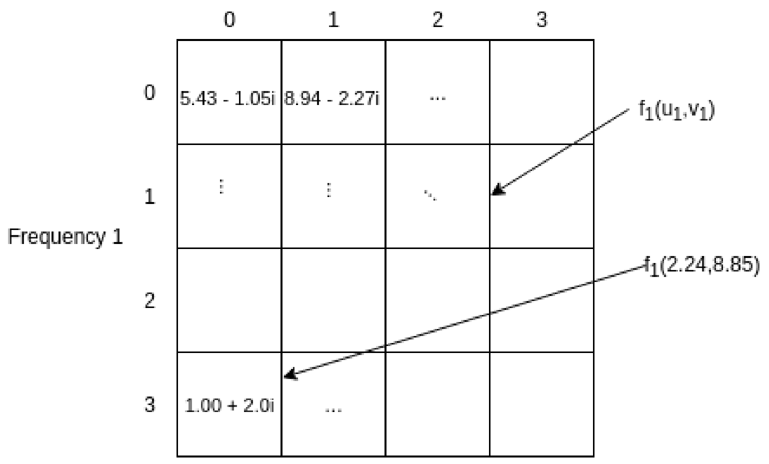


Figure 3.11: Assigning values to the grid of frequency matrix f_1

After this, the Fourier Transformation will be applied to each frequency matrix and the transformed matrices are streamed to the *output* topic, which is the end of our pipeline for this set of records.
Recall that the two dimensional FFT consists of transforming all rows and all columns of the matrix. Since the purpose of this section is solely illustrative, we will only show the calculations for the one dimensional FFT, based on the Radix-2 algorithm explained in the following chapter. If the reader is not familiar with the Cooley-Tukey Radix-2 FFT algorithm, we recommend revisiting this example after reading chapter 4.

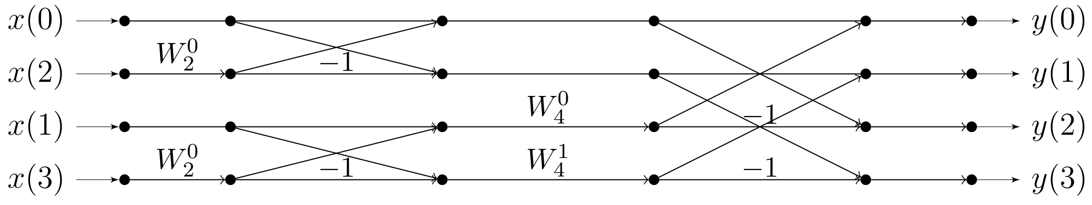


Figure 3.12: 4-Point Radix-2 FFT Butterfly

We will use x as our input array here and assume it to be a row of f_1 in the two dimensional FFT process. The Radix-2 FFT algorithm will be described based on the butterfly flow graph shown in fig. 3.12. In a first step the algorithm recursively divides x into two sub arrays until the base case of $N = 1$ is satisfied. At this point we have four 1-point DFTs:

$$x_0, x_2, x_1, x_3$$

Now the algorithm goes into the combination phase for the bottom-level recursion. The calculations made in this step correspond to the leftmost butterfly calculations in fig. 3.12. The exact calculations can be derived from the butterfly flow graph as described in chapter 4, however for the 2-point FFT for $x(0)$ and $x(2)$ an example is shown in eq. (3.2)

$$\begin{aligned} x_t(0) &= x(0) + x(2) * W \\ x_t(2) &= x(0) - x(2) * W \end{aligned} \tag{3.2}$$

This gives us 2 2-point DFTs $[x_t(0), x_t(2)]$ and $[x_t(1), x_t(3)]$ which will be returned to the top-level recursion and combined to a 4-point FFT $[y_0, y_1, y_2, y_3]$, as described in the second part of fig. 3.12 and in eq. (3.3).

$$\begin{aligned} y(0) &= x_t(0) + x_t(1) * W_4^0 \\ y(1) &= x_t(2) + x_t(3) * W_4^1 \\ y(2) &= x_t(0) - x_t(2) * W_4^0 \\ y(3) &= x_t(2) - x_t(3) * W_4^1 \end{aligned} \tag{3.3}$$

As previously mentioned, this is the last step in the pipeline. When the two dimensional FFT procedure is finished, the transformed frequency matrices are streamed to the output topic and once all matrices are transformed the stream processor waits until the next time window concludes.

4 Fast Fourier Transforms

In this chapter we will first cover the necessary prerequisites for the different representations of the FFT algorithms. Secondly we will illustrate how each algorithm can be derived from the definition of the DFT, as well as how this is translated into a program.

Butterfly Diagrams

In order to describe FFT algorithms, often so called *Butterfly Diagrams* are being used to show which point in the input contributes to which point in the output. In Figure 4.1 one can see an example butterfly for the 2-Point Radix-2 case and in Equation (4.1) its corresponding equation. Here, $x(0)$ and $x(1)$ correspond to the input points whereas $y(0)$ and $y(1)$ are the output points. An edge that is labeled with a term like W is multiplied with the value. An incoming edge that arises from a different input, indicates that the values are added together. With this notation we can denote the outputs in Figure 4.1 as:

$$\begin{aligned} y(0) &= x(0) + x(1) * W \\ y(1) &= x(0) - x(1) * W \end{aligned} \tag{4.1}$$

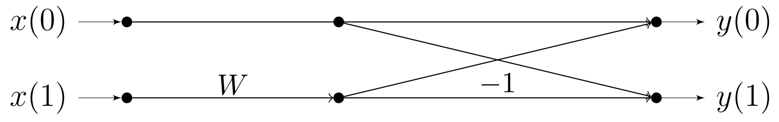


Figure 4.1: 2-Point Radix-2 FFT Butterfly

Twiddle Factors

The earlier introduced factor $W_N^j = e^{-2\pi ij/N}$ is often called *Twiddle Factor* in the context of FFTs and is a central part of the Fourier Transform. For example, in a Radix-2 FFT the twiddle factor has to be computed $\frac{N}{2} * log_2(N)$ times, which contributes a notable amount to the total runtime due to the trigonometrical computations associated with it. Because w is only dependent on the input length N and the iterator j, it can be computed without knowing the input data. This allows us to pre-compute W for all necessary values of N and j. Since every possible value of N is a power of 2, we have to store $log_2 N * N$ values. Because of

$$W_N^j = -W_N^i$$

with $j = 1, 2, ..., N/2 - 1$ and $i = j + N/2$ we could reduce this number to $log_2 N * \frac{N}{2}$ by only storing values from $N = 0, 1, ..., N/2 - 1$. This would however require some additional statements for each lookup to guarantee that we are using the correct value. We preferred the ability to make the lookup as simple and quick as possible over the small improvement in memory space efficiency. Thus we computed the full-length table on initialization and stored it as $twiddleTable$. In the algorithms presented later, it is assumed that this has been done in advance. For any input size N and iterator j, the value for W_N^j can be accessed from the lookup table by $twiddleTable[\log_2 N][j]$.

Pseudo Code Notation

In this section we will briefly go over the notation used in the presented algorithms. Assignments are denoted using the $\leftarrow$ sign. Using the keyword $Complex$ on the right side of an assignment gives a complex number or an array of complex numbers in the case of $Complex[]$. Loop boundaries are always including the lower boundary and excluding the upper boundary. Since all of the algorithms are based on complex values, all used arithmetic operations $(*, +, -)$ refer to their complex counterparts. For two complex numbers A and B, the complex addition $C = A + B$ is defined as

$$C.real = A.real + B.real$$
$$C.imag = A.imag + B.imag$$

and complex multiplication $C = A * B$ as

$$C.real = A.real * B.real - A.imag * B.imag$$
$$C.imag = A.real * B.imag + A.imag * B.real$$

Decimation in time vs Decimation in frequency

FFT algorithms can be performed either by *Decimation In Time* (DIT) or *Decimation In Frequency* (DIF). Both options lead to the same numerical result, although their respective butterflies look differently. All algorithms discussed in this work are DIT implementations. If an algorithm is considered DIT or DIF is determined by whether n_1 or n_2, with $N = n_1 * n_2$, is chosen as the radix [7]. FFT algorithms that perform the operations inplace either require the input array in bit-reversed order in the case of DIT or lead to the output being in bit-reversed[1] order in the DIF case. This is not required in the recursive algorithms presented here since the recursive splitting (e.g. by even or odd indexes in Radix-2) ensures the correct order of the input.

[1]The bit-reversed indexing order places the even-indexed elements of an array in the first half of the array and vice versa for the odd-indexed elements.

4.1 Radix-2

4.1.1 Theory

The Radix-2 Cooley-Tukey FFT algorithm divides an input sequence of size N into two sequences of size $N/2$. This resolves in a symmetric binary recursion tree, as shown in an example in Figure 4.2.

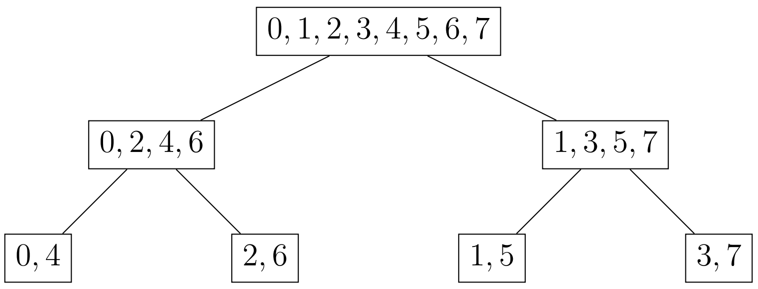


Figure 4.2: 8-Point Radix-2 FFT Array Division

The input values get assigned to one of the subsequences if their index in the original one is even and to the other one vice versa. In the following section we will cover how this decomposition is done mathematically, and how it helps to improve the runtime complexity of the conventional DFT algorithm. The mathematical derivation for all three algorithms is based on the calculations shown by William T. Cochran et al. [5].

Firstly, it is important to recall the definition of the DFT as

$$X(j) = \sum_{k=0}^{N-1} x(k) * W_N^{jk}, j = 0, 1, ..., N-1 \tag{4.2}$$

We can split this into two sums over $k = 0, 1, ..., N/2 - 1$ for the even- and odd-indexed input values $x(2k)$ and $x(2k+1)$:

$$X(j) = \sum_{k=0}^{N-1} x(k) * W_N^{jk} = \sum_{k=0}^{N/2-1} x(2k) * W_N^{j2k} + \sum_{k=0}^{N/2-1} x(2k+1) * W_N^{j(2k+1)} \tag{4.3}$$

Since W_N^j is a constant, we can factor it out in the second summation. Note that the 2 in the exponent of W_N is taken into brackets, which is purely cosmetic and helps in making the further steps clearer.

$$X(j) = \sum_{k=0}^{N/2-1} x(2k) * (W_N^2)^{jk} + W_N^j \sum_{k=0}^{N/2-1} x(2k+1) * (W_N^2)^{jk} \tag{4.4}$$

Now when we recall the definition of $W_N = e^{-2\pi i/N}$, we can take in the exponent 2 of W_N^2 into the exponential function:

$$W_N^2 = e^{-2\pi i/N*2} = e^{-2\pi i/(N/2)} = W_{N/2} \tag{4.5}$$

Thus we can substitute W_N^2 with $W_{N/2}$ in Equation (4.4):

$$X(j) = \sum_{k=0}^{N/2-1} x(2k) * (W_{N/2})^{jk} + W_N^j \sum_{k=0}^{N/2-1} x(2k+1) * (W_{N/2})^{jk} \tag{4.6}$$

This corresponds to the original definition of the DFT in Equation (4.2) for both the even- and the odd-indexed parts. We can express this relation in a more abstract manner as

$$X(j) = E(j) + W_N^j * O(j) \tag{4.7}$$

where $E(j)$ & $O(j)$ denote the DFT of the even/odd-indexed parts of the input sequence. The real and imaginary values for W_N^j are displayed in Figure 4.3. Here it can be seen that both parts are sinusoids with the same amplitude and a phase shift of $N/4$. In the Radix-2 case, we are using the fact that $W_N^j = -W_N^i$ with $j = 0, 2, ..., N/2 - 1$ and $i = j + N/2$.

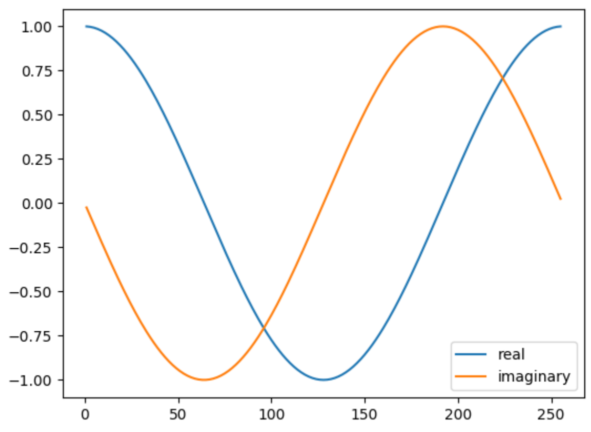


Figure 4.3: Symmetry of exponential function for N =256

With this information we can split Equation (4.7) into two parts

$$X(j_1) = E(j_1) + W_N^{j_1} O(j_1) \tag{4.8}$$

$$X(j_1 + \frac{N}{2}) = E(J_1) - W_N^{j_1} O(j_1) \tag{4.9}$$

for $j_1 = 0, 1, ..., \frac{N}{2} - 1$.

4.1.2 Radix-2 algorithm

In this section we will provide a recursive algorithm that calculates the DFT of an one dimensional complex input using the Radix-2 decimation in time Cooley-Tukey FFT. The algorithm depends on the input length being a power of 2, however this assertion has been omitted in the presented algorithm for simplicity. Since the input length is being cut in half with every recursive call, the base case of the recursion is reached when the input length is 1. In this case we have to perform a 1-point DFT on $x[0]$, which conveniently is just the element itself and thus just return a new array of complex numbers of size 1 with $x[0]$ in it. In every other case, the recursion splits the input array x into two separate arrays, of which one contains the even indexed values of x and the other contains the odd indexed values. The *conquer* part of the algorithm combines two smaller DFTs into a larger DFT. This part uses Equations (4.8) and (4.9) and the periodicity of W_N^j to calculate the DFT for the combination of the returned arrays from two recursive calls. Thanks to the periodicity of W_N^j we can assign $y[k]$ and $y[k + n/2]$ at the same time and thus only have to do $N/2$ iterations for a DFT of size N. In order to save one multiplication per iteration we store the second summand from Equation (4.8) as `oddVal` so it only has to be calculated once. This approach is also described by R. Neuenfeld et al. [11]. W_N^k can be accessed in constant time from the lookup table as described earlier. Without keeping $oddVal$ in memory, this algorithm would lead to $N/2 \log N$ times 2 complex multiplies and 2 complex additions. With the proposed optimization, the amount of multiplies can be reduced to $\frac{N}{2} \log N$ with the same amount of additions as the original approach.

4.1.3 Radix-2 butterfly diagram

Figure 4.4 shows the flow graph for a complete 8-Point Radix-2 FFT. Going from left to right, the first set of butterflies represent the combination of 8 1-point DFTs to 4 2-point DFTs. It is noticeable here, that the input elements are not in the same order as the output elements. The specific ordering is derived from recursively splitting the input into even- and odd-indexed sub-arrays as shown in Figure 4.2 and helps keeping the flow-graph clear.

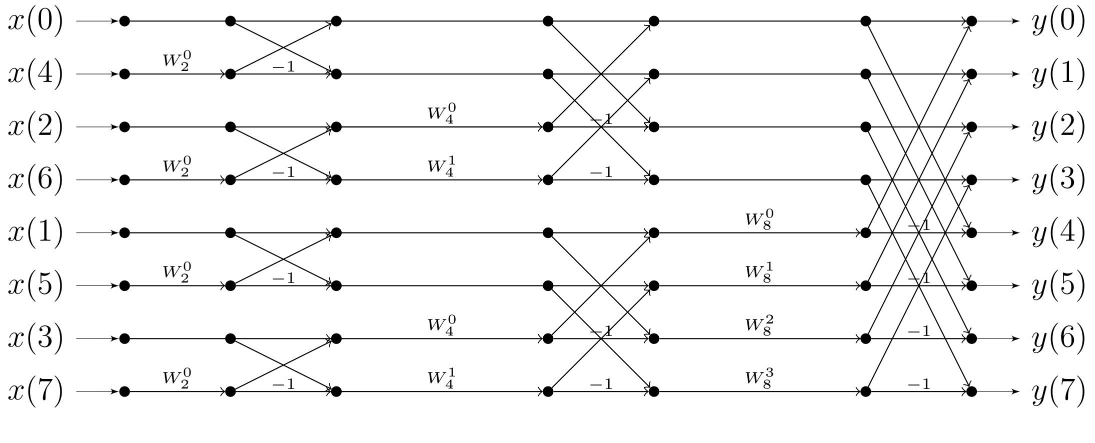


Figure 4.4: 8-Point Radix-2 FFT Butterfly

Algorithm 4 fftRadix2(x)

```
Require: $x.length\ is\ power\ of\ 2$
  $n \leftarrow x.length$
  if $n == 1$ then
    return $new\ Complex[]\{x[0]\}$
  end if
  $even \leftarrow new\ Complex[n/2]$
  $odd \leftarrow new\ Complex[n/2]$
  for K=0 to n/2 do
    $even[k] = x[2 * k]$
    $odd[k] = x[2 * k + 1]$
  end for
  $a \leftarrow fft2(even)$
  $b \leftarrow fft2(odd)$
  $y \leftarrow new\ Complex[n]$
  for k=0 to n/2 do
    $wk \leftarrow twiddleTable[log_2(n)][k]$
    $oddVal \leftarrow wk * b[k]$
    $y[k] \leftarrow a[k] + oddVal$
    $y[k + n/2] \leftarrow a[k] - oddVal$
  end for
  return $y$
```

4.2 Radix-4

4.2.1 Theory

The Radix-4 case works very similarly to the Radix-2 algorithm described earlier. The algorithm splits the input into 4 smaller arrays of size $n/4$, calculates the DFT of those and combines the results to get the DFT of the original input. Thus the input size needs to be a power of 4. The input is split based on the result of $index \ \% \ 4$, with $\%$ being the *modulus* operator, as shown in Figure 4.5.

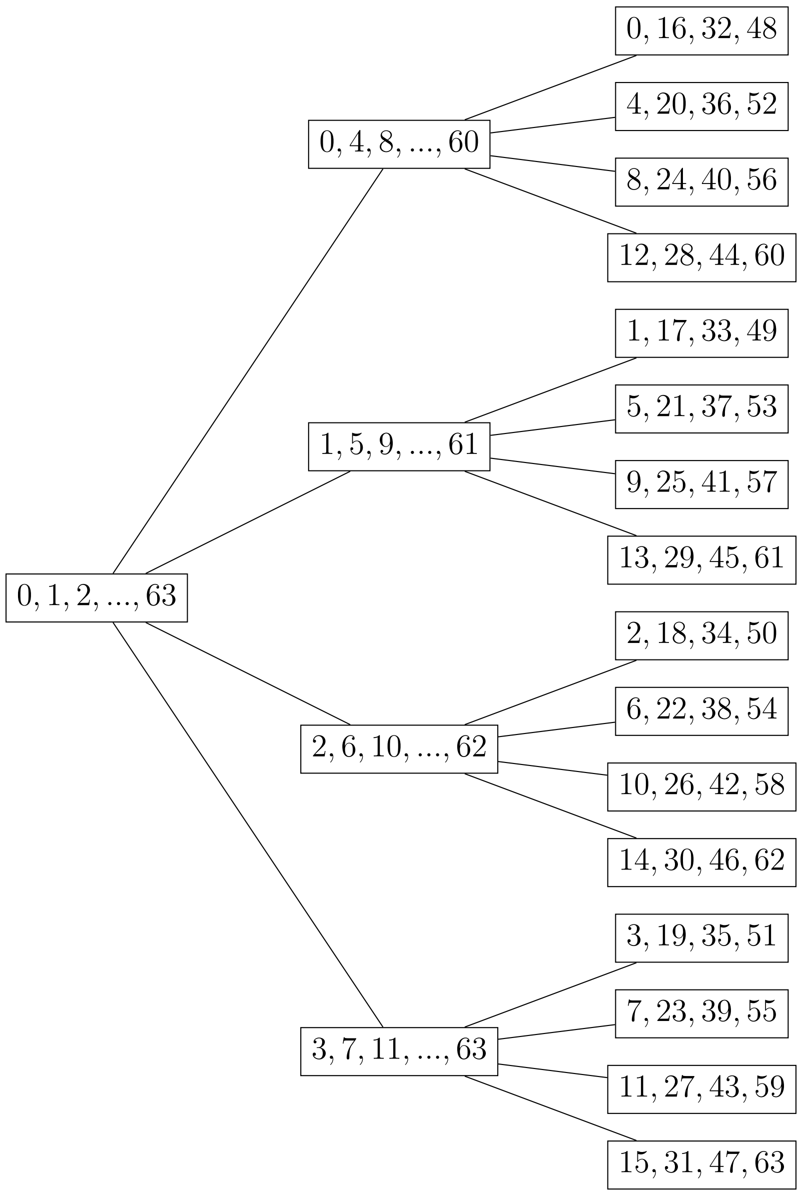


Figure 4.5: 64-Point Radix-4 FFT Array Division

We can derive the Radix-4 FFT in a very similar fashion as the Radix-2 version. With the original definition of the DFT being

$$X(j) = \sum_{k=0}^{N-1} x(k) * W_N^{jk} \tag{4.10}$$

we can split the sum into four parts that cover all indices of the input sequence

$$\begin{aligned} X(j) = \sum_{k=0}^{N/4-1} x(4k) * W_N^{j4k} + \sum_{k=0}^{N/4-1} x(4k+1) * W_N^{j(4k+1)} + \\ \sum_{k=0}^{N/4-1} x(4k+2) * W_N^{j(4k+2)} + \sum_{k=0}^{N/4-1} x(4k+3) * W_N^{j(4k+3)} \end{aligned} \tag{4.11}$$

and factor out the twiddle factors in the latter three sums.

$$\begin{aligned} X(j) = \sum_{k=0}^{N/4-1} x(4k) * (W_N^4)^{jk} + W_N^j \sum_{k=0}^{N/4-1} x(4k+1) * (W_N^4)^{jk} + \\ W_N^{2j} \sum_{k=0}^{N/4-1} x(4k+2) * (W_N^4)^{jk} + W_N^{3j} \sum_{k=0}^{N/4-1} x(4k+3) * (W_N^4)^{jk} \end{aligned} \tag{4.12}$$

With Equation (4.5) we can reformulate W_N^4 to $W_{N/4}$ at which point each summand is a complete DFT of the part of the input sequence its indices cover.

$$\begin{aligned} X(j) = \sum_{k=0}^{N/4-1} x(4k) * (W_{N/4})^{jk} + W_N^j \sum_{k=0}^{N/4-1} x(4k+1) * (W_{N/4})^{jk} + \\ W_N^{2j} \sum_{k=0}^{N/4-1} x(4k+2) * (W_{N/4})^{jk} + W_N^{3j} \sum_{k=0}^{N/4-1} x(4k+3) * (W_{N/4})^{jk} \end{aligned} \tag{4.13}$$

This can be stated in a more abstract way as a sum of four DFTs:

$$X(j) = FIRST(j) + W_N^j * SECOND(j) + W_N^{2j} * THIRD(j) + W_N^{3j} * FOURTH(j) \tag{4.14}$$

Again thanks to the periodicity of W_N we can assign four values of the output for the same j. In Figure 4.6 one can see the values of W_{256} divided into four equal parts, say $n1$, $n2$, $n3$, and $n4$. We can express the values for the latter three based on the values of $n1$ as shown in Equation (4.15)

$$\begin{aligned} n2.real &= n1.imag \\ n2.imag &= -n1.real \\ n3.real &= -n1.real \\ n3.imag &= -n1.imag \\ n4.real &= -n1.imag \\ n4.imag &= n1.real \end{aligned} \tag{4.15}$$

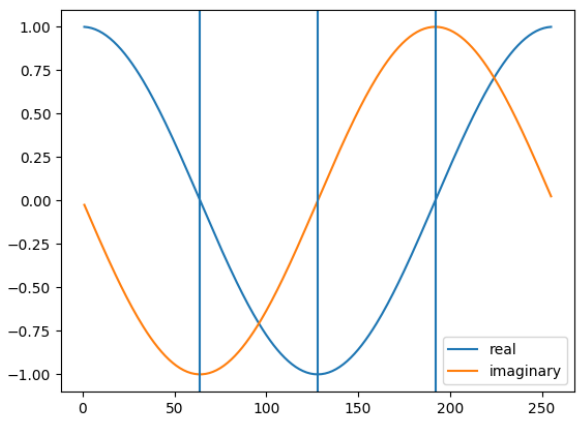


Figure 4.6: Symmetry of exponential function for N =256

This is used in the combination part of Algorithm 5, where we pre-compute the values of

$$X(j) = FIRST(j) + W_N^j * SECOND(j) + W_N^{2j} * THIRD(j) + W_N^{3j} * FOURTH(j)$$

for $j = 0, 1, ..., N/4 - 1$. Thanks to the aforementioned behavior of W_N, we can assign the values for $j + k * (N/4), k = 1, 2, 3$ in the same operation, simply by modifying the calculated values so that the periodicity of W_N is respected correctly. We can apply some optimizations similar to what we did in Radix-2 by temporarily storing repeated computations. With this optimization we need to perform $N/4 \log_4(N)$ times 8 complex additions and 3 complex multiplies. With some basic algebra we can compare this to the amount of complex additions of Radix-2:

$$\frac{N}{4} \log 4(N) * 8 = \frac{N}{4} \frac{\log 2(N)}{2} * 8 = N \log 2(N)$$

where we see that the amount of additions is the same for both cases. As for the multiplies, we can do the same:

$$\frac{N}{4} \log 4(N) * 3 = \frac{N}{4} \frac{\log 2(N)}{2} * 3 = \frac{3}{8} N \log 2(N) < \frac{N}{2} N \log 2(N)$$

Thus Radix-4 saves $\frac{1}{8} N \log 2(N)$ complex multiplications as compared to Radix-2.

4.2.2 Radix-4 algorithm

The recursive Radix-4 decimation in time algorithm works in a very similar way as the Radix-2 version. At first, we cover the base case of $N = 1$ and return the 1-point DFT in an array of size 1. If the the base case is not satisfied, N has to be of some power of 4 since this is a precondition of the algorithm and we only divide the length by exactly 4. Because of this assumption we can split x into 4 equally long parts, as also shown in Figure 4.5 and recursively calculate the DFT of all subsequences. The logic of the combination phase strictly corresponds to the butterfly shown in Figure 4.7. In a software implementation of the algorithm, some optimizations can be made here in order to minimize the amount of complex additions and multiplications. We can temporarily store $a + c$ and $a - c$ and use the result in the assignments to y since both of those terms are evaluated twice otherwise. The variables a, b, c, d are the result of the same optimization done in Radix-2 with the *oddVal* variable. In the butterfly in Figure 4.7 one can clearly see that all of these values are necessary for all 4 assigned values, making these assignments beneficial in terms of operation count. Also since a multiplication of a complex number with i is the same as switching real and imaginary parts of the number and a multiplication of the new real part with -1, we can directly express these operations of the butterfly by a new complex number and avoid a complex multiplication. It is worth to note that since the multiplication with i is rather trivial it might arguably be faster than a constructor call for a complex number, however we found that there was no significant difference in the measured time for both variants in our environment. Steven G. Johnson and Matteo Frigo [10] also argue, that these operations can be regarded as free, since they can be avoided by doing additions instead of subtractions or vice versa.

Algorithm 5 fftRadix4(x)

```
Require: $x.length\ is\ power\ of\ 4$
  $n \leftarrow x.length$
  $log2n \leftarrow log_2(n)$
  $n4 \leftarrow n/4$
  if $n == 1$ then
    return $new\ Complex[]\{x[0]\}$
  end if
  $first \leftarrow new\ Complex[n/4]$
  $second \leftarrow new\ Complex[n/4]$
  $third \leftarrow new\ Complex[n/4]$
  $fourth \leftarrow new\ Complex[n/4]$
  for K=0 to n4 do
    $first[k] \leftarrow x[4 * k]$
    $second[k] \leftarrow x[4 * k + 1]$
    $third[k] \leftarrow x[4 * k + 2]$
    $fourth[k] \leftarrow x[4 * k + 3]$
  end for
  $q \leftarrow fft2(first)$
  $r \leftarrow fft2(second)$
  $s \leftarrow fft2(third)$
  $t \leftarrow fft2(fourth)$
  $y \leftarrow new\ Complex[n]$
  for k=0 to n4 do
    $wk \leftarrow twiddleTable[log2n][k]$
    $wk2 \leftarrow twiddleTable[log2n][2 * k]$
    $wk3 \leftarrow twiddleTable[log2n][3 * k]$
    $a \leftarrow q[k]$
    $b \leftarrow r[k] * wk$
    $c \leftarrow s[k] * wk2$
    $d \leftarrow t[k] * wk3$
    $y[k] \leftarrow a + b + c + d$
    $y[k + n4] \leftarrow a + Complex(b.imag, -b.real) - c + Complex(-d.imag, d.real)$
    $y[k + 2 * n4] \leftarrow a - b + c - d$
    $y[k + 3 * n4] \leftarrow a + Complex(-b.imag, b.real) - c + Complex(d.imag, -d.real)$
  end for
  return $y$
```

4.2.3 Radix-4 butterfly diagram

The butterfly in Figure 4.7 for a Radix-4 DIT FFT requires three different values for W_N^k where k corresponds to j in Equation (4.14). The summands are multiplicated by $i, -i, 1, -1$ based on which output a specific edge contributes to in order to satisfy the properties stated in Equation (4.15).

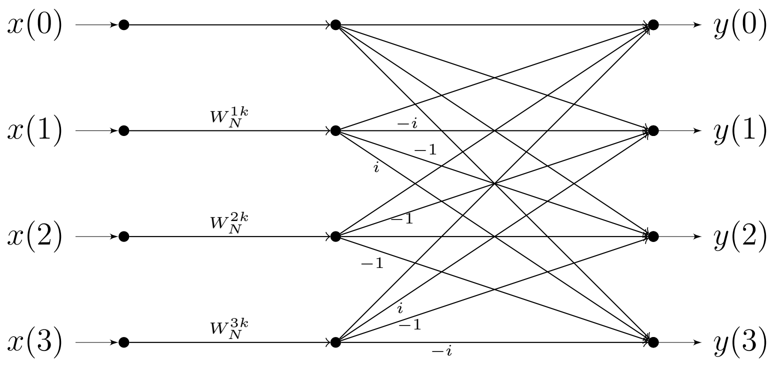


Figure 4.7: 4-Point Radix-4 DIT FFT Butterfly

Equation (4.16) expresses the butterfly mathematically. In this form it is clearly visible where there is room for optimization since factors like $x(1) * W_N^k$ are calculated three times over the course of all assignments.

$$\begin{aligned} y(0) &= x(0) + x(1) + x(2) + x(3) \\ y(1) &= x(0) - i * x(1) * W_N^k - x(2) * W_N^{2k} + i * x(3) * W_N^{3k} \\ y(2) &= x(0) - x(1) * W_N^k + x(2) * W_N^{2k} - x(3) * W_N^{3k} \\ y(3) &= x(0) + i * x(1) * W_N^k - x(2) * W_N^{2k} - i * x(3) * W_N^{3k} \end{aligned} \tag{4.16}$$

4.3 Split Radix

4.3.1 Theory

The Split-Radix FFT algorithm combines two different Radix-r algorithms resulting in an unbalanced recursion tree. We implemented a 4/2 Split Radix Algorithm, which means that in each recursive step, the input is split into one subsequence of size $N/2$ and two subsequences of size $N/4$. The $N/2$ length subsequence contains the elements with index $2 * k$ for $k = 0, 1, 2, ..., N/2 - 1$ while the other two contain elements $4 * k + 1$ and $4 * k + 3$ respectively for $k = 0, 1, 2, ..., N/4 - 1$. An example array division is shown in Figure 4.8.

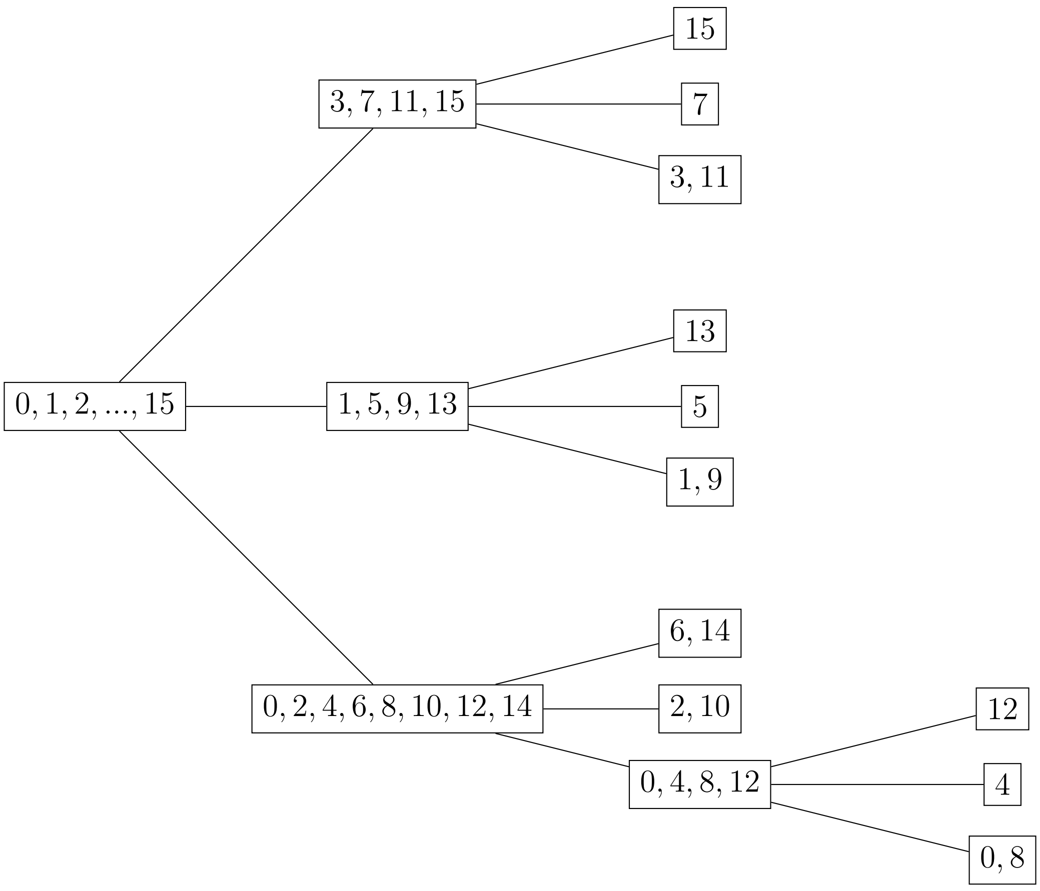


Figure 4.8: 16-Point Split-Radix FFT Array Division

The mathematical equation for the Split-Radix algorithm can be derived in similar fashion to the other two presented algorithms, however since we divide the sequence in an unbalanced fashion the sum boundaries and twiddle factors have to be evaluated carefully. In addition to [5], we used the mathematical formulations shown by Steven G. Johnson and Matteo Frigo [10] as a foundation of the following equations. In their work, the authors use a modified version of the division process for the elements with indices $4 * k + 3$, which is not the case in our work.

$$X(j) = \sum_{k=0}^{N-1} x(k) * W_N^{jk} \tag{4.17}$$

This time we split the original definition of the DFT into three parts. The even indexed part corresponds to the even indexed part of the Radix-2 algorithm, whereas the odd indexed elements are split much like in the Radix-4 algorithm.

$$\begin{aligned} X(j) = \sum_{k=0}^{N/2-1} x(2k) * W_N^{j2k} + \sum_{k=0}^{N/4-1} x(4k+1) * W_N^{j(4k+1)} + \\ \sum_{k=0}^{N/4-1} x(4k+3) * W_N^{j(4k+3)} \end{aligned} \tag{4.18}$$

Since the sum over the even indices does not have a $+$ operator in the exponent of the twiddle factor we don't have to factor anything out. For the other two sums, we do so in the same way as described in Section 4.2.

$$\begin{aligned} X(j) = \sum_{k=0}^{N/2-1} x(2k) * (W_N^2)^{jk} + W_N^j \sum_{k=0}^{N/4-1} x(4k+1) * (W_N^4)^{jk} + \\ W_N^{3j} \sum_{k=0}^{N/4-1} x(4k+3) * (W_N^4)^{jk} \end{aligned} \tag{4.19}$$

Again, with Equation (4.5) we can reformulate W_N^4 to $W_{N/4}$, as well as W_N^2 to $W_{N/2}$ at which point each summand is a complete DFT of the part of the input sequence its indices cover.

$$\begin{aligned} X(j) = \sum_{k=0}^{N/2-1} x(2k) * (W_{N/2})^{jk} + W_N^j \sum_{k=0}^{N/4-1} x(4k+1) * (W_{N/4})^{jk} + \\ W_N^{3j} \sum_{k=0}^{N/4-1} x(4k+3) * (W_{N/4})^{jk} \end{aligned} \tag{4.20}$$

This can be stated in a more abstract way as a sum of three DFTs:

$$X(j) = EVEN(j) + W_N^j * SECOND(j) + W_N^{3j} * THIRD(j) \tag{4.21}$$

Which can be split based on j:

$$\begin{aligned} X(j_1) &= EVEN(j_1) + W_N^j * SECOND(j_1) + W_N^{3j} * THIRD(j_1) \\ X(j_1 + \frac{N}{2}) &= EVEN(j_1) + W_N^j * SECOND(j_1) + W_N^{3j} * THIRD(j_1) \\ X(j_1 + \frac{N}{4}) &= EVEN(j_1 + \frac{N}{4}) + W_N^j * SECOND(j_1) + W_N^{3j} * THIRD(j_1) \\ X(j_1 + 3 * \frac{N}{4}) &= EVEN(j_1 + \frac{N}{4}) + W_N^j * SECOND(j_1) + W_N^{3j} * THIRD(j_1) \end{aligned} \tag{4.22}$$

for $j_1 = 0, 1, 2, ..., N/4 - 1$

Note that the indices $j_1 + \frac{N}{4}$ and $j_1 + 3 * \frac{N}{4}$ are calculated with $EVEN(j_1 + \frac{N}{4}$ and not $EVEN(j_1)$. This can be derived if we look at the sum for the even part in Equation (4.20). We combine the three DFTs into a DFT of size N, however EVEN is only of size N/2, meaning that we calculate it with $W_{N/2}$ so that $N/2$ is the period of W. With that knowledge we can use $EVEN(j_1)$ for $X(j_1)$ and $X(j_1 + N/2)$ because $W_{N/2}^{j_1} = W_{N/2}^{j_1+N/2}$ and the same goes for $X(j_1 + N/4)$ and $X(j_1 + 3 * N/4)$.

4.3.2 Split-Radix algorithm

Algorithm 6 depicts a split-radix 4/2 FFT where the input length is required to be a power of 2. Since the recursion tree for this algorithm is unbalanced due to multiple recursion calls with different split sizes, we have to check the extra base case of $n = 2$ here. In this case we perform a Radix-2 butterfly in order to calculate the 2-point DFT. Splitting the input is depicted in two separate iterations here for a better visual representation, this can be optimized to one iteration with $k = 0, 1, ..., n/2 - 1$ with a simple conditional statement for the $n/4$ splits. The assignments to y in the combination part represent equation Equation (4.22). The other variable assignments are mostly optimizations to prevent redundant operations of the same value.

Algorithm 6 Compute split radix DIT FFT on x

```
Require: $x.length\ is\ power\ of\ 4$
  $n \leftarrow x.length$
  $log2n \leftarrow log_2(n)$
  $n4 \leftarrow n/4$
  if $n == 1$ then
    return $new\ Complex[]\{x[0]\}$
  else if $n == 2$ then
    $y0 \leftarrow x[0] + x[1]$
    $y1 \leftarrow x[0] - x[1]$
    return $new\ Complex[]\ \{y0, y1\}$
  end if
  $first \leftarrow new\ Complex[n/2]$
  $second \leftarrow new\ Complex[n/4]$
  $third \leftarrow new\ Complex[n/4]$
  for k=0 to n/2 do
    $first[k] \leftarrow x[2 * k]$
  end for
  for k=0 to n/4 do
    $second[k] \leftarrow x[4 * k + 1]$
    $third[k] \leftarrow x[4 * k + 3]$
  end for
  $q \leftarrow fftSplit(first)$
  $r \leftarrow fftSplit(second)$
  $s \leftarrow fftSplit(third)$
  $y \leftarrow new\ Complex[n]$
  for k=0 to n4 do
    $wk \leftarrow twiddleTable[log2n][k]$
    $wk2 \leftarrow twiddleTable[log2n][3 * k]$
    $a1 \leftarrow q[k]$
    $a2 \leftarrow q[k + N4]$
    $b \leftarrow r[k] * wk$
    $c \leftarrow s[k] * wk2$
    $temp \leftarrow i * (a - b)$
    $y[k] \leftarrow a1 + b + c$
    $y[k + N2] \leftarrow a1 - b - c$
    $y[k + N4] \leftarrow a2 - temp$
    $y[k + 3 * N4] \leftarrow a2 + temp$
  end for
  return $y$
```

4.3.3 Split-Radix butterfly diagram

The combination of Radix-2 and Radix-4 in Split-Radix leads to the "reverse L"-shaped butterfly displayed in Figure 4.9. The equivalent mathematical expression is stated in Equation (4.23) where the values that are temporarily stored in Algorithm 6 are exposed. Similar to the other two FFT algorithms, we can reduce the count of multiplies by assigning the multiplications with the twiddle factors to a variable. Since $y(1)$ and $y(3)$ have the same second summand with the only difference being the sign, we can save one addition by storing this value.

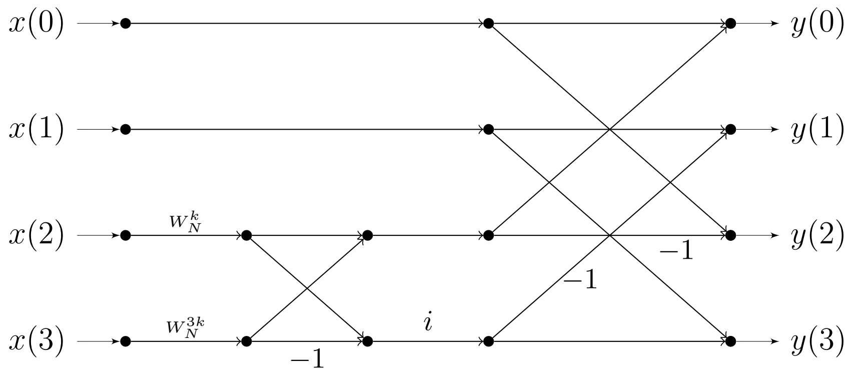


Figure 4.9: 4-Point Split-Radix DIT FFT Butterfly

$$\begin{aligned}
y(0) &= x(0) + x(2) * W_N^k + x(3)W_N^{3k} \\
y(1) &= x(1) - i * (x(2) * W_N^k - x(3) * W_N^{3k}) \\
y(2) &= x(0) - (x(2) * W_N^k + x(3) * W_N^{3k}) \\
y(3) &= x(1) + i * (x(2) * W_N^k - x(3) * W_N^{3k})
\end{aligned} \tag{4.23}$$

5 Experimental Evaluation

5.1 Goal

The goal of the performance analysis is to compare the runtime performance of Radix-2, Radix-4, and Split-Radix FFT algorithms on streaming data under different sized inputs. The accuracy of our implementations has been tested against *jTransforms*[1], an open source FFT library written in Java. The output was also partially cross-validated with *FFTW*[2], a well established software library for FFT computation. In any test with synthetic input data, our algorithms evaluated to a 100% accuracy, contemplating digits of up to 10^{-8}. We conducted two different experiments in order to gain insights of different aspects of the application. The first evaluations were focused on the isolated performance of the FFT algorithms on different sized inputs. Since our FFT algorithms rely on input dimensions that are a power of 2 (or 4 in the Radix-4 case), we mapped the continuously indexed data onto discrete grids of size N^4 for $N = 3, 4, 5, 6$, since powers of 2 are a subset of powers of 4. The second experiments conducted were time measurements of the complete pipeline with focus on the computational expensive parts. The total runtime of the pipeline will be broken down into its summation factors and compared across multiple datasets of different sizes. In Section 5.2 we will cover the experimental settings, including the hardware used for time measurements and descriptions of the tested scenarios.

5.2 Experimental Setting

All evaluations were done on an Intel Core i5-7600(3.50GHz) CPU and 8 GB of RAM with Arch Linux[3] as the operating system.
As described in Section 3.2.2, the data is being processed in time windows. For the evaluation, we set the time to 15 seconds per window, meaning that data is aggregated for 15 seconds until it is mapped onto the grid and processed by the three discussed FFT algorithms. The amount of records arriving per window may vary depending on the throughput of the streaming pipeline, however this variation is not considered to be important in the evaluation of the FFT performances since it does not change the grid size and thus does not influence the operation count of the FFTs. If the throughput would be exponentially high or low during a benchmark, this could influence total running time of the pipeline since the amount of batches received would strongly differ from the rest of the experiments. In order to evaluate the throughput

[1]https://sites.google.com/site/piotrwendykier/software/jtransforms
[2]http://fftw.org/
[3]https://www.archlinux.org/

the complete dataset has been processed $n = 10$ times in advance. For a time window of 15 seconds, the batch-size averaged at 17'000 data records.

The performance metric for the FFTs is the time in milliseconds it takes each algorithm to compute the discrete Fourier Transform on the input with all values of W pre-computed and accessible in constant time. For each grid size we evaluated multiple samples. For the second experiment we evaluated runtime performances for multiple parts of the application separately. The main parts of the application that require processing are the deserialization of the received records, the gridding of the data, and the FFT computations. The time necessary for each of those is measured individually on different sized datasets and presented in a break-down graph

5.3 Results

5.3.1 FFT Algorithm Performance

In the following section a comparison of the runtime for all three FFT Algorithms on multiple input sizes will be presented, as well as a comparison of their theoretical operation count. The algorithms were invoked in an isolated environment and all other variables like streaming throughput or deserialization time have been eliminated by only measuring the time passed while performing the FFTs. For all input sizes we decided to evaluate averages over 20 FFTs in order to achieve a smoother, more representative graph since the individual measurements contained spikes that made it hard to extract information from the graphs. According to our calculations on runtime complexity and complex operation count, we expect Split-Radix to perform best, followed by Radix-4 and Radix-2.

Operation Count

Table 5.1 shows the amount of operations each algorithm computes for the discussed input sizes, here *Adds* and *Mults* correspond to complex additions/subtractions and multiplications, respectively. In this table it is noticeable that the amount of complex additions/subtractions are the same for all algorithms, which is consistent with our calculations in Section 4.2.1 and ...

Size	Radix-2		Radix-4		Split-Radix	
	Adds	Mults	Adds	Mults	Adds	Mults
64	384	192	384	144	384	114
256	2048	1024	2048	768	2048	626
1024	10240	5120	10240	3840	10240	3186

Table 5.1: Operation Count Comparison.

One can clearly see where the difference in running time for the algorithm origins, as the multiplication count differs for all three algorithms. Split-Radix requires the lowest amount of multiplications followed by Radix-4 and Radix-2. In numbers, this corresponds to an improvement of 25% from Radix-2 to Radix-4 over all input sizes. The multiplication count of Split Radix is best expressed by the recursion in Algorithm 7 and is approximately 20% better than Radix-4 at the discussed input sizes.

Algorithm 7 Compute multiplies of Split Radix

```
if n==1 || n==2 then
    $return\ 0$
end if
$return\ n/4 + 2 * rec(n/4) + rec(n/2)$
```

Synthetic Input Data

In order to evaluate the isolated FFT performance we created a test environment which generates synthetic input data of different dimensions. With this testing environment we were able to do experiments of larger sizes compared to what would have been feasible through the pipeline. Recall that the two dimensional FFT procedure for a $n*n$ matrix consists of $2*n$ one dimensional FFT pass-troughs. Thus one could also evaluate the performance on one dimensional testing data and expect similar relations between the runtime of all algorithms. Since our real data is defined in two dimensions, we decided to follow that data structure also for this experiment in order to provide consistency when mentioning input dimension and also to keep time relations consistent. Discussed input dimensions are 64, 256, 1024, and 4096 whereas an input dimension of 64 corresponds to matrix of size 64 by 64 in this context. For each input dimension we evaluated 20 FFT pass-troughs and recorded the average time per Fourier Transform for each FFT algorithm. The results of the experiment are depicted in Figure 5.1 and will be discussed in the following section.
With dimensions of size $4^3 = 64$ all algorithms solved the Fourier Transform in under 7 ms. Radix-2 in average needed 6.9 ms of computation time, whereas Radix-4 and Split-Radix needed 2.65 and 1.15 ms, respectively. This pattern of Radix-2 performing notably worse than the other algorithms continues with increasing input dimension size, as shown in Table 5.2. We expect the effective running times to reflect the differences in theoretical computation count shown earlier. However since the addition count of all three algorithms is the same and the running time consists out of a combination of both additions and summations some deviation is also expected. In fact the ratio between measured times of Radix-2 and Radix-4 seems to approximately reflect the ratio of multiplies for both algorithms, which is 0.75 for $\frac{Radix-4}{Radix-2}$. This is more accurate for larger input sizes. The differences between Radix-4 and Split-Radix are less apparent. Although Split-Radix performs better across all input dimension variations, the relative difference is decreasing with increasing input size, which is consistent with the operation counts in Table 5.1.

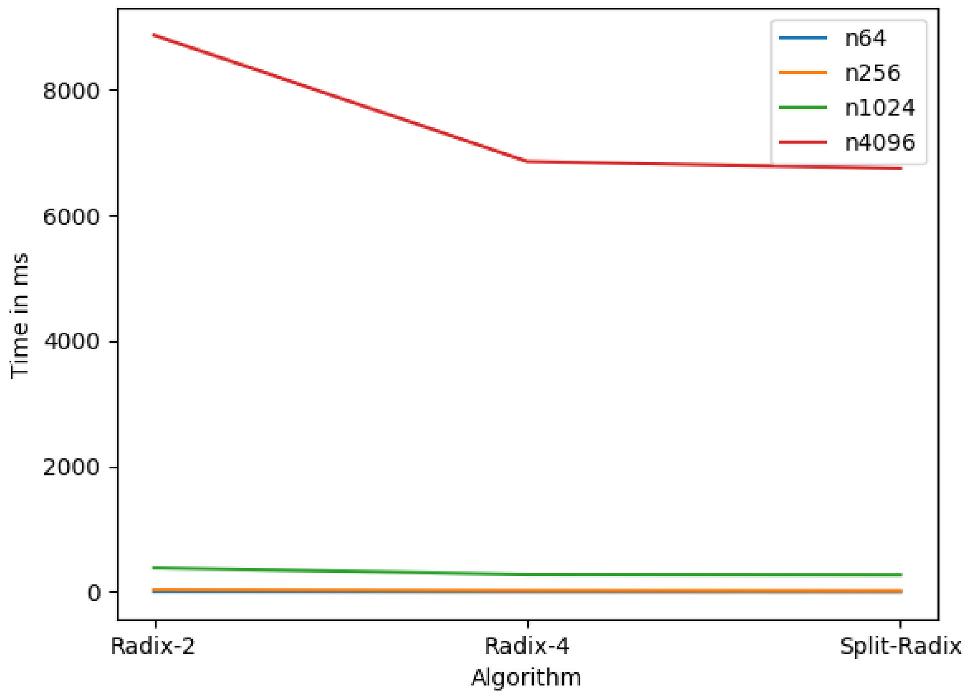
Size	Radix-2	Radix-4	Split-Radix
64	6.9	2.65	1.15
256	34.05	21.55	16.7
1024	381.65	274.85	270.15
4096	8870.85	6858.75	6745.65

Table 5.2: Synthetic data performance times in ms.

5.3.2 Pipeline performance

The goal of the second experiment conducted in this work was to evaluate the performance of each algorithm embedded in the pipeline. In order to individually extract the time consumed for each part of the pipeline, multiple times were recorded. Recall the abstract view of the

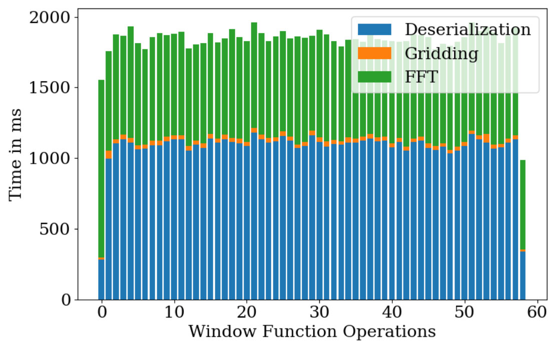
Figure 5.1: Experiment results for different input sizes



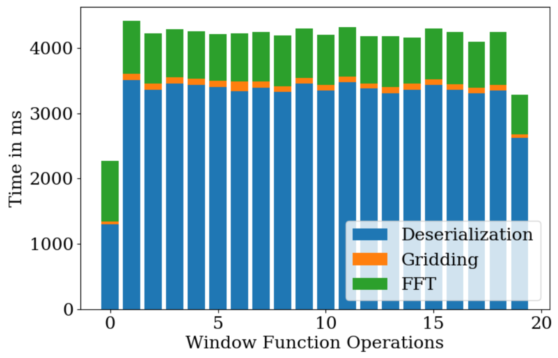
pipeline for each incoming data-point in Figure 3.9. The time necessary to deserialize it was recorded in a first variable. Secondly, the total amount of time that was spent to map each data-point onto the grid is recorded, as well as the time spent computing the FFT. Important variables for this experiment are the time-window size and the grid size, as well as the chosen FFT algorithm. The windows size defines how much data is being processed at the same time. As mentioned earlier, this does not influence the FFT performance which is only dependent on the grid size. It does however define how many times we calculate the FFT for the complete dataset. Thus the window size influences the granularity of our result. With a large window size, we process a large amount of records at once, leading to a reduced runtime since we have to perform less FFTs. However, this also decreases the amount of output data we gather, giving us less intermediate results. With a large window size, we process fewer batches with more data records, which leads to the effect of the deserialization- and griding time consisting out of fewer- but larger chunks. Since every record is deserialized and cast onto the grid exactly once independent of the window size, we argue that this factor does only influence the amount of time spent computing FFTs. This is true as long the processing time of a batch does not grow larger than the window's size, which would induce idle time.

We evaluated this for window sizes of 5, 15, and 30 seconds on the Radix-4 algorithm. The results of those tests can be observed in Figures 5.2a to 5.2c, where the complete dataset has been processed for the aforementioned window sizes. Since each datapoint corresponds to one batch of data contained in one window, the amount of datapoints in the plots decreases with a larger window. As we expected, the total amount of time spent deserializing the data was constant at around 64'000 ms, with a maximum delta of 897 ms. The measurements also support our assumption that the window size does not influence the total gridding time as it stayed at a constant 1700 ms with maximum deviation of 78 ms. These numbers can be derived from Figures 5.2a to 5.2c by addition of the deserialization/gridding times of all measured points on the x-axis. Logically, we have less measured points for larger window sizes, as we process more records from the dataset at once. As to be expected the largest window size leads to the lowest overall runtime as we compute less FFTs in total, as explained earlier in this section.

(a) Window Size 5



(b) Window Size 15



(c) Window Size 30

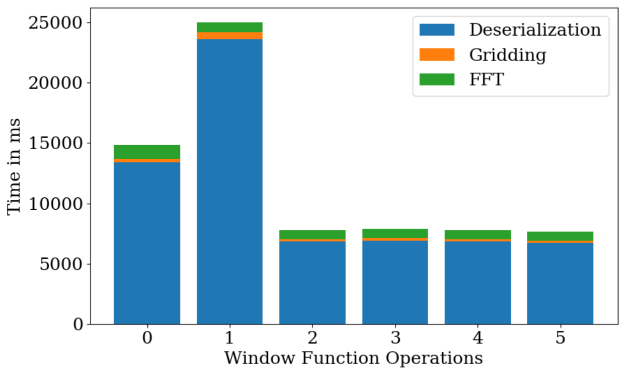
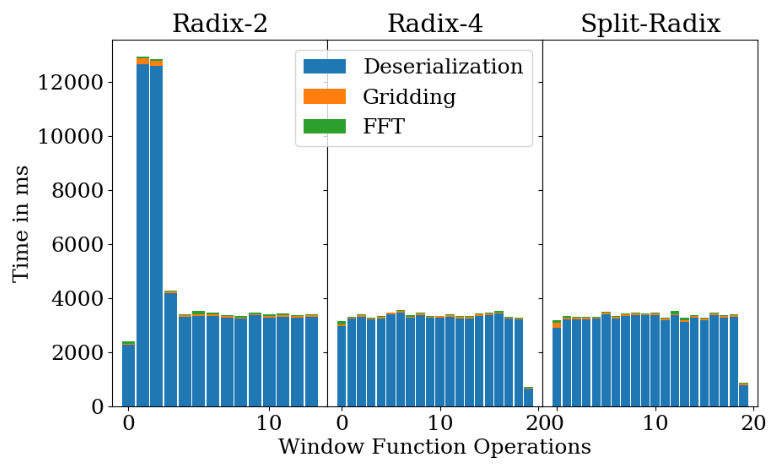


Figure 5.2: Time break down for window sizes of 5, 15, and 30 seconds.

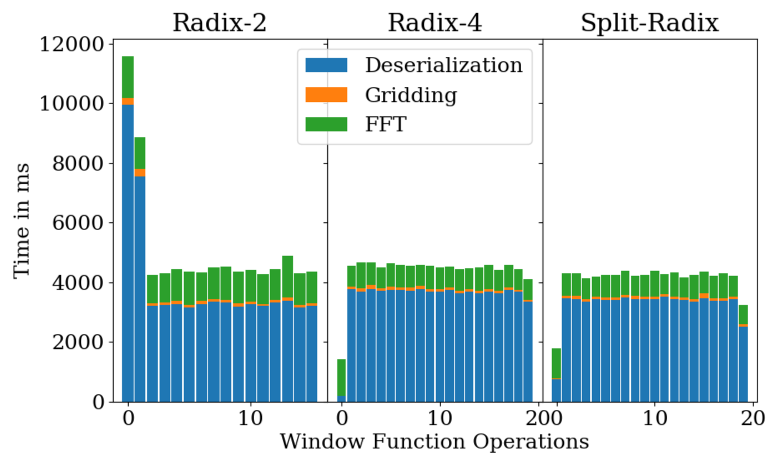
In contrast to the window size, the grid size does directly influence the computation cost of the FFT algorithms and thus is our main variable for this experiment. It is also important to note that the grid size may drastically influence the quality of the output of the pipeline. Choosing a small grid will be computationally fast, but may lead to losing accuracy as multiple coordinates could be mapped onto the same discrete set of coordinates, whereas a larger grid will be more accurate but increase FFT runtime and memory consumption of the application. In order to discuss the direct influence of different sized grids, we chose the same sizes as in Section 5.3.1, without the largest size of $4^6 = 4096$ and evaluated the times for the individual

steps presented above for each of those. The largest size was left out since it introduced major computation spikes due to CPU and memory limitations of the test hardware. We chose a fixed window size of 15 seconds for this experiment.
On the smallest grid size, the deserialization process is overwhelmingly more expensive than both the gridding and the FFT computation, as seen in Figure 5.3a. The spike in Radix-2 is explained by a larger amount of records in the corresponding time windows and thus larger batch sizes for those windows. This also shows how the deserialization and the gridding are directly dependent on the amount of records in a window, whereas we don't recognize a change in FFT calculation time. This spike in batch size also leads to the experimental data generated from the Radix-2 sample having less output values, since a large amount of data is being processed in one step. We can observe the same phenomenon in Figure 5.3b for Radix-2. We can clearly observe a shift in the composition of the total runtime here. The FFT time was negligible for input size 64 and now makes up a notable amount of the sum. This trend continues in Figure 5.3c where the FFT time is the dominant part of the sum. This graphic further strengthens the argument that the amount of records mainly influences the deserialization time and the gridding time by a smaller extent. In these samples one can observe a highly volatile record size and its influence on the different times. We argue that this is due to higher memory consumption on larger grid sizes, as a similar image could be replicated by multiple runs of the application with the same parameters.

(a) Input dimension 64 by 64



(b) Input dimension 256 by 256



(c) Input dimension 1024 by 1024

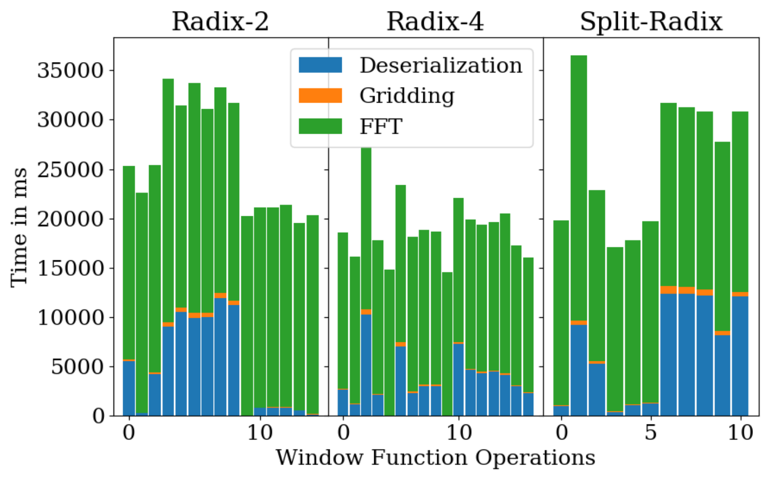


Figure 5.3: Time break down for grid sizes $64, 256, \&1024$.

6 Conclusion

We discussed the performance of three FFT algorithms to perform the Fourier Transformation on complex streaming data. Streaming environments come with multiple benefits and challenges. They allow for real-time or close-to-real-time computations on continuously generated data. In order to provide processing of the incoming data as fast as possible, it is vital to reduce runtime complexity and operation count for every procedure that is invoked in such an environment. We built a streaming environment for data from the radio astronomy domain, where the Fourier Transform is an often used algorithm in the process of generating images from data.
All discussed algorithms fall into the same category of divide-and-conquer and have a very similar structure. The main differences in performance come from the ability to re-use certain computations, which is exploited to a different degree in all three algorithms. In our theoretical operation count analysis we noted that all three algorithms that we implemented perform the same amount of complex additions. The performance difference origins in the amount of complex multiplications. Radix-2 uses the most multiplications, followed by Radix-4 and lastly Split-Radix with the difference of the latter two being very small in our implementations.
We furthermore conducted experiments in order to evaluate the actual runtime for all algorithms in an isolated environment. The results from those experiments are consistent with the theoretical evaluation in terms of performance ranking. Also the relative differences in runtime are closely related to the relative differences in operation count. Radix-2 performed notably worse than its competitors, while Split-Radix outperformed Radix-4 only by a very small margin. The same result could be observed when the algorithms were embedded in the streaming pipeline. The dataset could be processed the fastest using Split-Radix, when keeping all other parameters stable. Our results show that, although optimized from the initial procedure, the deserialization of the data in the stream processor is taking a large amount of time relative to the FFT computation, especially for smaller grid sizes. Thus it might be beneficial for a real world application to optimize the data already when it is created, in order to reduce serialization overhead and also to reduce network usage in a distributed scenario. Our experimental evaluation also showed how the window size influences the total running time and the output of the pipeline. With a larger window size we can process more data at once, leading to a reduced total runtime but also reduced amount of intermediate results and thus less information.

Bibliography

[1] Apache flink documentation. `https://ci.apache.org/projects/flink/flink-docs-release-1.5/`. Accessed: 2018-07-25.

[2] Pawsey. `https://www.pawsey.org.au/2017/01/live-askap-data-hits-galaxy/`. Accessed: 2018-07-09.

[3] G. Bergland. A guided tour of the fast fourier transform. *IEEE spectrum*, 6(7):41–52, 1969.

[4] R. N. Bracewell. The fourier transform. *Scientific American*, 260(6):86–95, 1989.

[5] W. T. Cochran, J. W. Cooley, D. L. Favin, H. D. Helms, R. A. Kaenel, W. W. Lang, G. C. Maling, D. E. Nelson, C. M. Rader, and P. D. Welch. What is the fast fourier transform? *Proceedings of the IEEE*, 55(10):1664–1674, 1967.

[6] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[7] M. Frigo and S. G. Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216–231, Feb 2005.

[8] M. T. Heideman, D. H. Johnson, and C. S. Burrus. Gauss and the history of the fast fourier transform. *Archive for history of exact sciences*, 34(3):265–277, 1985.

[9] Jeff Kern. Measurementset definition. `https://casa.nrao.edu/casadocs/casa-5-1.2/reference-material/measurement-set`, 2017. Accessed: 2018-07-11.

[10] S. G. Johnson and M. Frigo. A modified split-radix fft with fewer arithmetic operations. *IEEE Transactions on Signal Processing*, 55(1):111–119, 2007.

[11] R. Neuenfeld, M. Fonseca, and E. Costa. Design of optimized radix-2 and radix-4 butterflies from fft with decimation in time. In *2016 IEEE 7th Latin American Symposium on Circuits Systems (LASCAS)*, pages 171–174, Feb 2016.

[12] D. N. Rockmore. The fft: an algorithm the whole family can use. *Computing in Science Engineering*, 2(1):60–64, Jan 2000.

7 Appendix

Listing 7.1: Radix-2 implementation in Java

```
/**
 * complex-based recursive cooley-tukey radix-2 dit fft
 * @param x: Complex[] with length of power of 2
 * @return Fourier-transformed array of same length as x
 */
public static Complex[] fft2(Complex[] x)
{
   int N = x.length;
   //Check for power of 2 and get power of current size
   int log2n=-1;
   for(int i=0;i<=MAX_POW_2;i++){
      if(pow2[i]==x.length){
         log2n=i;
      }
   }
   if(log2n ==-1){
      throw new RuntimeException("Length of input is not a power
         of 2, n = "+N);
   }
   //Base case
   if (N == 1)
      return new Complex[] { x[0] };
   //split Array into new subarrays, recursively evaluate fft of
      those
   Complex[] even = new Complex[N / 2];
   for (int k = 0; k < N / 2; k++)
   {
      even[k] = x[2 * k];
   }
   Complex[] q = fft2(even);

   Complex[] odd = new Complex[N / 2];;
   for (int k = 0; k < N / 2; k++)
   {
      odd[k] = x[2 * k + 1];
   }
   Complex[] r = fft2(odd);
```

```
        Complex[] y = new Complex[N];
        //Combine
        for (int k = 0; k < N / 2; k++)
        {
            //Radix-2 Butterfly
            Complex wk = twiddleTable[log2n][k];

            Complex oddVal = wk.multiply(r[k]);
            y[k] = q[k].add(oddVal);
            y[k + N / 2] = q[k].subtract(oddVal);
            mults +=1;
            adds+=2;
        }
        return y;

    }
```

Listing 7.2: Radix-4 implementation in Java

```
   /**
     * Complex-based Cooley-Tukey Radix-4 dit FFT
     * @param x: Complex[] with length of power of 4
     * @return Fourier-transformed array of same length as x
     */
    public static Complex[] fft4(Complex[] x){

        int N = x.length;
        //Power of 4 check, get current power of 4
        int log2n=-1;
        for(int i=0;i<=MAX_POW_2;i++){
            if(pow2[i]==N){
                log2n=i;
            }
        }
        if(log2n== -1 || log2n%2!=0){
            throw new RuntimeException("Length of input is not a power
                of 4");
        }
        int N4 =N/4;
        //Base case
        if(N ==1){
            return new Complex[] {x[0]};
        }
        //Split array into four subarrays, recursive calls
        Complex[] first = new Complex[N/4];
        Complex[] second = new Complex[N/4];
        Complex[] third = new Complex[N/4];
        Complex[] fourth = new Complex[N/4];
        for(int k=0;k<N4;k++){
           first[k]=x[k*4];
           second[k]=x[k*4+1];
           third[k]=x[k*4+2];
           fourth[k]=x[k*4+3];
        }
        Complex[] q = fft4(first);
        Complex[] r = fft4(second);
        Complex[] s = fft4(third);
        Complex[] t = fft4(fourth);

        Complex[] y = new Complex[N];
        //Combination
        for (int k = 0; k < N / 4; k++)
        {
           //Radix-4 butterfly
           Complex wk = twiddleTable[log2n][k];
           Complex wk2 = twiddleTable[log2n][2*k];
           Complex wk3 = twiddleTable[log2n][3*k];

           Complex a = q[k];
           Complex b = r[k].multiply(wk);
           Complex c = s[k].multiply(wk2);
           Complex d = t[k].multiply(wk3);

           Complex t1 = a.add(c);
           Complex t2 = a.subtract(c);
           Complex t3 = b.add(d);
           Complex t4 = b.subtract(d);

           y[k] = t1.add(t3);
           y[k+N4] = t2.add(new
              Complex(t4.getImaginary(),-t4.getReal()));
           y[k+2*N4] = t1.subtract(t3);
           y[k+3*N4] = t2.add(new
              Complex(-t4.getImaginary(),t4.getReal()));
           mults+=3;
           adds+=8;

        }
        return y;

     }
```

Listing 7.3: Split-Radix implementation in Java

```java
/**
 * Split Radix 2/4 DIT recursive
 * based on "A modified split-radix FFT with fewer arithmetic
   operations"
 * @param x: Complex[] with length of power of 2
 * @return y Fourier-transformed array of same length as x
 */
public static Complex[] fftSplit(Complex[] x){
   int N = x.length;
   //Find Log2 for twiddle factor lookup and check if input is
      of valid length
   int log2n=-1;
   for(int i=0;i<=MAX_POW_2;i++){
      if(pow2[i]==N){
         log2n=i;
      }
   }
   if(log2n== -1){
      throw new RuntimeException("Length of input is not a power
         of 2");
   }
   int N4 =N/4;
   //Base case
   if(N ==1){
      return new Complex[] {x[0]};
   }else if(N==2){
      Complex y0=x[0].add(x[1]);
      Complex y1=x[0].subtract(x[1]);
      adds+=2;
      return new Complex[] {y0, y1};
   }
   //Split input into three subarrays of length N/4, N/4, N/2
      and make recursive calls
   Complex[] first = new Complex[N/2];
   Complex[] second = new Complex[N/4];
   Complex[] third = new Complex[N/4];
   for(int k =0;k<N/2;k++){
      first[k] = x[2*k];
   }
   for(int k=0;k<N4;k++){
      second[k]=x[k*4+1];
      third[k]=x[k*4+3];
   }

   Complex[] q = fftSplit(first);
```

```
      Complex[] r = fftSplit(second);
      Complex[] s = fftSplit(third);

      Complex[] y = new Complex[N];
      //Combination
      for(int k=0;k<N4;k++){
      //Split-Radix Butterfly
         Complex wk = twiddleTable[log2n][k];
         Complex wk2 = twiddleTable[log2n][3*k];

         Complex a1 = q[k];
         Complex a2 = q[k+N4];
         Complex b = r[k].multiply(wk);
         Complex c = s[k].multiply(wk2);
         Complex temp = new Complex(0,1).multiply(b.subtract(c));
         Complex temp2 = b.add(c);

         y[k] = a1.add(temp2);
         y[k+N/2] = a1.subtract(temp2);
         y[k+N4] = a2.subtract(temp);
         y[k+3*N4] = a2.add(temp);
         adds+=6;
         mults+=2;
      }
      return y;
```