**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

Jorge Francisco Teixeira Bastos da Mota

# High Performance Fourier Transforms on GPUs

February 2022

**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

Jorge Francisco Teixeira Bastos da Mota

# High Performance Fourier Transforms on GPUs

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
**Supervisor**
**Co-supervisor (if any)**

February 2022

## ABSTRACT

KEYWORDS FFT, GLSL, cuFFT, analysis, performance.

# RESUMO

Escrever aqui resumo (pt) ou importar respectivo ficheiro

PALAVRAS-CHAVE FFT, GLSL, cuFFT, análise, performance

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# 1

# INTRODUCTION

## 1.1 CONTEXTUALIZATION

The Fast Fourier Transforms have been present in our sorroundings for a long time, they're used extensively in digital signal processing and many other areas and they often need to be used in a realtime context, where the computations must be performed fast enough. Fast Fourier Transforms essentially are just optimized algorithms to compute the Discrete Fourier Transform of some data, data that might be sampled from a signal, an oscilating object or even an image, which is transformed into the frequency domain allowing any kind of processing for a relativelly low computational cost.

## 1.2 MOTIVATION

The continuous progress of the evolution of GPUs has increased the popularity of parallelizable algorithm implementations on this type of hardware. Notably the FFT algorithms family is constantly present in Computer Graphics, it's usual to find inlined implementations in shader code which offer reliable Fast Fourier Transforms **?**, but lack tuning of settings for a more optimized versions of these computations. On the other hand there's already out there libraries that provide efficient implementations of FFT on the GPU and CPU like cuFFT **?**, a library provided by NVIDIA exclusively for their GPU's implemented for CUDA, and FFTW **?**, a library dedicated to computations of FFT on the CPU.

Although this libraries can provide efficient transforms with specialized cases over a proper plan, in some applications its performance might be compromised for cases where, for example, the graphics pipeline needs to be sincronized with the computation of the Fourier Transform.

## 1.3 OBJECTIVES

The main objective of this dissertation is to provide efficient FFT alternatives in GLSL compared with dedicated tools for high performance of FFT computations like NVIDIA cuFFT library or FFTW, while analysing the intrinsic of a good Fast Fourier Transform implementation on the GPU. To accomplish the main objective there are two stages taken in consideration, *Analysis of CUDA and GLSL kernels* to be well settled in their differences and to have a reference for the second stage *Analysis of application specific implementations* which will cluster the study's main objective and where we'll use as case of study applications with implementation of the FFT in the field of Computer Graphics that require realtime performance.

With constant progression of the research needed for this project, some steps of the work plan were refactored to meet the needs. The two main stages of the objectives stay the same but there are some adjustments to the schedule dates and steps as shown in ??.

- **Research Fast Fourier Transform**
- **Study cuFFT**, understand internal optimizations and prepare specialized profiles.
- **Analysis of CUDA and GLSL kernels** for FFT raw computations.
- **Research of Application driven FFT**, specialized implementations on the context of the application.
- **Writing of pre-dissertation**
- **Writing of dissertation**

| | 2021 | | | 2022 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Nov | Dez | Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | Set |
| Research Fast Fourier Transform | | | | | | | | | | | |
| Study cuFFT | | | | | | | | | | | |
| Analysis of CUDA and GLSL kernels | | | | | | | | | | | |
| Research of Application driven FFT | | | | | | | | | | | |
| Writing of pre-dissertation | | | | | | | | | | | |
| Writing of dissertation | | | | | | | | | | | |

**Table 1:** Dissertation schedule

## 1.4 DOCUMENT ORGANIZATION

This dissertation is organized in 3 chapters. Firstly, the ?? exposes the main introduction to the subject of this dissertation with the respective background information and defines objectives including contextualization and this document organization section.

To give a state of the art overview of the theory and practice associated with Fourier Transforms, ?? covers most of basic understandings and algorithms needed for later chapters, this will only take simple approaches to each concept to give intuitive insight and empirical explanations without proving it formally.

# 2

# STATE OF THE ART

## 2.1 FOURIER TRANSFORM

### 2.1.1 *What is Fourier Transform*

The **Fourier Transform** is a mathematical method to transform the domain refered to as *time* of a function, to the *frequency* domain, intuitively the Inverse Fourier Transform is the corresponding method to reverse that process and reconstruct the original function from the one in *frequency* domain representation.

Although there are many forms, the Fourier Transform key definition can be described as:

$$X(f) = \int_{-\infty}^{+\infty} x(t)e^{-ift}dt \tag{1}$$

$$x(t) = \frac{1}{2\pi}\int_{-\infty}^{+\infty} X(f)e^{-ift}df \tag{2}$$

- $x(t), \forall t \in \mathbb{R} \rightarrow$ function in *time* domain representation with real $t$.
- $X(f), \forall f \in \mathbb{R} \rightarrow$ function in *frequency* domain representation with real $f$, also called the Fourier Transform of $x(t)$
- $i \rightarrow$ imaginary unit $i = \sqrt{-1}$

This formulation shows the usage of complex-valued domain since the imaginary unit $i$ doesn't represent a value in the set of real numbers, making the fourier transform range from real to complex values, one complex coefficient per frequency $X : \mathbb{R} \rightarrow \mathbb{C}$

If we take into account the Euler's formula, we can replace the Fourier Transform for an equivalent, fragmenting the euler constant for a sine and cosine pair.

$$e^{ix} = \cos x + i \sin x \tag{3}$$

$$X(f) = \int_{-\infty}^{+\infty} x(t)(\cos(-ft) + i\sin(-ft))dt \tag{4}$$

Hence, we can break the Fourier Transform apart into two formulas that give each coefficient of the sine and cosine components as functions without dealing with complex numbers.

$$\begin{aligned} X_a(f) &= \int_{-\infty}^{+\infty} x(t)\cos(ft)dt \\ X_b(f) &= \int_{-\infty}^{+\infty} x(t)\sin(ft)dt \end{aligned} \tag{5}$$

The above definition of the Fourier Integral **??** can only be valid if the integral exists for every value of the parameter $f$. This model of the fourier transform applied to infinite domain functions is called **Continous Fourier Transform** and its targeted to the calculation of the this transform directly to functions with only finite discontinuities in $x(t)$.

### 2.1.2 *Where it is used*

It's noticieable the presence of Fourier Transforms in a great variety of apparent unrelated fields of application, even the FFT is often called ubiquitous[1] due to its effective nature of solving a great hand of problems for the most intended complexity time. Some of the fields of application include Applied Mechanics, Signal Processing, Sonics and Acoustics, Biomedical Engineering, Instrumentation, Radar, Numerical Methods, Electromagnetics, Computer Graphics and more **?**.

One of the most well known cases of application is **Signal Analysis**, the Fourier Transform is probably the most important tool for analyzing signals, when representing a signal with amplitude as function of time, a signal can be translated to the frequency domain, a domain that consists of signals of sines and consines waves of varied frequencies, as demonstrated in **??**, but to calculate the coefficients of those waves we need to use the Fourier Transform.

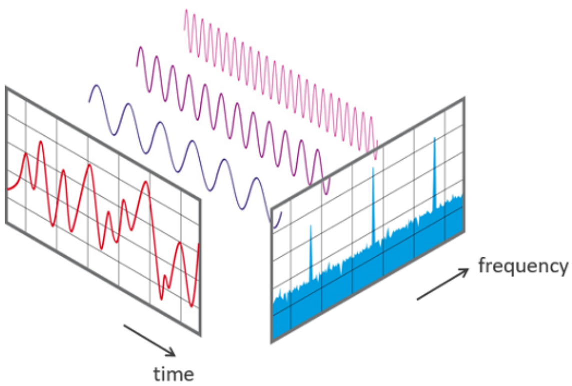


**Figure 1:** Time to frequency signal decomposition **Source: ?**

1 present, appearing, or found everywhere.

Since the sines and consines waves are in simple waveforms they can then be manipulated with relative ease. This process is constantly present in communications since the transmission of data over wires and radio circuits through signals and most devices nowadays perform ir frequently

And much more applications such as polynomial multiplication **?**, numerical integration, time-domain interpolation, x-ray diffracition ...

## 2.2 DISCRETE FOURIER TRANSFORM

The Fourier Transform of a finite sequence of equally-spaced samples of a function is the called the **Discrete Fourier Transform** (DFT), it converts a finite set of values in *time* domain to *frequency* domain representation. Its the most important type of transform since it deals with a discrete amount of data and has the popular algorithm in which is the center of attention of fourier transforms, which can be implemented in machines and be computed by specialized hardware.

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn} \tag{6}$$

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{\frac{i2\pi}{N}kn} \tag{7}$$

Notably, the discrete version of the Fourier Transform has some obvious differences since it deals with a discrete time sequence, the first difference is that the sum covers all elements of the input values instead of integrating the infinite domain of the function, but we can also notice that the exponential, similar to the aforesaid, divides the values by $N$ ($N$ being the total number of elements in the sequence) due to the inability to look at frequency and time $ft$ continuously we instead take the $k$'th frequency over $n$.

We can have a more simplified expansion of this formula with:

$$X_k = x_0 + x_1 e^{\frac{i2\pi}{N}k} + ... + x_{N-1} e^{\frac{i2\pi}{N}k(N-1)}$$

Having this sum simplified we then only need to resolve the complex exponential, and we can do that by replacing the $e^{\frac{i2\pi}{N}kn}$ by the euler formula as mentioned before to reduce the maths to a simple summation of real and imaginary numbers.

$$X_k = x_0 + x_1(\cos b_1 + i \sin b_1) + ... + x_{N-1}(\cos b_{N-1} + i \sin b_{N-1}) \tag{8}$$

$$\text{where } b_n = \frac{2\pi}{N}kn$$

Finally we'll be left with the result as a complex number

$$X_k = A_k + iB_k$$

EXAMPLE Let us now follow an example of calculation of the DFT for a sequence $x$ with N number of elements.

$$x = \begin{bmatrix} 1 & 0.707 & 0 & -0.707 & -1 & -0.707 & 0 & 0.707 \end{bmatrix}$$
$$N = 8$$

With this sequence we now want to transform it into the frequency domain, and for that we need to apply the Discrete Fourier Transform to each element $x_n \rightarrow X_k$, thus, for each $k$'th element of $X$ we apply the DFT for every element of $x$.

$$\begin{aligned} X_0 &= 1 \cdot e^{-\frac{i2\pi}{8} \cdot 0 \cdot 0} + 0.707 \cdot e^{-\frac{i2\pi}{8} \cdot 0 \cdot 1} + ... + 0.707 \cdot e^{-\frac{i2\pi}{8} \cdot 0 \cdot 7} \\ &= (0 + 0i) \\ X_1 &= 1 \cdot e^{-\frac{i2\pi}{8} \cdot 1 \cdot 0} + 0.707 \cdot e^{-\frac{i2\pi}{8} \cdot 1 \cdot 1} + ... + 0.707 \cdot e^{-\frac{i2\pi}{8} \cdot 1 \cdot 7} \\ &= (4 + 0i) \end{aligned}$$

$$...$$

$$\begin{aligned} X_7 &= 1 \cdot e^{-\frac{i2\pi}{8} \cdot 7 \cdot 0} + 0.707 \cdot e^{-\frac{i2\pi}{8} \cdot 7 \cdot 1} + ... + 0.707 \cdot e^{-\frac{i2\pi}{8} \cdot 7 \cdot 7} \\ &= (4 + 0i) \end{aligned}$$

And that will produce our complex-valued output in frequency domain, as simple as that.

$$X = \begin{bmatrix} 0i & 4+0i & 0i & 0i & 0i & 0i & 0i & 4+0i \end{bmatrix}$$

### 2.2.1 *Matrix multiplication*

The example shown above is done sequentially as if each frequency pin is computed individually, but there's a way to calculate the same result by using matrix multiplication **?**. Since the operations are done equally without any extra step we can group all analysing function sinusoids ($e^{-\frac{i2\pi}{N}kn}$), also refered to as twiddle factors.

$$W = \begin{bmatrix} \omega_N^{0 \cdot 0} & \omega_N^{1 \cdot 0} & \dots & \omega_N^{(N-1) \cdot 0} \\ \omega_N^{0 \cdot 1} & \omega_N^{1 \cdot 1} & \dots & \omega_N^{(N-1) \cdot 1} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_N^{0 \cdot (N-1)} & \omega_N^{1 \cdot (N-1)} & \dots & \omega_N^{(N-1) \cdot (N-1)} \end{bmatrix} = \begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & \omega & \dots & \omega^{(N-1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{(N-1)} & \dots & \omega^{(N-1) \cdot (N-1)} \end{bmatrix}$$

$$\text{where } \omega_N = e^{-\frac{i2\pi}{N}}$$

The substitution variable $\omega$ allows us to avoid writing extensive exponents.

The symbol $W$ represents the transformation matrix of the Discrete Fourier Transform, also called DFT matrix, and its inverse can be defined as.

$$W^{-1} = \frac{1}{N} \cdot \begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & \omega_N & \dots & \omega_N^{(N-1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_N^{(N-1)} & \dots & \omega_N^{(N-1)\cdot(N-1)} \end{bmatrix}$$

$$\text{where } \omega_N = e^{-\frac{i2\pi}{N}}$$

By using this matrix multiplication form we can have a more efficient way to compute the DFT in hardware.

$$X = W \cdot x$$

$$x = W^{-1} \cdot X$$

Moreover we might also want to normalize the matrix by $\sqrt{N}$ for both Matrix DFT and IDFT instead of just normalizing the IDFT by $N$, that will make $W$ a unitary matrix **?**. The advantage of using a unitary matrix is that we only need to reasign the constant substution variable $\omega_N$ to be able to invert the dft, the matrix multiplication stays the same for both DFT and IDFT. Nevertheless later we will verify that the use of sqrt function isn't desirable for the implementation of any dft.

EXAMPLE Continuing the example **??**, we can adapt the aplication of the DFT to the matrix multiplication form.

$$W = \begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & \omega_8 & \dots & \omega_8^7 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_8^7 & \dots & \omega_8^{49} \end{bmatrix}$$

$$\text{where } \omega_8 = e^{\frac{i2\pi}{8}}$$

$$X = W \cdot x = W \cdot \begin{bmatrix} 1 \\ 0.707 \\ \vdots \\ 0.707 \end{bmatrix} = \begin{bmatrix} 0 \\ 4+0i \\ \vdots \\ 4+0i \end{bmatrix}$$

It's conspicuous that the complexity time for each multiplication of every singular term of the sequence with the complex exponential value is $O(N^2)$, hence, the computation of the Discrete Fourier Transform rises exponentially as we use longer sequences. Therefore, over time new algorithms and techniques where developed to increase the performance of this transform due to its usefulness.

## 2.3 FAST FOURIER TRANSFORM

The Fast Fourier Transform (FFT) is a family of algorithms that effectively compute the Discrete Fourier Transform (DFT) of a sequence and its inverse. These algorihtms essentially compute the the same result as the DFT but the direct usage of the DFT formulation is too slow for its applications. Thus, FFT algorithms exploit the DFT matrix structure by employing a divide-and-conquer approach **?** to segment its application.

Over time serveral variations of the algorithms were developed to improve the performance of the DFT and many aspects were rethought in the way we apply and produce the resulting transform, in this section we'll cover at some of those variations.

There are many algorithms and aproaches on the FFT family such as the well known Cooley-Tukey, known for its simplicity and effectiveness to compute any sequence with size as a power of two, but also Rader's algorithm **?** and Bluestein's algorithm **?** which both deal prime sized sequences, and even the Split-radix FFT **?** that recursively expresses a DFT of length $N$ in terms of one smaller DFT of length $N/2$ and two smaller DFTs of length $N/4$.

We'll focus for now on the Cooley–Tukey algorithm, most specifically the radix-2 decimation-in-time (DIT) FFT and radix-2 decimation-in-frequency (DIF) FFT, which assume that the input sequence is a power of two sized.

### 2.3.1 *Radix-2 Decimation-in-Time FFT*

The Radix-2 Decimation-in-Time FFT algorithm rearranges the original Discrete Fourier Transform (DFT) formula into two subtransforms, one as a sum over the even indexed elements and other as a sum over the odd indexed elements. The **??** describes this procedure with already simplified math, and hints the recursive decomposition of the DFT of size $N/2$.

$$X_k = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} \cdot \omega_{N/2}^{k(2n)} + \omega_N^k \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} \cdot \omega_{N/2}^{k(2n+1)} \tag{9}$$

where $\omega_N = e^{\frac{i2\pi}{N}}$

This formulation successfully segments the full sized DFT into two $N/2$ sized DFT's of the even and odd indexed elements where the later is multiplied by a twiddle factor $\omega_N^k$.

This algorithm is a Radix-2 Decimation-in-Time in the sence that the time values are regrouped in 2 subtransforms, and the decomposition reduces the time values to the frequency domain. Since the understanding of this algorithm can be aplied recursively, the **??** illustrates the basic behaviour and represents the $N/2$ subtransforms with boxes that can be filled by the recursive application of this algorithm to produce the frequency domain sequence.

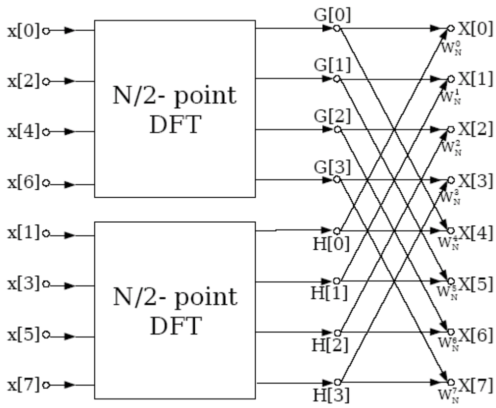


**Figure 2:** Radix-2 Decimation-in-Time FFT **Source: ?**

Effectivelly, this smaller DFT's are recursively reduced by this algorithm until theres only the computation of a length-2 DFT where its only applied the Cooley-Tukey butterfly operation **?** illustrated in **??**.

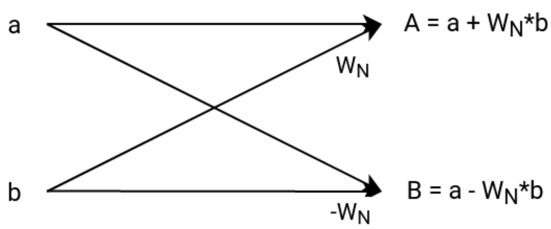


**Figure 3:** Cooley-Tukey butterfly

The complexity work within the algorithm is distributed with the DIT approach which decomposes each DFT by 2 having $\log N$ stages **?** while there are approximately $N$ complex multiplications needed for each stage of the DIT decomposition, therefore the multiplication complexity for a $N$ sized DFT is reduced from $O(N^2)$ to $O(N \log N)$ without any programming specific optimizations.

In practice, **??** demonstrates the aforesaid with an iterative representation of a possible implementation. Although this algorithm is congruent with a code implementation, its worth noting that the input sequence can either have real or complex numbers, since the arithmetic is the same for both domains the only thing that needs to be specialized is the operator overloading in the inner most loop.

**Algorithm 1:** Radix-2 Decimation-in-Time Forward FFT

**Data:** Sequence $in$ with size $N$ power of 2
**Result:** Sequence $out$ with size $N$ with the DFT of the input

```
/* Bit reversal step                                              */
foreach i = 0 to N − 1 do
    out[bit_reverse(i)] ← in[i]
end
/* FFT                                                            */
foreach s = 1 to log N do
    m ← 2^s;
    w_m ← exp(−2πi/m);
    foreach k = 0 to N − 1 by m do
        w ← 1;
        foreach j = 0 to m/2 do
            bw ← w · out[k + j + m/2];
            a ← out[k + j];
            out[k + j] ← a + bw;
            out[k + j + m/2] ← a − bw;
            w ← w · w_m;
        end
    end
end
return out;
```

### 2.3.2 *Radix-2 Decimation-in-Frequency FFT*

The Radix-2 Decimation-in-Frequency FFT algorithm is very similar to the DIT approach, its based on the same principle of divide-and-conquer but it rearranges the original Discrete Fourier Transform (DFT) into the computation of two transforms, one with the even indexed elements and other with the odd indexed elements; as in this simplified formulation **??**.

$$
\begin{aligned}
X_{2k} &= \sum_{n=0}^{\frac{N}{2}-1} (x_n + x_{n+\frac{N}{2}}) \cdot \omega_{N/2}^{kn} \\
X_{2k+1} &= \sum_{n=0}^{\frac{N}{2}-1} ((x_n - x_{n+\frac{N}{2}}) \cdot \omega_{N/2}^{kn}) \cdot \omega_N^{n}
\end{aligned}
\tag{10}
$$

where $\omega_N = e^{\frac{i2\pi}{N}}$

The DFT divided into these two transforms from the full sized DFT By separating these two transforms from the full sized DFT we get two distinct

Notably, this formulation distinguishes the full sized DFT into two $N/2$ sized DFT's of the even and odd indexed elements where the later is multiplied by a twiddle factor $\omega_N^k$ with both outside the same context.

This algorithm is a Radix-2 Decimation-in-Frequency since the DFT is deciminated into two distinct smaller DFT's and the frequency samples will be computed separately in different groups, as if the regrouping of the DFT's would reduce directly to the frequency domain. Since the understanding of this algorithm can be aplied recursively, the **??** illustrates the this behaviour and represents the $N/2$ subtransforms with boxes that can be filled by the recursive application of this algorithm to produce the frequency domain sequence. Aditionally this illustration can be compared to **??** since both are symmetrically identical.

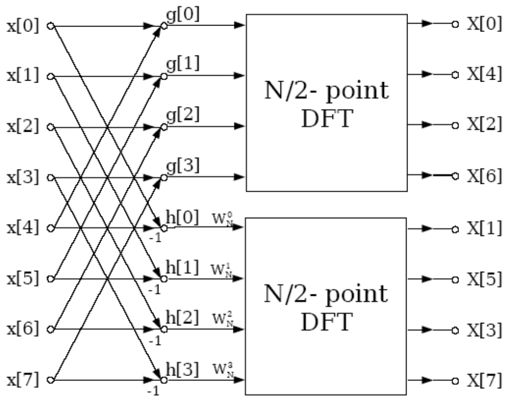


**Figure 4:** Radix-2 Decimation-in-Frequency FFT **Source: ?**

Similarly to the DIT version, the DFT can be recursively reduced by the DIF algorithm until theres only the computation of a length-2 DFT where its only applied the Gentleman-Sande butterfly operation **?** illustrated in **??**.

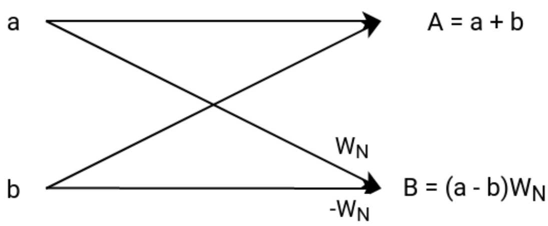


**Figure 5:** Gentleman-Sande butterfly

Since this algorithm has similarities with the DIT, its complexity also lives to this similarity, maintaining the same $O(N \log N)$ for number of multiplications, despite that, **??** and **??** might look different in number of arithmetic operations since the first has 1 addition, 1 subtraction, and 2 multiplications, and the second has 1 addition, 1 subtraction, and 1 multiplication, but effectively the $W_N \cdot b$ can be reused and only computed once as seen in **??**.

In practice, **??** demonstrates the aforesaid with an iterative representation of a possible implementation. Although this algorithm is congruent with a code implementation, its worth noting that the input sequence can either have real or complex numbers, since the arithmetic is the same for both domains the only thing that needs to be specialized is the operator overloading in the inner most loop.

**Algorithm 2:** Radix-2 Decimation-in-Frequency Forward FFT

**Data:** Sequence $in$ with size $N$ power of 2

**Result:** Sequence $out$ with size $N$ with the DFT of the input

```
/* FFT                                                   */
foreach s = 0 to log N − 1 do
    gs ← N ≫ s;
    w_gs ← exp(2πi/gs);
    foreach k = 0 to N − 1 by gs do
        w ← 1;
        foreach j = 0 to gs/2 do
            a ← in[k + j + gs/2];
            b ← in[k + j];
            in[k + j] ← a + b;
            in[k + j + gs/2] ← (a − b) · w;
            w ← w · w_gs;
        end
    end
end
/* Bit reversal step                                     */
foreach i = 0 to N − 1 do
    out[bit_reverse(i)] ← in[i]
end
return out;
```

NB: place here information about funding, FCT project, etc in which the work is framed. Leave empty otherwise.