Department of Informatics, University of Zürich

BSc Thesis

Implementation of Single-Point Discrete Fourier Transform on two dimensional data

Nicolas Spielmann

Matrikelnummer: 15-704-505

Email: nicolas.spielmann@uzh.ch

May 3, 2019

supervised by Prof. Dr. M. Böhlen and M. Saad

University of Zurich[UZH]

Department of Informatics

Acknowledgements

Firstly, I'd like to express my sincerest gratitude to my supervisor Muhammad Saad for providing valuable 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 is an algorithm that is used to transform the time representations of a function into its frequency components under the assumption that this given function is periodic. It is the base of many modern applications that use signal processing. Currently the fast Fourier transform algorithm, which reduces the run time of the discrete Fourier transform from $O(n^2)$ to $O(n * log(n))$ is the fastest way to compute the Fourier transform.

In order to calculate the Fourier transform of two dimensional data, the algorithm will just be used to transform all rows and then transform all columns. One use case of the two dimensional algorithm is to generate dirty sky images form radio astronomy data.

In this thesis a faster algorithm to compute the single point two dimensional Fourier transform is described and it is mathematically proofed. This new algorithm will describe all fields of the Fourier transform with its first row or column of the transformed grid. Furthermore the importance of Fourier transform in stream pipelines is considered and this new algorithm is implemented using the stream processing environment Apache Flink. After the initial implementation of the new algorithm, some stream specific optimization have been made to the algorithm. While computing the Fourier Transform, a lot of computations are made a multiple time. This fact is used to further improve the algorithm.

These different Algorithms are then evaluated using artificially generated source files to determine the run time. This shows that the new algorithm, for the biggest grid tested, is 60 times faster than the normal FFT implementation. With the small stream specific enhancements, the algorithm is even up to 120 times faster for the largest grid tested.

Zusammenfassung

Die diskrete Fourier Transformation ist ein Algorithmus, der benutzt wird, um aus Datenpunkten zu verschiedenen Zeiten die Frequenz-Komponenten eines Signals zu ermitteln, unter der Annahme das Signal sei periodisch. Sie ist die Basis vieler modernen Applikationen die Signalverarbeitung machen. Zurzeit ist die schnelle Fourier Transformation, welche die Laufzeit von der diskreten Fourier Transformation von $O(n^2)$ zu $O(n * log(n))$ reduziert der schnellste Weg die Fourier Transformation zu berechnen.

Um die Fourier Transformation von zwei dimensionalen Daten zu berechnenen, wird der eindimnensional Algorithmus benutzt um zuerst alle Reihen und dann anschliessend alle Spalten zu transformieren. Eine Anwendung des zweidimensionalen Algorithmus ist es, aus Radioastronomiedaten verunreinigte Bilder des Himmels zu machen.

In dieser Arbeit wird ein schnellerer Algorithmus für die Berechnung der zwei dimensionalen Fourier Transformation für einen Punkt beschrieben und mathematisch bewiesen. Der neue Algorithmus wird alle Ergebnisfelder der Transformation durch die Felder der Transformation der ersten Reihe oder der ersten Spalte ausdrücken können. Weiteres wird die Wichtigkeit der Fourier Transformation in Stream Pipelines berücksichtigt und der neue Algorithmus wurde in der Stream Processing Umgebung Apache Flink implementiert. Nachdem der ursprüngliche Algorithmus implementier wurde, wurden einige Stream spezifischen Anpassungen gemacht, um die Performance zu optimieren. Währendem die Fourier Transformation berechnet wird, werden viele Rechnungen mehrmals gemacht. Wir haben diesen Fakt benutzt, um den Algorithmus weiter zu verbessern.

Diese verschiedenen Algorithmen sind dann evaluiert worden mit einem künstlich generierten Datenset um die Laufzeit jener zu bestimmen. Das ergab, dass die ursprüngliche Implementation des neuen Algorithmus für das grösste Raster 60-mal schneller ist als die normale FFT Implementation. Mit kleinen stream spezifischen Anpassungen ist der Algorithmus sogar bis zu 120-mal schneller für das grösste getestete Raster.

Contents

List of Figures

List of Tables

Listings

1. Introduction

Multiple scientific fields rely on continuously observing certain object and processing the generated data. With the innovation in the current observation technologies, such as telescopes, the amount of data generated tends to increase. This demands the use of data streaming platforms and stream environment to allow real time processing of the data. Apache Flink is a stream processing framework ruining on java and allowing real time processing of stream data.

One concrete scientific field that could benefit from stream processing is astronomy. For example the Australian Square Kilometer Array Pathfinder (ASKAP) is one of the leading radio telescope facilities in the world. It generates roughly 2.5 GB/s of data, what adds up to nearly 216 TB a day[3]. On this data Fourier transform has to be applied to generate dirty sky images.

Fourier transform is used to decompose a signal over time in its constituent frequencies[4]. When looking at astronomy data, Fourier transform needs to be applied to generate dirty sky images out of the data received to enable further processing of the data. In the process of calculating the Fourier transform on a stream, for every single data point of the input, the Fourier transform will be calculated, and its output will add up to the final transformed image. In order to successfully transform a datastream, such a stream processing platform needs to be robust against failure of machines or the connecting network, since the output is dependent of every input point. Any loss in data will result in a distortion of the result.

Our goal was to implement a high performance single point discrete Fourier transform to be able to use the output data in a stream fashion instead of saving a certain amount of data in batches and then calculation the Fourier transform for each batch of data.

2. The Fourier Transform

The Fourier transform is used in many different fields of science. It can be used to process different type of wave-forms such as acoustical, electrical or optical signals[4]. The Fourier transform has been used since its discovery to break up a given function or signal in sine and cosine that add up to an alternate representation. It shows, that any waveform can be written as a sum of sinusoidal functions. For a continuous Signal the Fourier transform is defined as (2.1).

$$F(jw) = \int_{-\infty}^{\infty} f(t)e^{-jwt}dt \tag{2.1}$$

with $-\infty < f < \infty, -\infty < t < \infty$ and $j = \sqrt{-1}$. The $F(jw)$ is the Fourier-transformed output, whereas $f(t)$ is the input signal at a given time t.

This formula can not be used in a computer environment since its input and output are both in a continuous domain. To be able to analyze sampled signals using a computer system, the signal must be transformed in a discrete domain and then the discrete Fourier transform has to be used[5].

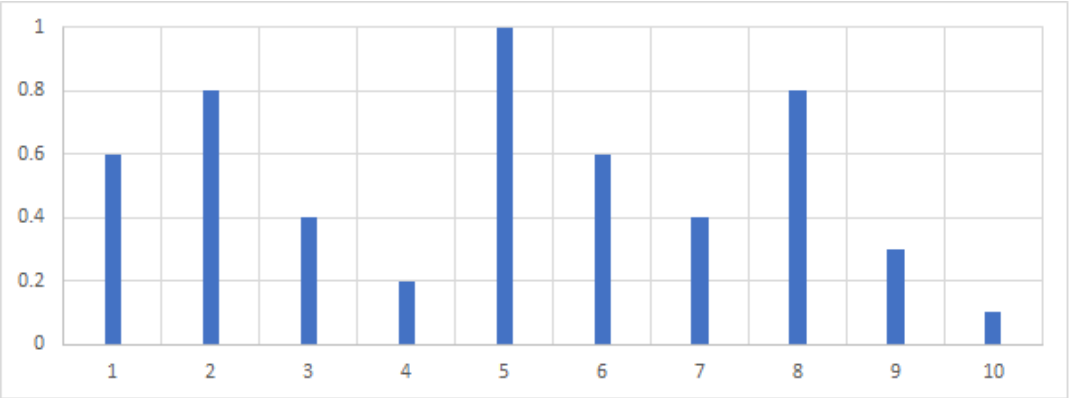
2.1. Discrete Fourier transform

The discrete Fourier transform takes a sequence of a finite length on a time domain as input and produces a frequency-domain finite length sequence as output. For this a signal is sampled at N instants separated by the sample time T. So the samples of the continuous signal $f(t)$ will be called $f[0], f[1], f[2], f[3], ..., f[k], ..., f[N-1]$. So the integral will only be calculated at each sampling point[5]. The Fourier transform of such a signal will be as shown in (2.2)

$$\begin{aligned} F(jw) &= \int_{0}^{(N-1)T} f(t)e^{-jwt}dt \\ &= f[0]e^{-j0} + f[1]e^{-jwT} + ... + f[k]e^{-jwkT} + ... + f[N-1]e^{-jw(N-1)T} \\ & \quad i.e. \quad F(jw) = \sum_{k=0}^{N1} f[k]e^{-jwkT} \end{aligned} \tag{2.2}$$

Since the interval of the sequence of data analysed is not anymore infinite, the discrete Fourier transform treats the data as if it were periodic (for example $f(0)$ to $f(N-1)$ is the same as $f(N)$ to $f(2N-1)$.

(a)



(b)

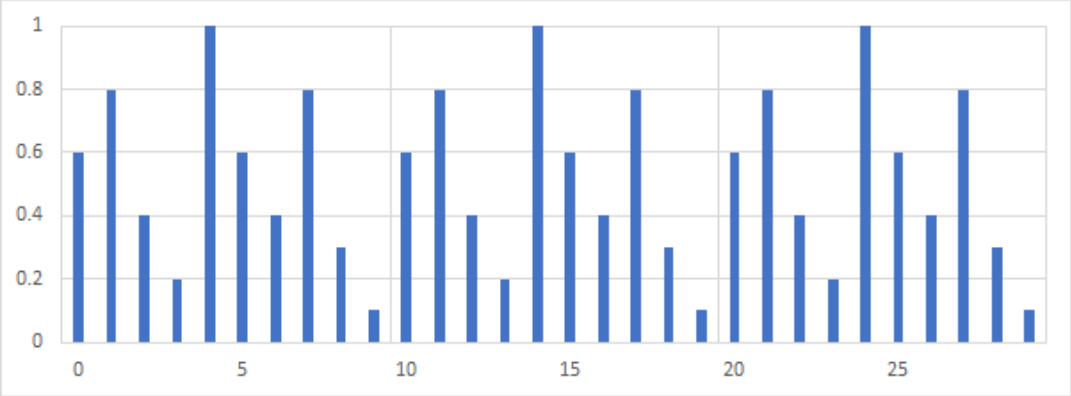


Figure 2.1.: (a) Sequence of N = 10 Samples. (b) periodicity in DFT

Since the data is treated as if it were periodic, the discrete Fourier transform equation is evaluated for the fundamental frequency and its overtones. The fundamental frequency corresponds to one cycle per sequence or $\frac{2\pi}{NT}$rad/sec. The overtones are the multiple of the fundamental frequency.This results in definition (2.3) for w:

$$w = 0, \frac{2\pi}{NT}, \frac{2\pi}{NT} \times 2, ..., \frac{2\pi}{NT} \times n, ..., \frac{2\pi}{NT} \times (N-1) \tag{2.3}$$

So in general the discrete Fourier transform for a sequence $f[k]$ is defined in the Equation (2.4).

$$F[n] = \sum_{k=0}^{N-1} f[k] e^{-j\frac{2\pi}{N}nk} \quad where\, n = 0 \;\; to \;\; N-1 \tag{2.4}$$

$e^{-j2\pi/N}$ can be re written as the W_N, the principal Nth root of unity. A Nth root of unity is a complex number that results in 1 when it is raised by N. $e^{-j2\pi/N}$ is an Nth root of unity since $e^{-j2\pi/N}$ raised by N results in 1 as shown in (2.5)[5].

$$e^{-j2\pi/N*N} = e^{-j2\pi} = \cos(-2\pi) + j * \sin(-2\pi) = 1 \tag{2.5}$$

W_N is a so called twiddle factor[6]. When the formula for discrete Fourier transform is rewritten using the so called twiddle factor W_N (2.6), it is easier to see, that a lot of computations of these twiddle factors can be easily reused and not computed again, in order to reduce the complex additions and trigonometric functions that are used to calculate W_N^{nk}, since the integer product nk may be repeated by different values of n and k during the computation and W_N^{nk} is periodic and will only have N distinct values[5]. The so called fast Fourier transform algorithm, discussed in 2.2, use this principle to store precomputed twiddle factors.

$$F[n] = \sum_{k=0}^{N-1} f[k] e^{-j\frac{2\pi}{N}nk} = \sum_{k=0}^{N-1} f[k] W_N^{nk} \quad where\, n = 0 \;\; to \;\; N-1 \tag{2.6}$$

Since W_N is the Nth root of unity, its values are symmetric such as $W_N^x = -W_N^y - x$. For an input of size 8 the twiddle factors would be as described in (2.7).

$$\begin{aligned} W_8^4 &= -W_8^0 \\ W_8^5 &= -W_8^1 \\ W_8^6 &= -W_8^2 \\ W_8^7 &= -W_8^3 \end{aligned} \tag{2.7}$$

This definition of the discrete Fourier transform can be implemented in an algorithm such as Algorithm 1. We assume the size of the input Array can be accessed using $x.size()$.

Algorithm 1 calculate dft(x)

```
N ← x.size()
result ← Complex[N]
for k = 1 to N do
    result[k] ← 0
    for t = 1 to N do
        w ← e^(−2jπ/N)tk
        result[k] += x[t] * w
    end for
end for
return result
```

For 1-D data the discrete Fourier transform algorithm has a complexity of $O(N^2)$.

2.1.1. 2D Discrete Fourier transform

The two dimensional discrete Fourier transform is defined as in (2.8).

$$F(u,v) = \sum_{m=0}^{N-1}\sum_{n=0}^{N-1} f(m,n)e^{-j2\pi\frac{um}{M}+\frac{vn}{N}} \quad where\ u,v = 0\ \ to\ \ N-1 \tag{2.8}$$

This two dimensional discrete Fourier transform is carried out by applying the one dimensional discrete Fourier transform to all rows of the input and then either transposing the resulting matrix and transform the rows again or just transforming the columns of the resulting matrix. The order of the calculations steps are not important. The algorithm 2 will compute the two dimensional discrete Fourier transform for an input matrix x. We assume we can aces the nth row of the matrix using $x.row[n]$ and aces the nth column using $x.column[n]$. Furthermore, the size of the row can be computed using $row.size()$.

Algorithm 2 calculate 2d dft(x)

$N \leftarrow x.column[1].size()$
$M \leftarrow x.row[1].size()$
for i = 1 to N **do**
  $x.row[i] \leftarrow dft(x.row[i])$
end for
for i = 1 to M **do**
  $x.column[i] \leftarrow dft(x.column[i])$
end for
return x

2.2. Fast Fourier transform

The fast Fourier transform is a method, that allows to compute the discrete Fourier transform in a more efficient way. Cooley and Tukey describe such an algorithm in their 1965 published paper "An algorithm for the machine calculation of complex Fourier series"[7]. They were able to reduce the number of complex additions and multiplications from $O(n^2)$ to $O(n * log(n))$, by splitting the problem with size N recursively in n smaller problems of size N/n, until the solving of the split problem gets trivial. They did also rely on the symmetric properties of the twiddle factors described in 2.1. This resulted in the so called Cooley-Tukey FFT algorithm[7].

3. Streaming Pipeline

Data stream platforms allow data sources producing data to provide data in a continuous flowing manner to other applications or data sinks. Most of the time, such data streams contain raw data, that needs some processing steps, until it can be consumed by a data sink or further processed by an other application. this so called stream processing can be done by a stream processor. There are many concerns using a streaming environment, since a failure of the network or a machine involved in the whole stream may result in loss of data or duplicated processing of a single record. Once a failure has occurred, the process must be restarted and its state needs to be recovered, since a loss of computations would result in lost output data and the recalculation of lost computations. This would make a failure a fatal risk. Such an open source stream processing platform is Apache Flink [1].

It can be used to perform a variety of operations on the streamed data in real-time or close-to-real-time. Apache Flink can process *bounded* or *unbounded* data streams. For this purpose it provides a DataSet API for *bounded* (batch) processing and an DataStream API for *unbounded* (stream) processing. Flink will provide a data flow graph consisting of statefull operators and data streams form a source or output of an operator. In order to be fault tolerant, Flink offers processing with strict only-once-processing guarantee, as it deals with failures via checkpointing, restoration of checkpoint and then partial re-execution. It also takes snapshots of the state of the operator and its current position of input stream to reduce recomputing when a failure occurred [8]. A basic streaming platform consists of a data source, a stream processor and a data sink. In our case the built in function of Apache Flink is used to read a file as the data source, use the operators of Apache Flink to process the data and then write the results back to a file sink. Our architecture can be seen in Figure 3.1.

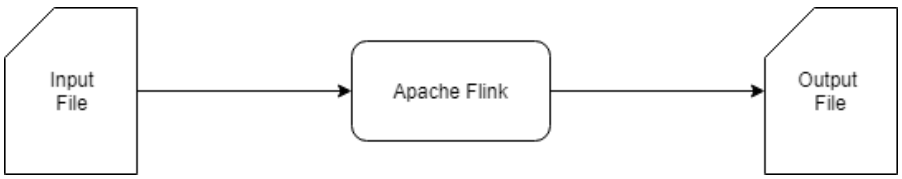


Figure 3.1.: Architectual overwiev

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

3.1. Dataset

In order to just evaluate the performance of the new implemented two dimensional Fourier transform, it was agreed on using a randomly generated input file as our data set, since one of the goals was to evaluate the performance for a variety of different grid sizes. Therefore a python script (Listing 3.1) is used to generate a file with a n amount of lines. Every line represents the value received at a given coordinate. The values are uniform values from -10 to 10. Generally, the random module is used to generate random integers or floats. So the amount of lines in the file corresponds to points that will be processed in our stream processor.

Listing 3.1: Source file generator

```
import random

size = 4096
f= open("Test"+str(size)+"_long.txt","w+")
for i in range(10000):
    f.write(str(random.randint(0,size-1)) + ","
    + str(random.randint(0,size-1)) + '='
    + str(random.uniform(-10, 10)) + "\n")
f.close()
```

This will result in a text file such as shown in Listing 3.2

Listing 3.2: Snippet of source file

```
1303,143=5.922648574340709
802,2497=-5.74036278862879
1853,421=-7.033751721577319
1426,124=-7.766737416794996
1760,1592=-2.6279102211014234
```

A data set for every grid size that will be evaluated will be generated. So there will be a data set for every grid size $N = 2$ to $N = 4096$. This data files will then be read by Apache Flink. It will be declared as the stream source. The file will then be read from Flink and every line of the file will be emitted as a single data point in a stream. In the next Section (3.2) it will be further evaluate how the data will be represented while it is processed.

3.2. Data Model and Processing functions

Apache Flink offers a variety of different operation that can be applied to data stream and will then result in another data stream or various other types of streams. One category of operators are the data stream transformations. The map function 3.2.2 will be used to process our strings, that are streamed from the input file. Then a regular expression is used to split the input of our map function to extract the coordinate values X and Y and the value V at the position (X, Y) that is updated with the line of the input file[9]. With the usage of the Tuple3[10] datatype provided by Apache Flink the input stream will now be represented as a tuple with 3 values.

The data types integer are used to represent the X and Y values and the double datatype is used to represent the value at the point(X, Y).

The next step in our processing pipeline is to compute the Fourier transform for every data point in our input stream. Since we wanted to evaluate different ways to compute the single point discrete Fourier transform, we will have different ways, how the data stream will be processed. These different ways will be presented in Subsection 3.2.2 to Subsection 3.2.5. To evaluate the performance, our implementation will also be compared to a fast Fourier transform implementation embedded in our processing pipeline.

3.2.1. Data Classes

For all the Implementations the same two classes will be used to represent and process the data.

Complex Number

Since the Fourier transform uses complex numbers consisting of a real part and an imaginary part, a class to represent the complex numbers will be needed. This data model is represented in Listing 3.3.

Listing 3.3: Data Model of Complex as a .java class

```
public class Complex implements Serializable {
    private final double re;   // the real part
    private final double im;   // the imaginary part

    // create a new object
    //with the given real and imaginary parts
    public Complex(double real, double imag) {
        re = real;
        im = imag;
    }
    ...
}
```

This class will also provide all functionalities needed during the calculation of the Fourier transform, mainly addition an multiplication of complex numbers. The addition of a real only version of multiplication was made, since the input values multiplied with the complex numbers tend to be real an therefore non complex. This reduces the computational power required.

Since the precision of floating point numbers, such as the datatype float or double in java, is not unlimited, a so called underflow may occur when calculations are done with very small numbers. This produced some error when comparing complex numbers. In order to solve this problem, a certain precision was added for the comparison of complex numbers. In order to achieve this, it is checked , whether the absolute amount of the difference of the real numbers

and the imaginary numbers of the compared complex numbers are within a certain area. A reasonable precision is 10^{-10} and therefore this level of precision is used.

Output Plane

The result of the Fourier transform will be a grid of complex numbers the same size as the input grid where measurements were taken. Therefor a data model for a grid of complex number needs to be made. This was implemented by making a class called OutputPlane consisting of a two dimensional array of complex numbers and two integer row and column to store the size of the plane.

Listing 3.4: Data Model of OutputPlane as a .java class

```
public class OutPutPlane implements Serializable {
    private Complex[][] Plane ;
    private int row;
    private int column;

    public OutPutPlane( int row, int column, boolean init){
        Plane = new Complex[row][column] ;
        this.row = row;
        this.column = column;
        if (init) {this.intialize();}
    }
    ...
}
```

Since this data model is either used to pass the values of a single point Fourier transform or to sum up all the values of each points Fourier transform, a boolean is added to the constructor to specify whether the plane of complex numbers should be initialized with complex numbers of value $(re = 0,\ im = 0)$, so it can be used to sum up values in order to get the final values of the Fourier transform or it should not be initialized, since the values of each complex number will be passed by a object reference to later add to the final result plane.

3.2.2. Discrete Fourier transform

For the discrete Fourier transform we will use two nested loops to calculate the Fourier transform for every point of the grid. This way an initialization of the grid is not necessary. Since there is only one point of the grid with a value non zero, the discrete Fourier transform for every point in the plane can be described as in Equation 3.1 where N stands for the length of rows and M stands for the length of columns.

$$F(u,v) = V(x,y) * (cos(-2\pi\frac{xu}{N}) + j * sin(-2\pi\frac{xu}{N})) * (cos(-2\pi\frac{yv}{M}) + j * sin(-2\pi\frac{yv}{M})) \tag{3.1}$$

This calculations will be done in a map function. The plane with the results of the transform will then be added to a class variable of type OutputPlane to sum up the transforms of every

single point. For every input tuple the plane with the sum of all transform until this point will be emitted. The complete algorithm will be be described in Section 4.1 .

Map Function

a map function in Apache Flink is a function that processes an element of a stream and produces exactly one output element. In Apache Flink, such operators are implemented by classes, which need to implement certain interfaces. In our case the map function. Once a stream is needs to be processed, for every parallel instance of the stream, an instance of the class implementing the map function is instantiated. This allows to pass some information to the class in its constructor and also store some values as class attributes, such as grid size. Furthermore, the summation of a whole OutputPlane can be done by declaring a class variable of this type[2]. A basic implementation of a map function is shown in listing 3.5.

Listing 3.5: Example for a Map Function [2]

```
new MapFunction<Integer, Integer>() {
    @Override
    public Integer map(Integer value) throws Exception {
        return 2 * value;
    }
}
```

3.2.3. Discrete Fourier transform with rotation

As mentioned in Section 3.2.2, the discrete Fourier transform of a single point will only be dependent on the value of the input and the coordinates of the input. This fact, combined with the periodic attribute of the Nth root of unity will lead to the fact, that every value of the Fourier transform for a single Point will either be in the first row or first column of the transformed plane, as long as the grid has the same length as width and the length is a power of 2. The other one (row or column) will be a subset of the row or column that contains all values. All point in the transformed grid can then be expressed using the first row or column and apply the shift $n - 1$ times, where n stands for the index of the row or column. Lets assume an example where the first row contains all values and the amount of shift is two. The first value of the third row will then be the fifth value of the first row. Mathematical proof will be done in Section 4.2. Such an example is illustrated in Figure 3.2.

Due to this properties, only the values of the first row and column will be computed and the type of shift[2] and the amount of shift will be determined by the map function. To reduce data in the stream, the map function will only emit the type and amount of shift and the first row or column instead of combining it into a OutputPlane. Without using window functions, the output of the map function will be processed by another map function, that constructs a full plane out of the shift type and amount and the first row or column. This full plane is then added to the final plane of the map class and emitted for every new data point.

[2]Whether the row or the column contains all values and therefore needs to be shifted to build the other values

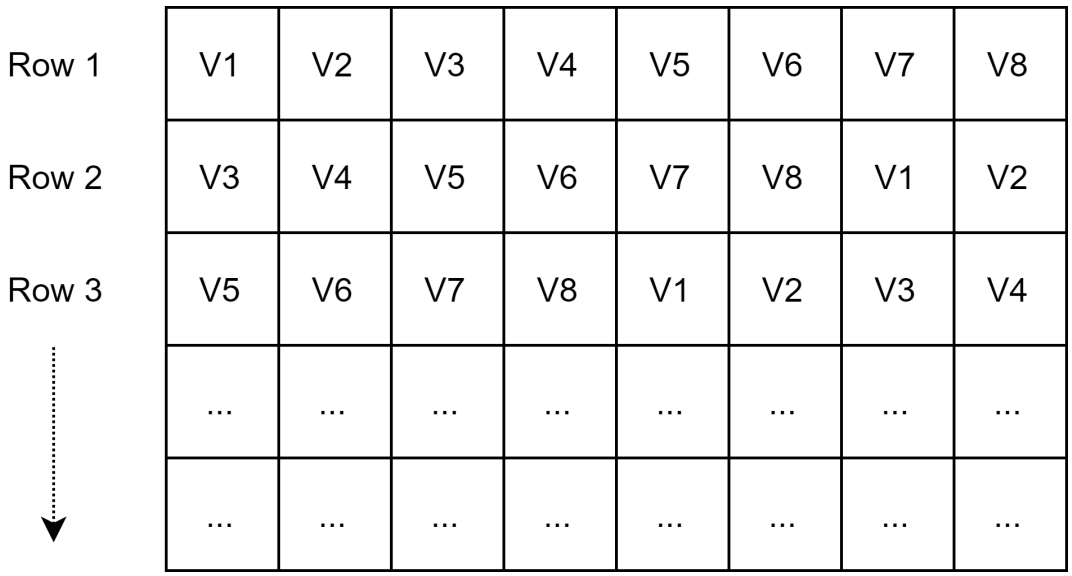


Figure 3.2.: Example of Shift of the Rows

The first map function, that provides shift and first row or column can operate in parallel, since this determination is independent. However, the combination of all the output of the first map function can not be done in parallel, since the OutputPlan needs to include the values of every point. If this process would be parallelized, multiple instances of an OutputPlane would exist and the data points would not flow into one OutputPlane containing all the data.

3.2.4. Discrete Fourier transform using Window Function

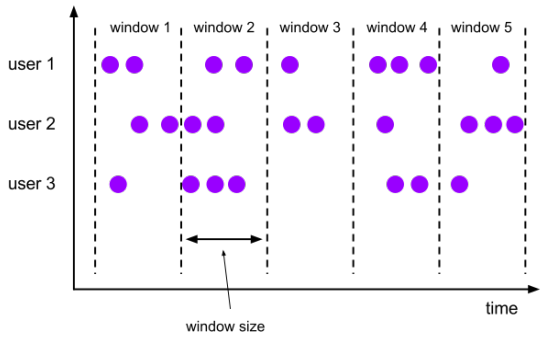
With the usage of a window function, the process of calculating the discrete Fourier transform can be optimized. The first map function described in Section 3.2.3 will be used again to determine shift properties and the first row or column. The data stream will then be be keyed. Keying is the action of partitioning a stream by a given key. Streams with the same key will then be processed by the same instance function. We will key the stream by its shift type and its shift value. This is done, because the first row or column of data point with the same shift properties can be added to each other and then the plan can be produced out of the summed up row or column.

Windows

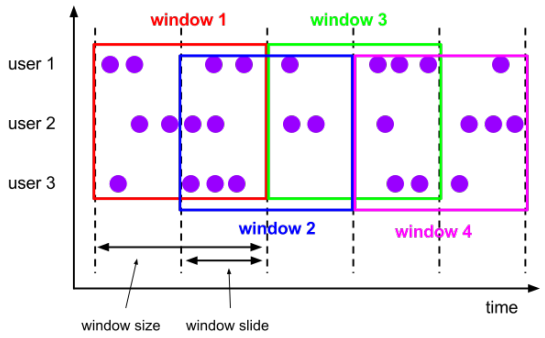
To still have a near-real-time-processing, the usage of a time window functions is advised. A window in general splits the stream in finite buckets of data. A time window will do this based on different implementations in combination with time. Apache Flink supports three different types of time windows.

- *The tumbling window*, which partitions the data into non overlapping windows of a given time t. A data point will be in exactly one window.

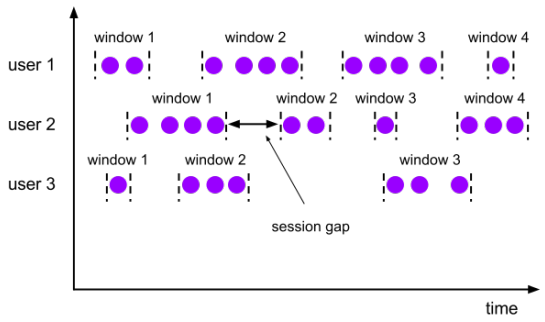
- *The sliding window* will make windows of a given time $t1$ every time $t2$. This allows sliding windows to overlap.
- *The session window* is a flexible length window, that will close once the data stream has been producing any new data for a given time t.



(a) Tumbling window



(b) Sliding window



(c) Session window

Figure 3.3.: Different window implementations in Apache Flink [1]

Figure 3.3 illustrates the different time windows.

Other types of windows, such as count windows[3] would not fulfil the requirement of a close-to-real-time processing of records. The time window is then used in combination with a reduce function. This function reduces the input of the window to a single output once the window is closed. The reduce function is then used to sum up the values of the first row or column. At the closing of the window, the summed up array will be emitted in a tuple with its shift properties. The window providing the least time between the input in the window and the windows closing while not processing multiple times is the tumbling window. A sliding window would either do the same as the tumbling window or process a record multiple times. A session window might stay busy to long and keep the input values to long in the reduce function until releasing the summed up values and therefore might even exclude a often occurring shift from the result, since the window may never close. Therefore a tumbling window has to be chosen[1].

After the tuples have been emitted, they can be added to the result plane with the same map function described and used in Section 3.2.3. In contrast to the previous implementations, the OutputPlane will no longer be evaluated for every datapoint but for every keyed stream created by keying the stream. This keyed streams will include multiple datapoints with the same key that were received during the in the tumbling window defined time.

3.2.5. Fast Fourier transform

For the fast Fourier transform a two dimensional array of complex numbers, using our class $Complex.java$, is initialized for every single input tuple. The complex number at the point $X = tuple.f0$ and $Y = tuple.f1$ is then set to the value of $tuple.f2$. Similar to the normal two dimensional dft-algorithm (see Algorithm 2) the two dimensional fft algorithm is implemented by transforming every row of the plane with fft and then transforming every column of the plane. In Listing 3.6 the map function doing exactly this fast Fourier transform is described.

The implementation of the *dofft(x)* function will be described further in 4.5. The class variable $myplane$ is used to add up all the values of the fast Fourier transform of every datapoint processed. The type of the output of this map function is in the Section 3.2.1 described OutputPlane. This plane will be emitted every time a new datapoint has been processed and its result has been added to the result plane.

[3] A count window takes n input data points and closes after these n data points have been taken. The next n data point will be processed by the next count window. This can be used to control the size of batches processed.

Listing 3.6: FFT Map Function

```
public OutputPlane map(Tuple3<Integer, Integer,
                    Double> inTuple3 ) {
        Complex[][] inp = new Complex[row][row];
        Complex[] x = new Complex[row];
            for (int i = 0; i < row; i++) {
                for(int j=0; j<row;j++) {
                    inp[i][j] = new Complex(0, 0);;
                }
            }
            inp[inTuple3.f0][inTuple3.f1] =
            new Complex(inTuple3.f2,0);
            for (int i = 0; i < row; i++) {
                for (int j = 0; j < row; j++) {
                    x[j] = inp[j][i];
                }
                dofft(x);
                for (int k = 0; k < row; k++) {
                    inp[k][i] = x[k];
                }
            }
            for (int i = 0; i < row; i++) {
                for (int j = 0; j < row; j++) {
                    x[j] = inp[i][j];
                }
                dofft(x);
                for (int k = 0; k < row; k++) {
                    inp[i][k] = x[k];
                }
            }
        for (int i = 0; i < row; i++) {
            for(int j=0; j < row;j++) {
                myplane.addValue(i, j, inp[i][j]);
            }
        }
        return myplane;
    }
}
```

3.3. Complete Pipeline

The before mentioned functions and techniques used will result in a finished stream pipeline. For an example data input $2, 3 = 2.00$ with a grid of size 4, the full streaming pipeline will process the data point as followed.

The datapoint will be read from the file and a string with the value $2, 3 = 2.00$ will be produced. To control the data flow, a count window is used. This window will create data batches of the size of the window. This string will then be converted in a Tuple3<Integer, Integer, Double>. This conversion in done by the splitter class, that implements a map function. After the values are represented in this tuple, the different implementations of the Fourier transform will be applied to the data. In the experiment setup, there are five different ways the Fourier transform is computed. These are defined as following:

- *DFT with Rotation:* With this implementation, the shift type and amount will be calculated. For the example point the *shift type* will be *row shift* and the *amount* will be *2*. The complex vector that makes up the first row is $[2.0, 2.0i, -2.0, -2.0i]$. Based on the shift properties the full grid will be calculated. The resulting grid will be show in Table 3.1. This values will then be added to the OutputPlane.

2.0	2.0i	-2.0	-2.0i
-2.0	-2.0i	2.0	2.0i
2.0	2.0i	-2.0	-2.0i
-2.0	-2.0i	2.0	2.0i

Table 3.1.: Result of Fourier transform of Point $V(2, 3) = 2.00$

- *DFT with Rotation and Window Functions:* In this implementation, the shift properties and the first vector are computed in the same way as mentioned in *DFT with Roatation*. After these computations have been made, the stream will be keyed by the shift amount and shift type. Each stream with a different key will then be processed by a time window of five seconds. In the time window, all the first row vectors will be summed up.

  Consider the Input of our example occurs twice in the time window. The two points will have the same shift properties and therefore be processed by the same window function. This results in a vector with the values $[4.0, 4.0i, -4.0, -4.0i]$. The summed up vector together with the shift properties will then be emitted. Out off this, the plane will be constructed and added up.

- *DFT with Rotation and Twiddle Factor precomputation:* In this implementation, the twiddle factors will be pre computed. We will pre compute W_N^0 to W_N^{N-1}. We could even reduce this to W_N^0 to $W_N^{N/2}$. This would make the lookup of the twiddle factor more complicated, because the lookup function then would need to determine if the position is smaller or larger than $N/2$. Since the computation is only made once, but the look ups are done multiple times, the simplicity of the lookup function is preferred.

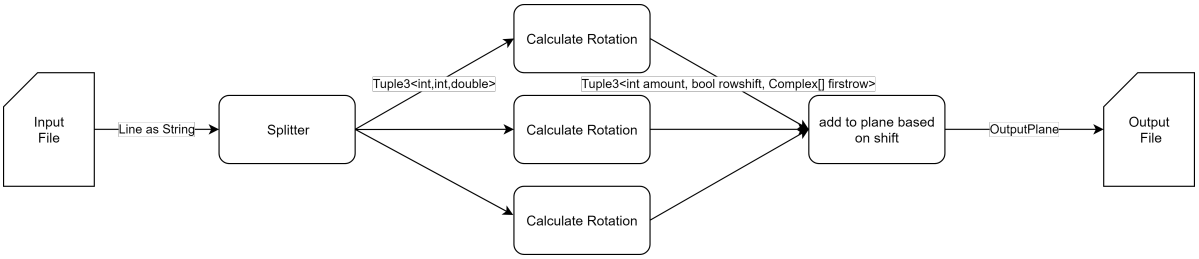
For the example points, the index of the twiddle factors would be $[0, 3, 2, 1]$ the twiddle factors would be a s described in Table 3.2. The Twiddle factors at the index are looked up and then multiplied with the value $V(u, v) = 2.00$. The vector for the first row will then be emitted and processed in the same way as in the before mentioned implementations.

W_N^x	rel	img
$x = 0$	1.0	0.0i
$x = 1$	0.0	-1.0i
$x = 2$	-1.0	0.0i
$x = 3$	0.0	1.0i

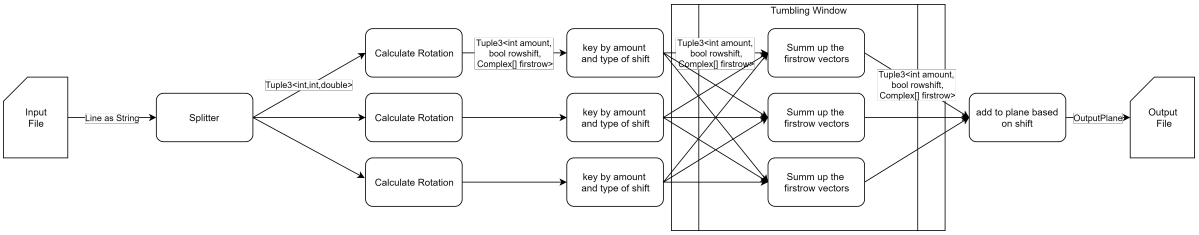
Table 3.2.: Twiddle factors for grid size $N = 4$

- *DFT with Rotation, Twiddle Factor precomputation and use of Window Functions:* As the name implies, this implementations combines the twiddle pre computation with the window function.
- *Fast Fourier transform:* For FFT the in Listing 3.6 described map function will be used. This measurement will be used as base line reading.

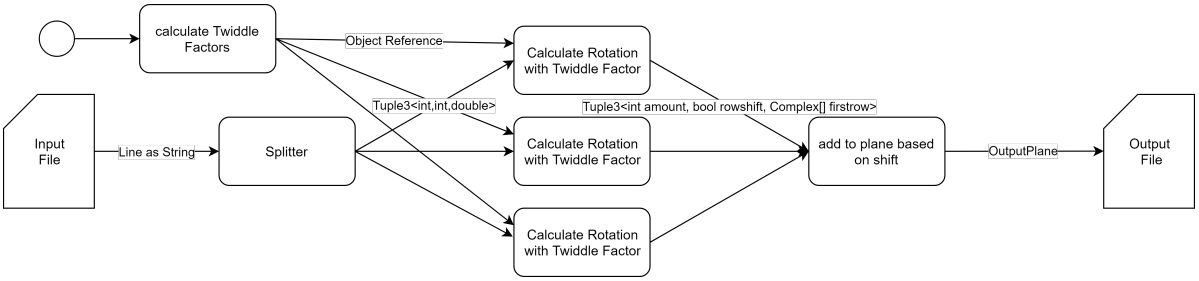
After the Fourier transform has been computed in one of the described ways, the resulting OutputPlane will be emitted and saved to the Output File. The graphic representation of the different implementations is provided in Figure 3.4.



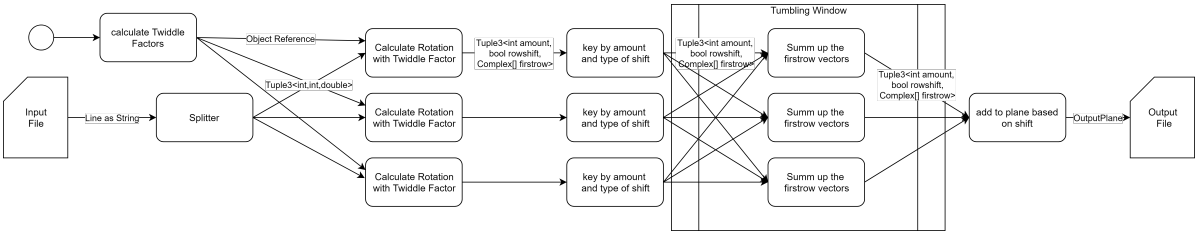
(a) DFT with Rotation



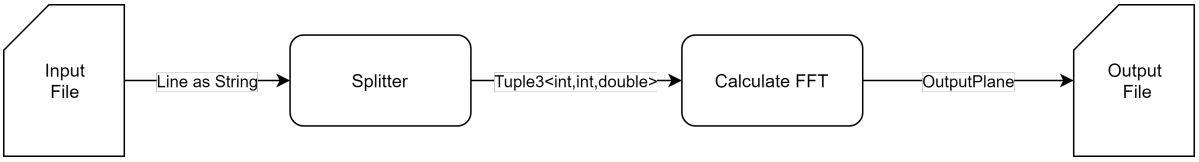
(b) DFT with Rotation and Window Functions



(c) DFT with Rotation and Twiddle Factor pre-computation



(d) DFT with Rotation and Twiddle Factor precomputation and use of Window Functions



(e) Fast Fourier transform

Figure 3.4.: Graphic representation of different stream pipelines

4. Fourier Transform implementation

In this Section the different implementations of the Fourier transform will be discussed and mathematical proof of correctness will be done. Especially the implementation wit the rotation of the first row or column will be described. For these implementations, the grid has to be same length as with and the length needs to be a power of 2. The discussed algorithm is then applied to our standard example.

Example

To illustrate the different algorithms discussed, we will use a set of example points with their value in a grid of size 4 to show how the Fourier transform is calculated.

Point Number	x-coordinate	y-coordinate	value
1	1	1	4
2	2	2	1
3	3	3	2

Table 4.1.: Example points and its values

4.1. Single Point DFT

Since a single point implementation of the discrete Fourier transform is discussed, the general formula see (2.8) of the 2D discrete Fourier transform can be simplified. The normal discrete Fourier transform at one point (u, v) would consist of a sum of all input values at point (m, n) multiplied with W_M^{um} and W_N^{vn}. Therefor the Value at point (u, v) would be a sum of M times N complex values, where M and N correspond to the grid size.

Since single point DFT is discussed, only the value at our single input point (x, y) is not zero. Therefore the output of the Fourier transform is only dependent on this one input point, since all the multiplications with the input value zero would result in zero. Therefor every point (u, v) of the transformed matrix can be described by a single multiplication of the input value at point (x, y) with its twiddle Factors W_M^{ux} and W_N^{vy} as described in (4.1).

$$F(u, v) = v(x, y) * W_M^{ux} * W_N^{vy} \tag{4.1}$$

So the two dimensional discrete Fourier transform can easily be implemented using two for loops to loop over the whole plane. For the Nth root of unity $e^{-j2\pi/N}$ is used, which can be re-written as $\cos(-2\pi/N) + j * \sin(-2\pi/N)$. The terms $-2\pi * u * x/M$ and $-2\pi * v * y/N$

are then used to describe the angle, of which the cosine and sine will describe our complex twiddle factor. This will result in the implementation listed in Listing 4.1.

Listing 4.1: Map function for DFT

```
public OutPutPlane map(Tuple3<Integer,
            Integer, Double> inTuple3) {
    double inverse1 = 1.0 / myplane.getRowSize();
    double inverse2 = 1.0 / myplane.getColumnSize();
    for (int x = 0; x < myplane.getRowSize(); x++)
        for (int y = 0; y < myplane.getColumnSize(); y++) {
            double angle1 =
            -2 * Math.PI * x * inTuple3.f0 * inverse1;

            double angle2 =
            -2 * Math.PI * y * inTuple3.f1 * inverse2;

            Complex W1 =
            new Complex(Math.cos(angle1), Math.sin(angle1));

            Complex W2 =
            new Complex(Math.cos(angle2), Math.sin(angle2));

            Complex result = W1.times(W2).times(inTuple3.f2);
            myplane.addValue(x, y, result);
        }
    return myplane;
}
```

Example

For the example point number 1 from our Table 4.1 we would make the following computations to calculate the discrete Fourier transform. First the inverse will be calculated which is equal to $1/4$ for both values, since we are using a grid with same length as width. We will first calculate the first column of the output, since the y-coordinate is in the inner loop. To calculate the final value at the point we will illustrate the calculation of the complex numbers angles, the complex numbers and the resulting number in Table 4.2a.

When then calculating all numbers for x and y from 0 to 3 the resulting grid will be as shown in Table 4.2b. When now computing the Fourier transform for the second point, the resulting grid after the transform would be as shown in Table 4.2c. When added together the Fourier transform of the grid containing point 1 and 2 would be as shown in Table 4.2d.

x	y	Angel1	Angel2	Complex $w1$	Complex $w2$	Result ($w1 * w2 * value$)
0	0	0	0	1	1	4
0	1	0	$-1/2\pi$	1	-i	-4i
0	2	0	$-\pi$	1	-1	-4
0	3	0	$-3/2\pi$	1	i	4i
1	0	$-1/2\pi$	0	-i	1	4i
1	1	$-1/2\pi$	$-1/2\pi$	-i	-i	4
1	2	$-1/2\pi$	$-\pi$	-i	-1	-4i
1	3	$-1/2\pi$	$-3/2\pi$	-i	i	-4

(a) computations for the first two columns of DFT

4	-4i	-4	4i
-4i	-4	4i	4
-4	4i	4	-4i
4i	4	-4i	-4

(b) Result of DFT at point (1,1) with value 4

1	-1	1	-1
-1	1	-1	1
1	-1	1	-1
-1	1	-1	1

(c) Result of DFT at point (2,2) with value 1

5	-1-4i	-3	-1+4i
-1-4i	-3	-1+4i	5
-3	-1+4i	5	-1-4i
-1+4i	5	-1-4i	-3

(d) Result of DFT of the points 1 and 2

Table 4.2.: Computation for DFT of Point 1 and 2 of discussed Example

4.2. Shift Implementation

As stated in Section 4.1, the Fourier transform for a point can be described by a single function without any summation. Since the twiddle factor is periodic, any values X for W_N^X bigger than N will result in the same as X modulo N. Since there are only N different values of the twiddle factor we will refer to a specific value as the $X - th$ twiddle factor = W_N^X. As stated in the definition of the twiddle factor, this value X is in our case defined by ux and vy. Since $W_M^{ux} * W_N^{vy}$ have the same base, we can rewrite the exponentiation as an addition of the exponents with the same base. For the Fourier transform of a input value $v(u, v)$ the transform would then be defined as in Equation (4.2).

$$F(u, v) = V(x, y)e^{-j2\pi \frac{ux}{M} + \frac{vy}{N}} \tag{4.2}$$

When only looking at the first row, the twiddle factor will be determined by the x-coordinate of the input value and the x-coordinate of the cell of the output, since the y-coordinate of the

first row is zero. Therefor the twiddle factor will be a multiple of the x-coordinate of the input for all values of the first row ux. Since we are only interested in values from 0 to $N-1$ we can state, that the twiddle factor of the first row can be described by

$$W[u] = W_N^{u*x \bmod N} \quad for\ u\ from\ 0\ to\ N-1. \tag{4.3}$$

For any odd values of x, the result of this equation will contain all the values from 0 to $N-1$.

To proof this, we have to look at the greatest common divisor of the value x and N. Since N is a power of 2^b, the only prime factors N consists of will be b times the prime factor 2. Every odd number x is per definition not divisible by 2. Therefore its prime factors will never contain 2. This leads to the fact, that the lowest common multiple is in fact the multiplication of the number x and N.

Therefore $u * x\ mod\ N$ will only be zero if $u * x$ is either 0 or the lowest common multiple or a multiple of the lowest common multiple. $u * x$ would be the lowest common multiple once u is equal to N. However, this will never happen, since as stated in Equation (4.3), u is only in the range of 0 to $N-1$. The modulo of $u * x$ by N will be $u * x$ until $u * x$ is greater than N.

Since x is odd and does not contain any common prime factor with N, at the point, where $u * x$ is the first time bigger than N it will be bigger by an number r that is in the range of 1 and $x-1$ because the previous number, that was smaller than N has been increased by x. The general range of r is 0 to $x-1$.

The first case, where $u * x\ mod\ N$ yields 0 is when u is 0. The next point will be at the the lowest common multiple of x and N. Modulo can be re written as $u * x - k * N$, where k should be the biggest possible number that will yield a non negative results. The term $u * x$ will range from 0 to $N * x - x$. Therefore k will range from 0 to $x-1$. Since k ranges from 0 to $x-1$, the number r defined as the rest once $u * x$ is bigger than N, has to progress to a point where the rest would be zero again. This would happen when $u = N$. Therefore for every different k, we will have a different number r. Since k covers every number from 0 to $x-1$, r will also cover every number from 0 to $x-1$.

Until $u * x$ is bigger than N, every multiple of x will be a result of $u * x\ mod\ N$. Once it gets bigger than N, we will start at the point r and from there on add all multiples of x. Since r is every number from 0 to x, we therefore will cover every number from 0 to $N-1$.

When looking at an even value of x, it will depend on how many time the number x contains the prime factor 2. The more this prime factor 2 is contained, the more the lowest common multiple is divided by 2. The lowest common multiple defines how many different rest r will occur. This will range from $\frac{x}{2^1}$ to $\frac{x}{N/2}$. Therefore the set of numbers that build the rest r once $u * x$ exceeds N still ranges from 0 to $x-1$ but will only be every $2^n th$ Number, where n is how many times the number x has 2 as a prime factor. For example the number 10 would have the values $0, 2, 4, 6, 8$ for r, since it only contains the prime factor 2 once. For 12 it would be $0, 4, 8$.

Therefor every multiple of 2^n from 0 to $N-1$ will be in the result set of $u * x\ mod\ N$, where u from 0 to $N-1$ and n = amount of prime factor 2 in number x.

The same is true for the first column, which is only defined by vy. Therefore the first row

and the first column will either contain the same values or either one will be a subset of the other one.

When looking at the second row, the index of the twiddle factor will still be defined by ux, but it is now also dependent on $1 * v$. So the index will be one y bigger than in the first row. But the increase of the twiddle factor for the next value in the same row will still be the same as in the first row. For the third row, it will be two y bigger and so on. The same is true for all columns. There are now three different cases that can occur.

- *The first row and column can both contain all numbers.* If this is the case, all rows or columns can be expressed by a shift of the first one, since it will contain all index values.
- *One first row or column contains all numbers, the other one not.* If this is the case, the following rows or column will be expressed by the first row or column containing all values. The increase of the index does not matter since it will contain all index values.
- *Both rows are only a subset of* $M = \{a|0 \leq a \geq N - 1\}$. In this case, the first row or column with the greater set of distinct numbers will be used to express the following ones. If both have the same amount it does not matter which one is taken. Otherwise the row or column with the smaller set will now determine how much the index of the second row is increased. This is exactly the value x or y, depending on whether the row or the column has the smaller set of numbers. As described, the size of the set of numbers contained is related to how many times 2 is a prime factor of x or y. Therefore an addition of the increase defined by x or y will always be made with a number that is at least dividable with the same power of two or higher as the other number, since its prime factorization will contain at least the same amount of 2 as its prime factor. Thus the index of the twiddle factor of point (0,0) with the addition of the value x or y to get the index at point (0,1) or (1,0) will result in a number that is still contained in the bigger set of numbers that is represented by the row or column. Therefore, it can still be used to describe all the values of the grid.

To determine the amount and type of shift the following Equation (4.4) has to be solved.

$$x * i \; mod \; N = y \quad or \quad y * i \; mod \; N = x \quad \text{Where i is the amount of shift} \tag{4.4}$$

The Algorithm 3 does compute exactly this to determine the shift

Assuming, that the row contains all values, the values of the output of the Fourier transform can then be expressed by the following equation (4.5). If the first column contains all the values, x and y have to be swapped.

$$F(x, y) = \text{Firstrow}[(\text{shiftamount } * x + y) \bmod N] \tag{4.5}$$

Firstrow refers to the array of complex numbers of the first row. With the use of Firstrow$[x]$, the xth element of the array will be accessed.

Algorithm 3 determine shift

```
for i = 0 to N do
  if (x*i) mod N == y then
    columshift ← true
    Shiftamount ← i
  end if
  if (y*i) mod N == x then
    columshift ← false
    Shiftamount ← i
  end if
end for
```

Example

For the example point 1 described in Table 4.1 we will first determine what type and amount of shift occurs. For this we have to evaluate the terms $(x*i)$ mod $N == y$ and $(y*i)$ mod $N == x$ for i from 0 to $N-1$. This is shown in Table 4.3a. Since both row and column shift are possible it was decided, that in this case the column shift will be preferred. Therefore we will have a column shift of 1. So we will compute the values of the first row as described in Table 4.2a. We can then calculate the index of the first column that represents any point in the grid as stated in (4.5). This will result in the values shown in Table 4.3b.When now constructing the plane out of the values of the first column, we will have the same resulting grid as shown in 4.2b.

For the second point $(2, 2) = 1$ the shift type will be column shift and also by an amount of 1. Therefore the indices for the point in the grid will be the same as ahown in 4.3b. After calculating the values of the first column, we can also construct the full plane, which will result in a representation excatly the same as shown in Table 4.2c. Therefore the result of the addition will also be the same.

4.3. Twiddle Factor precomputation

As described in Section 4.2, the twiddle factor of the first row or column can be determined by only the index of the field of the row and the coordinates of the input values. Therefore, instead of calculating the value of the twiddle factor for every point an then multiply it with the value at the input point, all values of the twiddle factor can be precomputed and then just looked up.

The twiddle factor for the first row is defined as W_N^{x*u}. So for the output point $(u, 0)$ we will look up the twiddle factor at index $x*u \; mod \; N$. To precompute the twiddle factor, a class can be used. All values will be calculated as soon as an object of the class is instantiated.

Listing 4.2: TwiddleFactors class

```
public class TwiddleFactors implements Serializable {
    private Complex [] myVector ; // to store the values
```

i	$(x * i)$ mod $N == y$	$(y * i)$ mod $N == x$
0	0 == 1 = false	0 == 1 = false
1	1 == 1 = true	1 == 1 = true
2	2 == 1 = false	2 == 1 = false
3	3 == 1 = false	3 == 1 = false

(a) Shift determination for point (1,1)

x	y	Index ((shiftamount $* x + y$) mod N)
1	0	1
1	1	2
1	2	3
1	3	0
2	0	2
2	1	3
2	2	0
2	3	1

(b) Index of first column for following column for point (1,1)

Table 4.3.: Computations made to determine shift and construct plane

```
    public TwiddleFactors(int size) {
        myVector = new Complex[size];
        double inverse1 = 1.0 / size;
        for (int i = 0; i < size; i++) {
            double angle1 = (-2.0 * Math.PI * i  * inverse1);

            Complex W1 =
            new Complex(Math.cos(angle1), Math.sin(angle1));

            myVector[i] = W1;
        }
    }
    public Complex getComplex (int index){
        return myVector[index];
    }
}
```

The value of a point in the first row $(u, 0)$ is defined as shown in Equation 4.6.

$$F(u,0) = v(x,y) * W_N^{x*u \ mod \ N}. \tag{4.6}$$

Therefore the values of the first row of the output of the Fourier transform can then be calculated without the use of trigonometric functions and only one complex factor in the multiplication, since we are only considering real input values.The complex multiplication that is

defined as

$$\begin{aligned} z_1 = (a + bi), \;\; z_2 = (c + di) \\ z_1 * z_2 = (ac - bd) + (ad + bc)i. \end{aligned} \tag{4.7}$$

can be re written as

$$\begin{aligned} z_1 = (a + bi), \;\; z_2 = (c + 0i) \\ z_1 * z_2 = (ac) + (bc)i. \end{aligned} \tag{4.8}$$

where z_2 is a real number. This reduces the number of computations required to calculate the Fourier transform.

Example

For our example the twiddle factors will be as shown in Table 4.4a. The values of the first row can then be calculated using the formula shown in Equation (4.6). For the first point this will result in the values shown in Table 4.4b. For the second point the values are as shown in Table 4.4c

Index	Complex W
0	1
1	-i
2	-1
3	i

(a) Twiddle factors for N = 4

Index	Twiddle Factor	Resulting Value
0	$W_N^0 = 1$	4
1	$W_N^1 = -i$	-4i
2	$W_N^2 = -1$	-4
3	$W_N^3 = i$	4i

(b) Values for the first row using twiddle factor precomputation for point 1

Index	Twiddle Factor	Resulting Value
0	$W_N^0 = 1$	1
1	$W_N^2 = -1$	-1
2	$W_N^0 = 1$	1
3	$W_N^2 = -1$	-1

(c) Values for the first row using twiddle factor precomputation for point 2

Table 4.4.: Computations made for twiddle factor precomputation

4.4. Window Function

Since the goal of our stream processing use case is to continuously compute the Fourier transform, every input is separately transformed and all the transformed output will be added up.

As described in the shift implementation, the Fourier transform can be described by the shift properties and the first row or column. Instead of constructing the full plane out of this information an the adding together all planes to get the final result, we can key the stream by the shift properties. This is done, because for data points with the same shift properties, the plane will be constructed in the same way as shown in (4.5).

After the construction of the plane, the whole planes would be added to the final output plane. This would require $N * N$ complex additions. Instead of adding the output plane of every single data point individually, a time window is used to sum up the first vector of streams with the same key. This additions will only take N complex additions for every data point that is an input to the time window and then the already summed up plane can be added to the final plane, which will still require $N*N$ complex additions. The additions in the time window will be done by a reduce function. The implementation of keyed streams and window functions is very straight forward in Apache Flink.

The $.keyBy()$ function can easily be called on any data stream. As argument it will take the index of the element that should made up the key or its field name. After the data stream has been keyed, a window can be applied with the use of the $window()$ function that takes a window as function argument such as $TumblingProcessingTimeWindows.of(Time.seconds(5))$, which will create a five second tumbling time window. The windowed stream can then be reduced by calling the $.reduce()$ function that takes a class that implement the $reduce$ method. In this case the reduce function is defined as listed in Listing 4.3.

Listing 4.3: Reduce Function for addition of first row or column

```
public Tuple3<Complex[], Boolean, Integer> reduce
        (Tuple3<Complex[], Boolean, Integer> v1,
        Tuple3<Complex[], Boolean, Integer> v2){
    for(int i = 0; i < row; i++){
        v1.f0[i] = v1.f0[i].plus(v2.f0[i]);
    }
    return new Tuple3<>(v1.f0, v1.f1, v1.f2);
}
```

Example

Assuming the three points shown in Table 4.1 are processed in the same window, depending on their keys, they might be processed by the same instance of the reduce function. As we already evaluated in the shift example, the first two point both have a column shift of one. Therefore they have the same key and will be processed by the same reduce function. This function will sum up the values of the first column. In our example this would result in a first column as shown in Table 4.5a. Because the shift has an amount of 1, the plane expressed by the first column will be the same as shown in Table 4.2d.

Now once we also look at the third point in our window, we realize, that its shift also is a column shift of the amount of 1. Therefor the third point will also be added up in our reduce function, which will result in an output vector of the reduce function as shown in Table 4.5b. This will then result in an plane that describes the Fourier transform of a grid containing all

the input points such as shown in Table 4.5c. The resulting grid constructed of shifting the reduced vector will then be as shown in 4.5d

Point 1	Point 2	Resulting column
4	1	5
-4i	-1	-1-4i
-4	1	-3
4i	-1	-1+4i

(a) Resulting column of reduce function for point 1 and 2

Point 1	Point 2	Point 3	Resulting column
4	1	2	7
-4i	-1	2i	-1-2i
-4	1	-2	-5
4i	-1	-2i	-1+2i

(b) Resulting column of reduce function of all three points

0	0	0	0
0	4	0	0
0	0	1	0
0	0	0	2

(c) The full input grid

7	-1-2i	-5	-1+2i
-1-2i	-5	-1+2i	7
-5	-1+2i	7	-1-2i
-1+2i	7	-1-2i	-5

(d) The result of the Fourier transform of all three points

Table 4.5.: Computations made with usage of window function

4.5. Fast Fourier transform

For the evaluation of the performance of the fast Fourier transform, a standard implementation published by Sedgewick and Wayne has been used[11]. This one dimensional implementation has then been used as shown in listing 3.6. The standard implementation is defined as in Algorithm 4.

Algorithm 4 dofft()[11]

```
n = x.length;
shift = 1 + Integer.numberOfLeadingZeros(n);
for int k = 0; k < n; k++ do
    int j = Integer.reverse(k) > > > shift;
    if j > k then
        Complex temp = x[j];
        x[j] = x[k];
        x[k] = temp;
    end if
end for
for int L = 2; L <= n; L = L+L do
    for int k = 0; k < L/2; k++ do
        double kth = −2 ∗ k ∗ Math.PI/L;
        Complex w = new Complex(Math.cos(kth), Math.sin(kth));
        for int j = 0; j < n/L; j++ do
            Complex tao = w.times(x[j*L + k + L/2]);
            x[j*L + k + L/2] = x[j*L + k].minus(tao);
            x[j*L + k] = x[j*L + k].plus(tao);
        end for
    end for
end for
```

5. Experimental Evaluation

5.1. Goal

The goal of the performance evaluation made, was to analyze and compare the performance of the new implemented discrete Fourier transform algorithm, which uses the shift computation and then the computation of the first row or column, with the traditional FFT implementation. Furthermore the performance behavior of the other enhancements proposed should also be analyzed. Another goal is to evaluate whether the parallelization of the shift determination grants an increase in performance or not. This may give some additional information about how much the additions of the whole plane are still the bottleneck of these computations. Therefore the performance of the algorithm will be evaluated on different grid sizes. When referring to a grid of size N a grid with N rows and N columns is meant. This means the amount of Points in the grid is N^2. The different implementations have been evaluated with a grid size of 2^N where N ranges from 5 to 12 resulting in grids from 32 to 4096. For all the different grid sizes, a separate input file with a total of $10'000$ values has been generated with the python script mentioned in Listing 3.1.

5.2. Experimental Setup

To evaluate the performance of the different implementations of Fourier transform in the stream processing environment, a constant platform for running the experiments had to be used. For our results, Apache Flink was run on a desktop computer with an Intel core i7 sixth generation quad core processor with a total of eight threads boosting up to 3.8 GHz processing speed. To not limit the performance by input/output operations, all files have been saved on an m.2 nvme SSD. The computer running Flink used two 16GB modules of DDR4 RAM adding up to a total capacity of 32 GB of RAM. The computer is running Windows 10 as operating system.

Since the running time of the larger grids ($n = 4096$) tend to bee at around the 30 minute mark, we settled on running the configuration we want to evaluate ten times and then use the mean and median of these measurements to compare the different implementations. All the measurements are made by using the $System.nanoTime()$ function in java to get the current system time in nanoseconds. This function is used at the start of the main method to get the start time and is then taken again after the stream environment has been executed. By calculating the difference and multiplying with 10^6 we will get the run time in milliseconds. All the listed run times will be in milliseconds if not explicitly noted different.

The experiments have then been run for the following different configurations:

- *FFT:* the FFT implementation described in 3.2.5 and 4.5 will build our base line reading to compare our new algorithm with the current standard implementation.
- *DFT with rotation:* This implementation will be evaluated with a variety of different parallelism levels of the shift determination algorithm ranging from one to four, since the computer running the experiments has four cores. The final additions can not be calculated in parallel manner since all input points have to add up to one final value.
- *DFT with rotation and twiddle factor precomputation:* Instead of computing the twiddle factors for every point, we will now just precompute the twiddle factors
- *DFT with rotation and window Functions:* The window function implementation will bee evaluated for different sizes of window ranging from 1 to 10 seconds, since we still want to be at the close-to-real-time execution of computations.
- *DFT with rotation, twiddle factor precomputation and use of window Functions:* As stated we will combine the window function with the twiddle factor precomputation and evaluate its performance.

5.3. Results of Experiments

In this Section the raw data of the experiments will be presented and discussed. All the raw data is included in the appendix A.

5.3.1. Linear Stream Pipelines

In this Section the FFT implementation, the rotation implementation, the rotation with twiddle precompute implementation and the window implementation, all of these in a serial configuration without any parallel computation steps, will be compared. The run times are visualized in Figure 5.1.

We can clearly see, the run time of the FFT implementation drastically increases with the grid size. If the computation time for one point in the FFT algorithm is compared to the same measurement of the rotation algorithm, we can clearly see, that the computation time starts at barely the same time for one record at a grid of size 32 but then drastically increases until the FFT algorithm needs 60 times more time to calculate than the FFT for one point at a grid size of 4096 as seen in Figure 5.2. When only comparing the implementation with rotation and its enhanced variants with twiddle factor precomputation and the window function, We can see that both of the enhancements made, grant an increase in performance. We can see that the twiddle factor precomputation only grants a small increase in performance in contrast to the window implementation, which grants a much bigger increase in speed. This is due to the fact, that the precomputation implementation only reduces the computation in the for-loop where the first row or column is computed. therefore the saving of computation is only for a computational step that is executed N times.

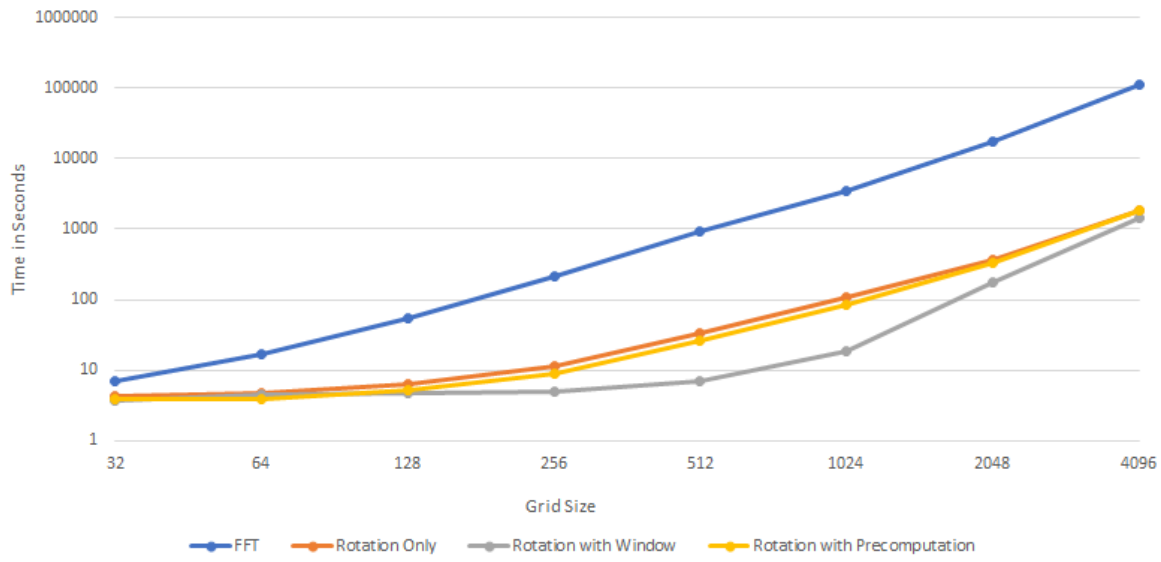


Figure 5.1.: Linear Run Times

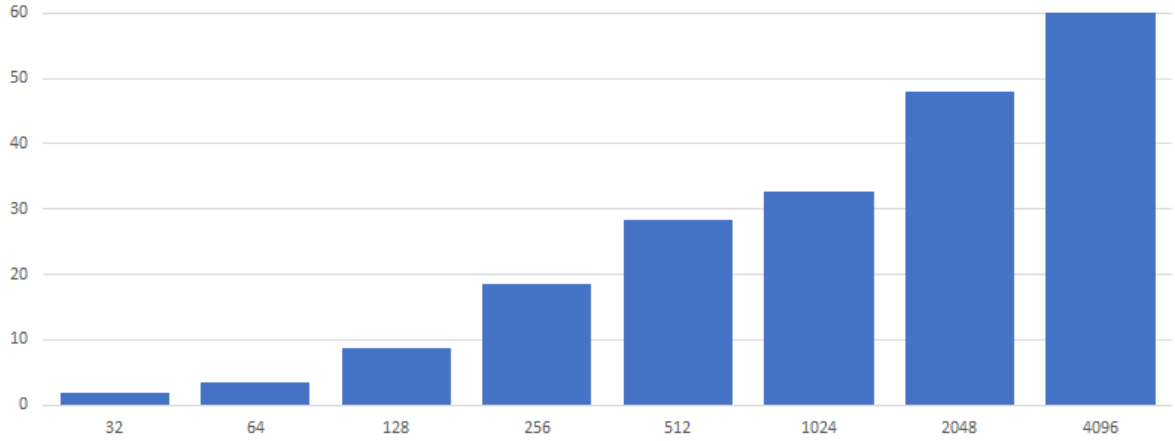


Figure 5.2.: Ratio of computation time for one single point for FFT and rotation implementation

The window function in contrast reduces the amount of times a whole plane is added to the OutputPlane. Therefore the additions that are carried out N^2 times can be reduced. How many of these additions can be reduced depends on how many data point with the same shift are in the time window. This effect will be evaluated in Section 5.3.3

5.3.2. Parallel Stream Pipelines

When comparing the normal rotation implementation, we can barely see in Figure 5.3, that the computations of the shift properties and the first vector tend to get more efficient once the operations are performed in a parallel manner. However, the performance gain for the

computation of one record for the grid of size 4096 is only in the millisecond range and therefore is pretty negligible. This indicates that the additions of the single Fourier transformed grids to the final OutputPlane still take the most computational power and therefore bottleneck the stream processing. Therefore a more efficient computation of the rotation will in either way be useless since these outputs can not be processed any faster.

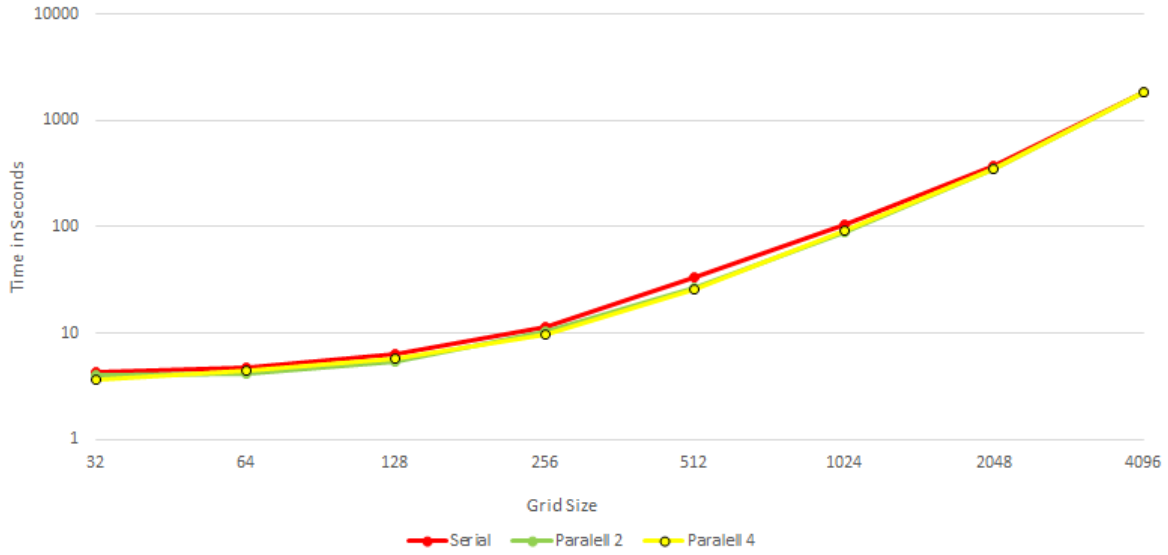


Figure 5.3.: Run Times for different parallelism levels of the rotation implementation

The parallel implementations of the rotation algorithm in combination with the twiddle factor precomputation even sees a decrease in performance as seen in Figure 5.4. This may occur because the parallel computation of the shift properties and the first vector by just looking up the twiddle factor may overload the map function that adds up the plane and the final OutputPlane. Further evaluation must have been done to be able toe exactly specify the reason of this performance decrease.

5.3.3. Window Size

The size of the time window will determine the amount of records processed by one window. This will also influence the probability of having a case where the shift properties of two data point are the same and out window function can be used to decrease the additions to add the transformed grid to the final OutputPlane. As we can see in Figure 5.5, the computation times for the smaller grids is nearly the same for all tree different sized time windows. This may be due to the fact that the total running time is within the five to ten seconds mark. Therefore the amount of data points per window are enough to have points with the same shift in one window.

When the grid size increases, we can see that the performance of the one second time window drastically decreases in comparison to the five second window. The five second window is also getting slower in comparison to the ten second window. This is due to the fact, that the

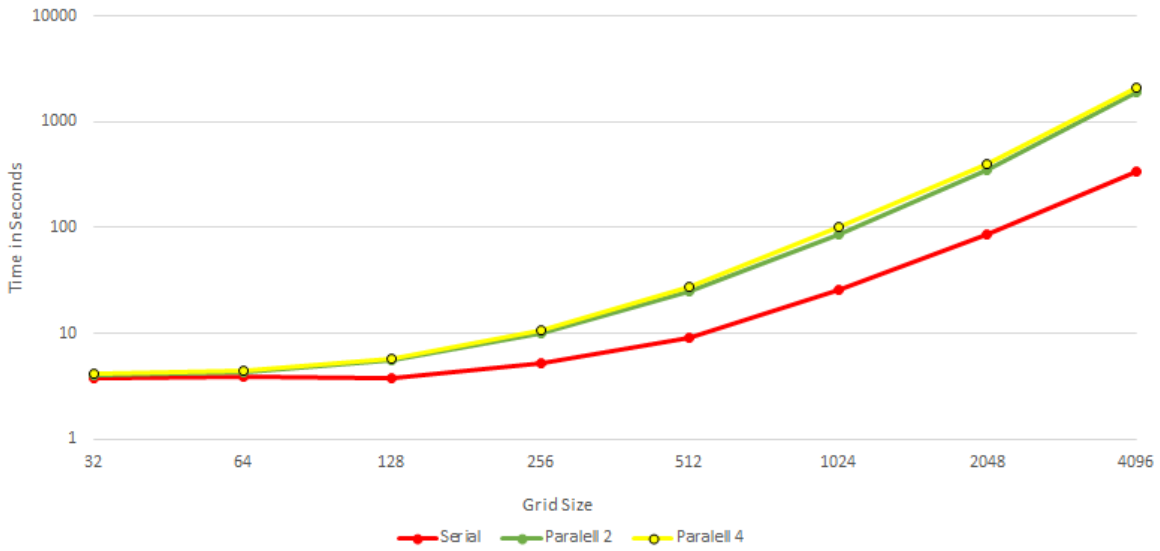


Figure 5.4.: Run Times for different parallelism levels of the Twiddle precomputation implementation

shift determination of one single data point will take more time and also the number of different possibilities of shift property combinations increase. Therefore a window with a bigger time slot will increase the chance to have data points that can be reduced.

However, choosing a larger window, for example with a time slice of 30 seconds, will destroy the idea of the close-to-real-time computation we are aiming for when using stream processing on the given data. So a trade-of between real time data and efficient processing must been made, once a certain size of grid is used to calculate the Fourier transform on. We settled on a five second time window since this unites the performance and the close-to-real-time processing in a way we want the performance to be.

5.4. Combination of findings

As evaluated in Section 5.3, the combination of the window function with a time window of size five seconds and the twiddle factor precomputation should yield the best performing implementation of a single point Fourier transform. The shift determination will be done in serial manner since a parallel computation will not affect the running time, since the additions after the transform are still the bottleneck. When combining these, the run time for the computation has decreased further as can bee seen in Figure 5.6.

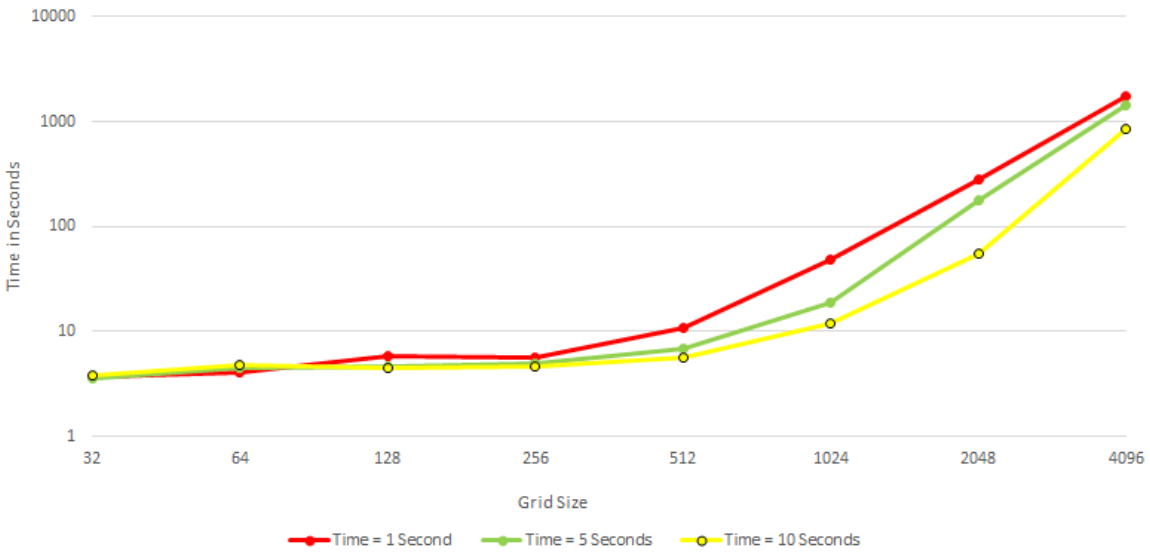


Figure 5.5.: Run Time for different window sizes

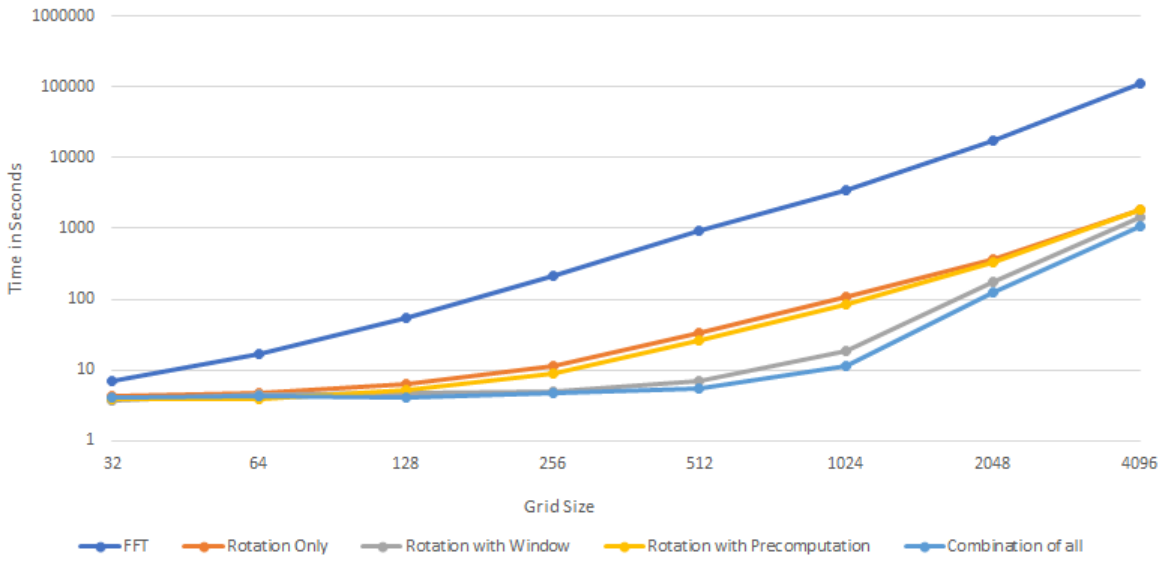


Figure 5.6.: Run Time of combined stream versus the other implementations

6. Conclusion

We showed that with the concern of having a grid with a size that is the power of 2, as it is with most of the FFT implementations, the computation of the Fourier transform for a single data point can be done in such a manner, that it is only necessary to compute the first row or column of the output grid and all other values can be represented by the first row or column. This implementation will still be bottle-necked by the complex additions that have to be done to sum up all the single point Fourier transforms to then yield the final result. We proposed the use of Window functions to decrease this bottleneck. However, until there is parallel no way to add up the single point Fourier transform, while still preserving a stream order, the final additions of the transforms to the OutputPlane are the bottleneck.

We showed with our artificially generated test data sets, that our base line implementation using only the rotation algorithm already is a huge performance upgrade compared to the normal FFT implementation. With the biggest grid used in our test, we can record a 60-times faster processing time in comparison to the standard FFT implementation. During the experimental evaluation, we found out, that the parallelization of the determination of the shift properties is only leading to a performance gain as long as the computations made use a certain amount of computational power. When we combined the determination of shift with the precomputation of the twiddle factors, the performance was even reduced when executing the computation of shift in parallel. The performance gain introduced by the usage of window function was the biggest gain of all enhancements done to the algorithm since it reduced the amount of times the whole grid needed to be added up to the final output grid. The combination off all our enhancements of the algorithm even lead to a result that is up to 120 times faster than the standard FFT implementation when comparing run times of the biggest grid.

The findings of our experiments are consistent with the theory our algorithm and its enhancement are base on. As predicted does the window function yield a bigger improvement in performance of our algorithm than the precomputation of the twiddle factors. A way to further increase the performance of this algorithm would be to find a way to aggregate the input of the windows to create less output data that has to be added separately.

Bibliography

[1] "Windows," Accessed: 05.04.2019. [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.7/dev/stream/operators/windows.html

[2] "Operators," Accessed: 05.04.2019. [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.7/dev/stream/operators/index.html

[3] C. A. T. N. Facility and E. NSW, "Askap computing," Mar 2019, Accessed: 30.04.2019. [Online]. Available: https://www.atnf.csiro.au/projects/askap/computing.html

[4] R. N. Bracewell and R. N. Bracewell, *The Fourier transform and its applications*. McGraw-Hill New York, 1986, vol. 31999.

[5] S. Roberts, "Signal Processing & Filter Design, Lecture Notes: Lecture 7 - The Discrete Fourier Transform," 2003, Accessed: 05.04.2019. [Online]. Available: http://www.robots.ox.ac.uk/~sjrob/Teaching/SP/l7.pdf

[6] 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*. ACM, 1966, pp. 563–578.

[7] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[8] 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.

[9] "Apache flink documentation," Accessed: 05.04.2019. [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.7/

[10] "Class tuple3," Apr 2019, Accessed: 07.04.2019. [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-stable/api/java/org/apache/flink/api/java/tuple/Tuple3.html

[11] R. Sedgewick and K. Wayne, "Inplacefft.java," Nov 2017, Accessed: 30.04.2019. [Online]. Available: https://introcs.cs.princeton.edu/java/97data/InplaceFFT.java.html

Appendices

A. Running Times

Grid Size	Mean Running Time	Median Running Time	Mean Time for one record
32	7034.23091	6977.18069	0.70342309
64	16459.0537	16318.4034	1.64590537
128	55153.9608	55299.6424	5.51539608
256	209247.925	208702.07	20.9247925
512	952755.145	952755.145	95.2755145
1024	3442480.214	3323940.188	344.2480214
2048	17815213.6	174665.277	1781.52136
4096	112408954.6	1126814.524	11240.89546

Table A.1.: FFT results

Grid Size	Mean Running Time	Median Running Time	Mean Time for one record
32	4334.8109	4340.10722	0.43348109
64	4821.30645	4761.48184	0.48213064
128	6344.19153	6277.7676	0.63441915
256	11369.1983	11400.87	1.13691983
512	33121.2038	33254.7626	3.31212038
1024	106002.559	105044.837	10.6002559
2048	371454.968	365470.651	37.1454968
4096	1875693.675	1790838.998	187.5693675

(a) Parallelism of 1

Grid Size	Mean Running Time	Median Running Time	Mean Time for one record
32	4085.97328	4048.45939	0.40859733
64	4144.76954	4033.77275	0.41447695
128	5426.98434	5446.6667	0.54269843
256	10352.6889	10122.6981	1.03526889
512	27048.8749	26810.08	2.70488749
1024	90399.7504	90324.0165	9.03997504
2048	353510.491	352237.221	35.3510491
4096	1861801.179	1862878.082	186.1801179

(b) Parallelism of 2

Grid Size	Mean Running Time	Median Running Time	Mean Time for one record
32	3707.93107	3732.46348	0.37079311
64	4479.06623	4450.05659	0.44790662
128	5698.08133	5602.07678	0.56980813
256	9786.97422	9621.34588	0.97869742
512	26048.0055	26019.7405	2.60480055
1024	92546.8154	93949.1715	9.25468154
2048	345590.425	348657.094	34.5590425
4096	1856966.512	1847328.082	185.6966512

(c) Parallelism of 4

Table A.2.: DFT with rotation results

Grid Size	Mean Running Time	Median Running Time	Mean Time for one record
32	3705.78683	3702.87597	0.37057868
64	4092.3037	3788.72801	0.40923037
128	5751.10938	5827.1325	0.57511094
256	5646.15491	5345.2206	0.56461549
512	10747.1106	11035.4118	1.07471106
1024	48233.9416	45236.4681	4.82339416
2048	284571.323	284786.323	28.4571323
4096	1760642.268	1669198.484	176.0642268

(a) Window of 1 Second

Grid Size	Mean Running Time	Median Running Time	Mean Time for one record
32	4325.11508	4266.47084	0.43251151
64	4428.50672	4384.1828	0.44285067
128	4624.1195	4620.38569	0.46241195
256	4975.5717	4964.14812	0.49755717
512	6837.92085	6268.83238	0.68379209
1024	18928.7245	22613.3962	1.89287245
2048	177283.32	186215.641	17.728332
4096	1433764.544	1417721.828	143.3764544

(b) Window of 5 Second

Grid Size	Mean Running Time	Median Running Time	Mean Time for one record
32	3745.31403	3693.59793	0.3745314
64	4853.6401	4778.14609	0.48536401
128	4447.03168	4083.60318	0.44470317
256	4637.02773	4558.86479	0.46370277
512	5605.10155	5451.12371	0.56051015
1024	11773.5315	9001.44105	1.17735315
2048	55431.5831	59191.5831	5.54315831
4096	841038.6781	792712.7601	84.10386781

(c) Window of 10 Second

Table A.3.: DFT with rotation and window functions results

Grid Size	Mean Running Time	Median Running Time	Mean Time for one record
32	3923.07624	3898.73394	0.39230762
64	3840.7794	3808.51495	0.38407794
128	5187.00331	5159.68501	0.51870033
256	9049.81903	9042.40482	0.9049819
512	26084.0635	26506.1302	2.60840635
1024	85447.5594	85413.6733	8.54475594
2048	337222.158	325700.696	33.7222158
4096	1851801.179	1852878.082	185.1801179

(a) Parallelism of 1

Grid Size	Mean Running Time	Median Running Time	Mean Time for one record
32	4060.84767	3994.89442	0.40608477
64	4285.1733	4247.87508	0.42851733
128	5523.90716	5444.49125	0.55239072
256	9958.11166	9780.59186	0.99581117
512	25386.6875	24082.0054	2.53866875
1024	85858.0007	85602.8528	8.58580007
2048	355143.812	350169.714	35.5143812
4096	1916437.376	1833924.316	191.6437376

(b) Parallelism of 2

Grid Size	Mean Running Time	Median Running Time	Mean Time for one record
32	4143.86973	4111.99913	0.41438697
64	4392.11484	4349.12792	0.43921148
128	5839.60354	5705.02722	0.58396035
256	10730.0365	10916.0444	1.07300365
512	27883.9713	27692.2457	2.78839713
1024	102765.458	97404.0039	10.2765458
2048	403657.764	419939.233	40.3657764
4096	2102232.158	2099898.825	210.2232158

(c) Parallelism of 4

Table A.4.: DFT with rotation and twiddle factor precomputation results

Grid Size	Mean Running Time	Median Running Time	Mean Time for one record
32	4174.85544	4153.64587	0.41748554
64	4248.66991	4201.59961	0.42486699
128	4185.89422	4141.91118	0.41858942
256	4625.36345	4591.51868	0.46253634
512	5481.485	5500.42667	0.5481485
1024	11499.6276	9002.29273	1.14996276
2048	123687.836	133629.645	12.3687836
4096	1101319.311	1208856.182	110.1319311

Table A.5.: DFT with rotation, twiddle factor precomputation and use of window functions results