Universidade do Minho
Escola de Engenharia
Departamento de Informática

Jorge Francisco Teixeira Bastos da Mota

High Performance Fourier Transforms on GPUs

November 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
António Ramires

November 2022

STATEMENT OF INTEGRITY

ABSTRACT

The continuous progress of the evolution of GPUs has increased the popularity of parallelizable algorithm implementations on this type of hardware. Fast Fourier Transforms is a family of algorithms which are very useful for the computation of Discrete Fourier Transforms which is applied in many practical scenarios for several areas.

Due to its usefulness and the need to optimize Fast Fourier Transforms this algorithms are effectively computed on the GPU to take advantage of its parallelizable variants

In this dissertation we provide, compare and analyze FFT implementations with popular libraries that are known to compute this efficiently, and provide specially forged GLSL implementations in the context of applications.

KEYWORDS FFT, GLSL, cuFFT, analysis, performance.

RESUMO

O progresso contínuo da evolução dos GPUs aumentou a popularidade das implementações de algoritmos paralelizáveis neste tipo de hardware. *Fast Fourier Transforms* são uma família de algoritmos úteis para o cálculo de transformadas de Fourier discretas que são aplicadas em muitos cenários práticos para diversas áreas.

Devido à sua utilidade e à necessidade de otimizar as *Fast Fourier Transforms*, esses algoritmos são efetivamente computados na GPU para aproveitar suas variantes paralelizáveis

Nesta dissertação, fornecemos, comparamos e analisamos implementações de FFT com bibliotecas populares que são conhecidas por computar isso de forma eficiente e fornecemos implementações GLSL especialmente forjadas no contexto de aplicativos.

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

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

1 INTRODUCTION

1.1 CONTEXTUALIZATION

The Fast Fourier Transforms have been present in our surroundings for a long time, they're used extensively in digital signal processing including 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 optimized algorithms to compute the Discrete Fourier Transform of some data, data that might be sampled from a signal, an oscillating object or even an image, which is transformed into the frequency domain allowing any kind of processing for a relatively low computational cost. Despite already existing pretty fast computations of the FFT, many applications require the processing of several transforms so its necessary to manage the implementations properties and achieve the best speed.

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 Flügge (2017), 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 Nvidia, a library provided by NVIDIA exclusively for their GPU's implemented for CUDA, and FFTW Frigo and Johnson (2012), a library dedicated to computations of FFT on the CPU with SIMD instructions support.

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 synchronized 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 and even make a one to one comparison of implementations on different frameworks. 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 Table 1.

- **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 chapter 1 exposes an 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, chapter 2 covers most of basic understandings and algorithms needed for later chapters, this will only take simple approachs to each concept to give intuitive insight and empirical explanations without proving it formally.

2

THE FOURIER TRANSFORM

It's noticeable the presence of Fourier Transforms in a great variety of apparent unrelated fields of application, even the FFT is often called ubiquitous due to its effective nature of solving a great hand of problems for the most intended time complexity. Some applications include polynomial multiplication Jia (2014), numerical integration, time-domain interpolation, x-ray diffraction. Furthermore it is present in several fields of study such as Applied Mechanics, Signal Processing, Sonics and Acoustics, Biomedical Engineering, Instrumentation, Radar, Numerical Methods, Electromagnetics, Computer Graphics and more Brigham (1988).

In **Signal Analysis** when representing a signal with amplitude as function of time, it can be translated to the frequency domain, a domain that consists of signals of sines and cosines waves of varied frequencies, as illustrated in Figure 1, but to calculate the coefficients of those waves we use the Fourier Transform.

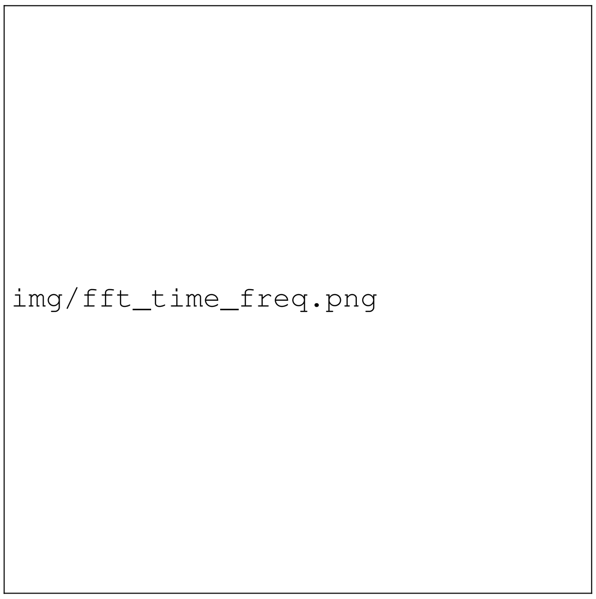

Figure 1: Time to frequency signal decomposition **Source:** NTiAudio

Since the sine and cosine waves are in simple wave forms 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 it frequently.

In this introductory chapter we present a rudimentary introduction to the Fourier Transform in section 2.1 and describe the discrete version of the Fourier Transform, which we focus more in this dissertation in section 2.2 and finalizing with the state of the art of the most popular algorithms in section 2.3.

2.1 CONTINUOUS 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:

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

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

The definition of Equation 1 integral is only valid if the integral exists for every value of parameter f. This formulation shows the usage of complex-valued domain, 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 Equation 2, we can rewrite the Fourier Transform for an equivalent by resolving the Euler's constant for a sine and cosine sum, as represented in Equation 3.

$$e^{ix} = \cos x + i \sin x \tag{2}$$

$$X(f) = \int_{-\infty}^{+\infty} x(t)(\cos(-ft) + i\sin(-ft))dt \tag{3}$$

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.

$$\begin{aligned} X_a(f) &= \int_{-\infty}^{+\infty} x(t)\cos(ft)dt \\ X_b(f) &= \int_{-\infty}^{+\infty} x(t)\sin(ft)dt \end{aligned} \tag{4}$$

This model of the Fourier transform applied to infinite domain functions is called **Continuous Fourier Transform** and it is targeted to the calculation of the this transform directly to functions with only finite discontinuities in $x(t)$.

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. It is an important version of the Fourier transform since it deals with a discrete amount of data, therefore programmers use it to implement in machines or to compute it in specialized hardware.

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn} \tag{5}$$

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

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 sum of real and imaginary numbers.

$$X_k = x_0 + x_1(\cos b_1 + i \sin b_1) + ... + x_{N-1}(\cos b_{N-1} + i \sin b_{N-1}) \tag{7}$$

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

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 $x_n \to X_k$, 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 Rao and Yip (2018). 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.

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

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

The substitution variable ω 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.

$$X = W \cdot x$$

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

Its also worth noting that normalizing the DFT and IDFT matrix be by $\sqrt{N}$ instead of just normalizing the IDFT by N, will make W a unitary matrix Horn and Johnson (2012). However this normalization by $\sqrt{N}$ is not common in FFT implementations.

EXAMPLE Continuing the example 2.2, 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 compute the Discrete Fourier Transform (DFT) of a sequence, and its inverse, efficiently. 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 Chu and George (1999) 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.

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 Rader (1968) and Bluestein's algorithm Bluestein (1970) to deal with prime sized sequences, and even the Split-radix FFT Yavne (1968) that recursively expresses a DFT of length N in terms of one smaller DFT of length $N/2$ and two smaller DFTs of length $N/4$.

The next two sections focus on the Cooley–Tukey algorithm, most specifically the radix-2 decimation-in-time (DIT) FFT and radix-2 decimation-in-frequency (DIF) FFT, both requiring the input sequence to have a power of two size. These two variations of the Cooley–Tukey algorithm represent the state of the art of what an individual in need to implement FFT will most likely be familiar with.

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. Cooley and Tukey proved this possibility of dividing the DFT computation into two smaller DFT by exploiting the symmetry of this division, as presented in Equation 8. Hence it is hinted the recursive definition of this algorithm on both DFT of size $N/2$.

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

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} \cdot e^{-\frac{i2\pi}{N}k(2m)} + \sum_{m=0}^{N/2-1} x_{2m+1} \cdot e^{-\frac{i2\pi}{N}k(2m+1)}$$

$$X_k = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} \cdot \omega_{N/2}^{k(2n)} + \omega_N^k \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} \cdot \omega_{N/2}^{k(2n+1)} \tag{8}$$

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 ω_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 Figure 2 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.

img/dit_fft.png

Figure 2: Radix-2 Decimation-in-Time FFT **Source:** Jones (2014)

Effectively, 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 Chu and George (1999) illustrated in Figure 3.

The complexity work within the algorithm is distributed with the DIT approach which decomposes each DFT by 2 having $\log N$ stages Smith (2007) while there are 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 optimisations.

One consequence of using this algorithm is that the elements it produces are out of order, therefore, applying the inverse on the Fourier transform of a sequence won't return the same elements. Consequently, we need to apply some kind of data reordering to the sequence.

The most well-known reordering technique involves explicit bit reversal of the elements position, hence this algorithm and the one in subsection 2.3.2 both contain a bit reversal step but used in different ways. For the DIT FFT the input sequence of the algorithm must be in bit reversal order since the algorithm returns a natural order sequence but requires the input to be bit reversed. This way we need to apply the bit reversal at the beginning before applying the algorithm.

img/dit_butterfly.png

Figure 3: Cooley-Tukey butterfly

However it seems shadowed in algorithm 1 the implementation of the bit reversal is quite simple and any decent version can be used in regards of this algorithm. The key idea is that each element must be placed in its bit reversed index, so for each element of the sequence the $bit\_reverse$ version of the index is calculated and the element is swapped with the on in the original index.

The $bit\_reverse$ of an index depends directly on the indexing domain of the input sequence, therefore it needs the size N, or more precisely the $\log N$ value, to use as a reference to reverse the bit order while maintaining the value within the sequence range.

For example, for a sequence of size 16, we have some index with $\log 16$ bits $b_1 b_2 b_3 b_4$, which corresponds to the bit reversed index as $b_4 b_3 b_2 b_1$.

In practice, algorithm 1 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
    $out[\text{bit\_reverse}(i)] \leftarrow in[i]$
end
/* FFT                                                                */
foreach $s = 1$ to $\log N$ do
    $m \leftarrow 2^s$;
    $w_m \leftarrow \exp(-2\pi i/m)$;
    foreach $k = 0$ to $N - 1$ by $m$ do
        $w \leftarrow 1$;
        foreach $j = 0$ to $m/2$ do
            $bw \leftarrow w \cdot out[k + j + m/2]$;
            $a \leftarrow out[k + j]$;
            $out[k + j] \leftarrow a + bw$;
            $out[k + j + m/2] \leftarrow a - bw$;
            $w \leftarrow w \cdot w_m$;
        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 Equation 9.

$$
\begin{aligned}
X_{2k} &= \sum_{n=0}^{\frac{N}{2}-1} (x_n + x_{n+\frac{N}{2}}) \cdot \omega_{N/2}^{kn} \\
X_{2k+1} &= \sum_{n=0}^{\frac{N}{2}-1} ((x_n - x_{n+\frac{N}{2}}) \cdot \omega_{N/2}^{kn}) \cdot \omega_N^n
\end{aligned}
\tag{9}
$$

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

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 ω_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 Figure 4 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 Figure 2 since both are symmetrically identical.

img/dif_fft.png

Figure 4: Radix-2 Decimation-in-Frequency FFT **Source:** Jones (2014)

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 Chu and George (1999) illustrated in Figure 5.

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, Figure 5 and Figure 3 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 algorithm 1.

As mentioned in subsection 2.3.1 the bit reversal in DIF works a bit differently, this algorithm does the exact opposite of the DIT since it requires a natural order sequence and returns a bit reversed output, justifying why this step is applied after the algorithm.

In practice, algorithm 2 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 img/dif_butterfly.png

Figure 5: Gentleman-Sande butterfly

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
    $gs \leftarrow N \gg s$;
    $w_{gs} \leftarrow \exp(2\pi i/gs)$;
    foreach $k = 0$ to $N - 1$ by $gs$ do
        $w \leftarrow 1$;
        foreach $j = 0$ to $gs/2$ do
            $a \leftarrow in[k + j + gs/2]$;
            $b \leftarrow in[k + j]$;
            $in[k + j] \leftarrow a + b$;
            $in[k + j + gs/2] \leftarrow (a - b) \cdot w$;
            $w \leftarrow w \cdot w_{gs}$;
        end
    end
end
/* Bit reversal step                                                    */
foreach $i = 0$ to $N - 1$ do
    $out[\text{bit\_reverse}(i)] \leftarrow in[i]$
end
return $out$;
```

3

ALGORITHMS ANALYSIS

To flavour this pre-dissertation report some work of benchmarking and analysis were done to compete with the theoretical explanations addressed on chapter 2. Hence, some implementations were tested to provide coherence to what has been studied, and algorithms such as algorithm 1 Radix-2 Decimantion-in-Time algorithm 2 Radix-2 Decimantion-in-Frequency were timed in Table 2.

	Size 128	Size 256	Size 512	Size 1024	Size 2048
DFT	5.16593	17.2782	70.5689	293.104	1246.44
FFT DIT	0.169113	0.37668	0.86415	1.8793	4.47742
FFT DIF	0.159458	0.378722	0.881921	1.90661	4.13369
Recursive FFT	0.210895	0.485643	1.4421	2.32922	5.1178

Table 2: FFT algorithms benchmark. Results are measured in milliseconds for forward and inverse computation with varying input sizes

As we can see the Discrete Fourier Transform increases exponentially for higher sized sequences, as expected all FFT variants perform critically better than the original formulation.

One variant that wasn't exposed much in the above chapters is the Recursive FFT algorithm, which corresponds to a Decimation-in-Time aproach with recursive reduction, therefore this algorithm aggregates the divide-and-conquer method but with the disadvantage of recursive function overhead. This recursive look at the DIT approach can be easier to implement since the bit reversal step isn't explicitly applied before starting the FFT.

Finally, as expected the DIT and DIF algorithms overrule the other alternatives for every sized input.

4

COMPUTATION OF THE FOURIER TRANSFORM

4.1 IMPROVING THE COOLEY-TUKEY ALGORITHM

Nowadays there is a lot more to the computation of FFT's than just the basic Cooley-Tukey algorithm described in 2.3.1 there are more algorithms, variations and improvements that enhace the computation in many aspects. Even the selected hardware can change the restrictions on the computation of this primitive.

One could optimize the fft by providing precomputed twiddle factors, or spare space for the output sequence by adopting an in-place FFT algorithm and all those factors influence the performance. Evidently a balance must be stablished depending on the constraints for the FFT to take advantage on the target environment.

«««< HEAD

4.1.1 *Natural order Cooley-Tukey*

The main characteristic of CooleyTukey algorithm is the presence of the bit reversal step needed to store the result in natural order. Both the DIT and DIF version apply this step, for the first it is done before the first stage, and for the later version its delayed up until the end of the last stage to be applied. This bit reversal step is essential for this algorithm to obtain the same input sequence if the inversion of the forward fft is computed, however, it is possible to change the algorithm a bit to rearrange the output in natural order. =======

4.1.2 *Natural order Cooley-Tukey*

TODO: Paste here the lost paragraph

Despite existing already really fast solutions for index bit reversal (Prado (2004)), this *shufle* step still weights the algorithms with extra overhead. The natural order Cooley-Tukey FFT is a modification of the Cooley-Tukey algorithm that allows the removal of this step by computing the butterfly and reordering the elements per stage OKAHISA).

At the end of each stage the even and odd elements are composed in such a way that the elements will be in natural order at the end. This composition follows the indexing scheme described in Equation 10.

img/dif_elements_composition.png

Figure 6: Chain of even and odd compositions over each stage for a natural order DIF

$$x[q + 2 * p] = y[q + p] \tag{10}$$

$$x[q + 2 * p + 1] = y[q + p + m] \tag{11}$$

Where x and y are alternated sequences for read write over each stage, q corresponds to the sub FFT offset in this stage, p is the index of the sub fft element shift for the current butterfly being computed, and finally the m corresponds to the size of the sub FFT divided by 2.

Although this composition got rid of the bit reversal step in the Cooley-Tukey's algorithm, the performance is deprecated with more work for each stage. Work which may seem unnecessary after we find out about the Stockham algorithm that supersedes this algorithm.

We can also note that the composition of even and odd elements wont be necessary on the last stage of the DIF FFT since it will be a passthrough of the elements, so this is an unnecessary step, however this algorithm is an intermediate step to make the stockham algorithm more rational so there's no need to worry about this detail. »»»> 0d7be6795fd826dc05857cf7de8a9f39e5ea5e7f

To replace the bit reversal step we can introduce an equivalent procedure applied in each stage after the butterflies are computed, this procedure is illustrated in Figure 7

However this change gets rid of the bit reversal step, it is not optimal, for each stage there's a swap over the subtransform elements beyond the butterflies and that's what leads us to the next section, the stockham algorithm.

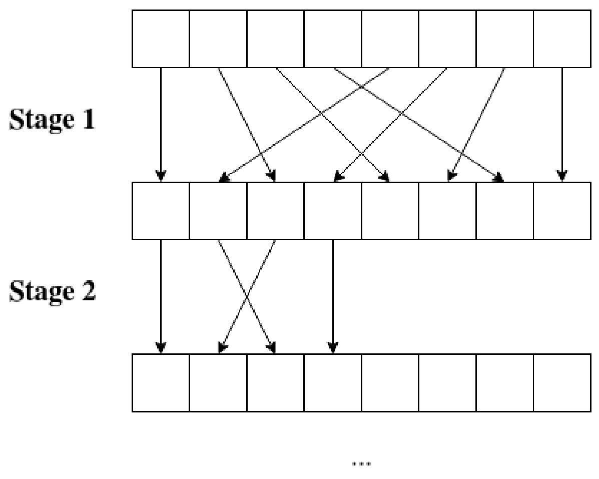

Figure 7: Natural order composition

4.1.3 *Stockham algorithm*

As mentioned on the former section, the Stockham algorithm comes to save us from the bit reversal step, it does this by taking advantage of a reordering of the elements Govindaraju et al. (2008) illustrated in Figure 6. The natural order elements are composed stage by stage and the butterfly computations stay the same, so this approach takes advantage of the Cooley-Tukey algorithm and turns it into a more suitable form for highly parallelizable hardware such as GPUs, making it a best fit for our implementation in any GPU programmable language.

Similarly to subsection 4.1.2 the algorithm requires the usage alternated sequences for read write over each stage, this prevents the read of an element which has been altered before read in a given stage therefore the return sequence will depend on the $\log N$ number parity.

The stockham algorithm is described in algorithm 4 and this version may seem strictly different from the Cooley Tukey, specially the inner most loop, however most of the logic stays the same, the indexing is a bit different and simpler for this version and the if statements are a consequence of using alternated ping pong sequences since there has to be branches for read and write of both arrays for this out-of-place algorithm.

Algorithm 3: Stockham Radix-2 Decimation-in-Time Forward FFT

```
Data: Sequence pingpong0 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
    $gs \leftarrow N \gg s$;
    $stride \leftarrow 1 \ll s$;
    foreach $i = 0$ to $N - 1$ do
        $p \leftarrow i$ div $stride$;
        $q \leftarrow i$ mod $stride$;
        $w_p = \exp(-2\pi i/gs * p)$;
        if $stage\ mod\ 2 == 0$ then
            $a \leftarrow pingpong0[q + s * (p + 0)]$;
            $b \leftarrow pingpong0[q + s * (p + gs/2)]$;
            /* Perform butterfly                                    */
            $pingpong1[q + s * (2 * p + 0)] = a + b$;
            $pingpong1[q + s * (2 * p + 1)] = (a - b) * w_p$;
        end
        else
            $a \leftarrow pingpong1[q + s * (p + 0)]$;
            $b \leftarrow pingpong1[q + s * (p + gs/2)]$;
            /* Perform butterfly                                    */
            $pingpong0[q + s * (2 * p + 0)] = a + b$;
            $pingpong0[q + s * (2 * p + 1)] = (a - b) * w_p$;
        end
    end
end
if $\log N\ mod\ 2 == 0$ then
    return pingpong1;
end
else
    return pingpong0;
end
```

Despite this algorithm description being an algorithmic or more close to a CPU implementation than GPU it will give us a solid structure to use as reference for our comparison subjects since it is feasible to make a similar implementation for GPGPU programmable languages due to the way of indexing and the usage of alternating sequences.

4.1.4 *Radix-4 instead of Radix-2*

Stepping forward on the need to optimize this algorithm we reach the topic of higher radix alternatives other than just radix-2, FFT algorithms can use higher radix for better performance and even mixed radix Singleton (1969) for more irregular sized input sequences.

The Radix-4 is a good improvement over Radix-2 since getting to higher radixes than 4 might result in reduced performance, so improving the Stockham algorithm with Radix-4 upgrades the computation of the butterflies while reducing the number of stages which will be crucial in later sections.

Theoretically Radix-4 formulation can be twice as fast as a Radix-2 Hussain et al. (2010) since it only takes half the stages with a bit more complexity in the butterflies which are usually called dragonflies, and additionally can use less multiplications with better factorizations Marti-Puig and Bolano (2009).

`tese/img/dragonfly.png`

Figure 8: Radix-4 FFT butterfly structure **Source:** Marti-Puig and Bolano (2009)

TODO: WIP

Algorithm 4: Stockham Radix-2 Decimation-in-Time Forward FFT

```
Data: Sequence pingpong0 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
    $gs \leftarrow N \gg s$;
    $stride \leftarrow 1 \ll s$;
    foreach $i = 0$ to $N - 1$ do
        $p \leftarrow i$ div $stride$;
        $q \leftarrow i$ mod $stride$;
        $w_p = \exp(-2\pi i / gs * p)$;
        if $stage\ mod\ 2 == 0$ then
            $a \leftarrow pingpong0[q + s * (p + 0)]$;
            $b \leftarrow pingpong0[q + s * (p + half_g s)]$;
            /* Perform butterfly                                      */
            $pingpong1[q + s * (2 * p + 0)] = a + b$;
            $pingpong1[q + s * (2 * p + 1)] = (a - b) * w_p$;
        end
        else
            $a \leftarrow pingpong1[q + s * (p + 0)]$;
            $b \leftarrow pingpong1[q + s * (p + gs/2)]$;
            /* Perform butterfly                                      */
            $pingpong0[q + s * (2 * p + 0)] = a + b$;
            $pingpong0[q + s * (2 * p + 1)] = (a - b) * w_p$;
        end
    end
end
if $\log N\ mod\ 2 == 0$ then
    return pingpong1;
end
else
    return pingpong0;
end
```

5

IMPLEMENTATION ON THE GPU

To be able to analyse the implementation of FFT on the GPU, we need to provide some background on what kind of hardware were dealing with, whats constraints are relevant and what we can or can't do, after, we proceed with with description of how the Fourier Transform suits the GPU programming model and finally we'll move forward to a detailed analysis of the implementation of FFT algorithms in a compute shader in GLSL.

5.1 GPU PROGRAMMING MODEL

The Graphics Processing Unit (GPU) provides much higher instruction throughput and memory bandwidth than the CPU within a similar price and power envelope having a much faster performance growth curve. Many applications leverage these higher capabilities to run faster on the GPU than on the CPU. While the hardware may change some components and architecture on different models, fundamentally most modern GPUs adopt a specific kind of single instruction multiple data (SIMD) stream architecture which is single instruction multiple threads (SIMT) that is an extension of the SIMD paradigm with large scale multi-threading, streaming memory and dynamic scheduling.

A SIMT stream architecture forwards an instruction to multiple threads with dedicated memory for each instance, allowing data-level parallelism to compute more quickly and effectively. A modern CPU uses a von Neumann architecture, so it executes instructions sequentially one at a time and updates the memory progressively, however a stream architecture processor works in a slightly different way, they contain multiple streams of simpler processors with shared memory just like the illustration in Figure 9. These processors execute programs called kernels that receive a finite set of input fragments to produce another set of output fragments in parallel Fernando et al. (2004).

With the introduction of programmable General Purpose Graphics Processing Unit (GPGPU) the industry required a lot more parallel algorithms.

platforms allow developers to ignore the language barrier that exists between the CPU and the GPU and, instead, focus on higher-level computing concepts.

Developers and researchers often attend more specific scenarios and they configure multiple GPUs with distributed workflows in sync with each other to take better advantage of this hardware for more intensive problems Heldens et al. (2022).

img/cpu_gpu.png

Figure 9: CPU architecture compared with a GPU architecture

The GPU programming model although sophisticated, at its core it provides thread groups hierarchies, shared memories and GPU barriers for synchronization of threads, such abstractions provide data and thread level parallelism for the developer to utilize.

In this programming model we introduce the concept of granularity, that refers to the amount of computation relatively to the transfer of data. It is used to describe types of parallelism as fine-grained or coarse-grained. Fine-grained parallelism describes a small multi threaded tasks in kernel size and execution time with frequent small data transfers between the processors. On the opposite side there's coarse-grained parallelism that describes larger amounts of computation followed by larger infrequent data transfers NVIDIA.

This concept is important to reflect on the association of the GPU programming model with further details on the provided implementations within the next chapters.

Most GPGPU programming frameworks such as CUDA provide coarse-grained data and task parallelism with nested fine-grained data and thread parallelism. This type of task hierarchy architecture promotes the partitioning of the problems to independently solvable subgroups of smaller problems which fit into smaller blocks of threads that can be solved cooperatively in parallel. This relevant information provides us more insight of how the work groups for FFT should be dispatched.

5.2 2D FOURIER TRANSFORM ON THE GPU

Computing a 2D Fourier Transform requires a two dimensional input sequence to produce another for the output. However we could use any usable arbitrary values in the the real world we often use this 2D FFT on images and it isn't immediately obvious how this should be applied since the target data source might have three color channels, therefore three different values for this multidimensional sequence element that can be used, and not even in floating point complex domain. Precisely, it needs to be adapted to the application use case, we may use it as greyscale image if want a derivation of the luminance via quantized RGB signals of the image (**?**), use only the values of one channel, or compute multiple FFT's for each channel values. Either way we will preferably at the end have a two dimensional buffer with floating point complex values prepared to be used.

The 2D FFT is computed by performing single dimension FFT's for every row and then for every columns after that **?**, so we can divide its application to a horizontal and vertical pass. This describes the way 2D FFT are computed but it is independent of the 1D FFT implementation chosen, so there's freedom to use any type of algorithm.

tese/img/2d_fft.png

Figure 10: High level illustration of horizontal and vertical passes

At the end of a forward FFT vertical pass the result will be a 2D complex buffer with the frequency domain values of the original image as illustrated in Figure 10.

5.3 GLSL IMPLEMENTATION

The implementations were made using GLSL, a high-level shader language for graphics API's such as OpenGL, and it was used the compute pipeline from OpenGL to integrate a FFT implementation using compute shaders, which are general purpose programmable shaders.

Since there are many aspects that may impact on the performance the implementation was an interactive process that required researching and testing to compose an optimal solution. One of the main targets was to keep the code generalised so that it can be used as base for other implementations using FFTs.

The next sections go in detail about the way every major iteration evolved into the next one and why there was the need to do it, starting by the Cooley-Tukey algorithm then progressively improving on the Stockham algorithm, all this while implementing good GPGPU programming strategies.

5.3.1 *Cooley-Tukey*

The GPU implementation took as a starting point was with the DIT Cooley-Tukey algorithm, since it is the most popular one with time complexity of $O(N \log N)$, and it is based on the iterative version adapted for parallel processors.

Since this implementation is highly parallel there's the need to separate the reads and writes for each processor into two different buffers in memory due lack of order between processors, therefore the declaration of two complex pingpong buffers.

```
layout (binding = 0, rg32f) uniform image2D pingpong0;
layout (binding = 1, rg32f) uniform image2D pingpong1;
```

Listing 5.1: Input buffer bindings

The read write control for this buffers can be achieved with a flag variable `pingpong`.

Initially this algorithm will work with a pass per stage approach, where a kernel is dispatched every stage and since there's a synchronization step at the end of the pass its granted that all the work groups have finished off writing to the buffer when a pass ends. This case holds up for both the horizontal and vertical FFT steps.

As a result, each kernel has the opportunity to work within every segment of the image, so the local threads can be dispatched with two dimensions, so each work group will have a total of 32 local threads, a reference number used in this implementations for setting up local threads in a work group, this may vary depending on the GPU for optimal performance but 32 is a good number to fill in the thread warp size os most GPUs.

So an example dispatch group for this implementation could be $(fft\_width/8, fft\_height/8)$ work groups since 8 is the number of threads in the y axis

By using GLSL there is some advantages on the complex values operations, since the addition and subtraction for vector types already have operators overloading which function the same as in the complex domain. However the multiplication works a bit differently so we need to provide an auxiliary function to abstract and support this operator.

```
vec2 complex_mult(vec2 v0, vec2 v1) {
 return vec2(v0.x * v1.x - v0.y * v1.y,
       v0.x * v1.y + v0.y * v1.x);
}
```

Listing 5.2: Complex multiplication

Due to the adoption of a different programming paradigm the FFT segment iteration loop doesn't exist such as in algorithm 1, instead the processors identifiers are used to fetch the index of the butterflies they're gonna work on based on the work groups and threads dispatch setup, and this holds up for any implementation using compute shaders.

```
int line = int(gl_GlobalInvocationID.x);
int column = int(gl_GlobalInvocationID.y);
```

Listing 5.3: Invocation indices

Since this first approach is a dynamic implementation that invokes a pass per stage some stage control variables need to be feed into the shader in order to compute the correct butterfly index or control the butterfly process.

```
uniform int pingpong;
uniform int log_width;
uniform int stage;
uniform int fft_dir;
```

Listing 5.4: Uniform control variables

Effectively, we use these shader uniform input variables and obtain actual index we're gonna use on the 1D FFT of the image.

```
int group_size = 2 << stage;
int shift = 1 << stage;

int idx = (line % shift) + group_size * (line / shift);
```

Listing 5.5: FFT element index

To calculate the twiddle factor we use Euler's formula such as in Equation 2 and resort to the control variable `fft_dir` to flip the twiddle factor for the inverse if we want to reuse this shader.

```
vec2 euler(float angle) {
    return vec2(cos(angle), sin(angle));
}
```

```
void main() {
    // ...
    vec2 w = euler(fft_dir * 2 * (M_PI / group_size) * ((idx % group_size) %
  shift));
    // ...
}
```

Now with the computed twiddle factor we may proceed to compute and store the Cooley-Tukey FFT butterfly, and that's where we need to control the reads and writes for each stage. Since the `pingpong` variable is toggled every pass invocation, it is used to choose with what image we will use to lookup the elements and which one to store into, and that is achieved with an if statement, just like it is used in algorithm 4.

The butterfly computation itself is simply calculated as a Cooley-Tukey DIT butterfly as illustrated in Figure 3.

```
if (pingpong == 0) {
    // Read
    a = imageLoad(pingpong0, ivec2(idx, column)).rg;
    b = imageLoad(pingpong0, ivec2(idx + shift, column)).rg;

    // Compute and store
    vec2 raux = a + complex_mult(w, b);
    imageStore(pingpong1, ivec2(idx, column), vec4(raux, 0, 0));
    raux = a - complex_mult(w, b);
    imageStore(pingpong1, ivec2(idx + shift, column), vec4(raux, 0, 0));
}
else {
    // Read
    a = imageLoad(pingpong1, ivec2(idx, column)).rg;
    b = imageLoad(pingpong1, ivec2(idx + shift, column)).rg;

    // Compute and store
    vec2 raux = a + complex_mult(w, b);
    imageStore(pingpong0, ivec2(idx, column), vec4(raux,0,0));
    raux = a - complex_mult(w, b);
    imageStore(pingpong0, ivec2(idx + shift, column), vec4(raux,0,0));
}
```

Finally there's only one step missing, the bit reversal of indices to have a natural order result. A `bit_reverse` function could be easily defined, however there's already a GLSL alternative which is `bitfieldReverse` together with `bitfieldExtract` **?**.

Despite not having a noticeable performance hit, it is good practice to use GLSL predefined functions and operators since they might be optimised for that specific device hardware and using these functions instead of a handmade implementation reduces the kernel size significantly about approximately 400 bytes (evaluated using *glslang*) since they are reusable functions.

```
int bit_reverse(int k) {
    uint br = bitfieldReverse(k);
    return int(bitfieldExtract(br, 32 - log_width, log_width));
}
```

This auxiliary function will be conditionally used inside the branching if statement for alternating read writes to be only applied on the first stage of transform both for the possible values of `pingpong == 0` and `pingpong == 1`, since the vertical pass might start reading on the first pingpong buffer. This is not the case for the horizontal pass, its ensured that the first stage will always read from `pingpong0` reforcing that the bit reverse branching when `pingpong == 0` is disposable.

```
if (pingpong == 0) {
    if (stage == 0) {
        a = imageLoad(pingpong0, ivec2(bit_reverse(idx), column)).rg;
        b = imageLoad(pingpong0, ivec2(bit_reverse(idx + shift), column)).rg;
    }
    else {
        a = imageLoad(pingpong0, ivec2(idx, column)).rg;
        b = imageLoad(pingpong0, ivec2(idx + shift, column)).rg;
    }

    // ... Compute and store results
}
else {
    a = imageLoad(pingpong1, ivec2(idx, column)).rg;
    b = imageLoad(pingpong1, ivec2(idx + shift, column)).rg;

    // ... Compute and store results
}
```

With all this steps aggregated, the shader for the horizontal pass of this FFT DIT Cooley-Tukey implementation is presented in Listing A.1, see Appendix A.

For the vertical pass shader there is however one extra multiplication by `mult_factor` in the butterfly results for the last stage when the pass is inverse. This corresponds to the normalization of the multidimensional transform similar to the normalization in the inverse DFT in **??**, this is demonstrated in Listing A.2.

We can already see in **??** the above code a lot of branching that might be undesirable on the GPU ...

All stages in one pass

The previous implementation demonstrates a generic 2D FFT that may be reused for multiple FFT sizes, however this comes at a cost of efficiency when it comes to the synchronization of the of the stage itself, moreover it requires use multiple uniform variables for control of the FFT that are transferred between CPU and GPU when there are updates in between stages.

Undoubtedly the best solution here is to port this implementation synchronization step to be kernel-wise and this detail changes quite a bit the properties of how the code is structured and how it may be dispatched.

At the moment there isn't a way to trivially ensure the reads and writes of all work groups **?** without interrupting at the end of the stage, but there is functions that make use of barriers that synchronise all the threads within a work group. However there is a lot of literature on behalf of GPU compute execution synchronization (maybe list some references here) we'll make use of the GLSL predefined barrier synchronization functions.

The current grouping of threads doesn't allow the use of these barrier synchronization since the computation of one dimensional FFT is distributed between multiple work groups, so using a call to `barrier()` inside the kernel wouldn't fix the race conditions of several segments of the image. We could however, change the setup of these work groups in such way that each 1D FFT threads fit in one work group as illustrated in Figure 11.

`tese/img/inv_space_2d.png`

`tese/img/inv_space_1d.png`

Figure 11: Difference in invocation spaces for size 256 FFT to allow barrier synchronization between local threads

By restricting the invocation space to be only one dimensional we grant the possibility to use the barrier correctly but at the cost of resizability, the work group local size must now restricted to half the size of the FFT which corresponds to the width of the image.

5.3.2 *Radix-2 Stockham*

5.3.3 *Radix-4 Stockham*

5.4 CASE OF STUDY

5.4.1 *Tensendorf waves*

5.4.2 *Implementation details*

5.4.3 *Results*

6 ANALYSIS AND COMPARISON

Finally on this chapter an evaluation of the explored implementations and improvements is provided followed by an empirical analysis based on the results and testing that was done.

To establish a reference point on the results provided, we used cuFFT which is the CUDA Fast Fourier Transform library from NVIDIA.

Additionally, to deepen the analysis, this chapter also delivers an equivalent comparison of the researched algorithms applied to a different compute framework such as CUDA.

6.1 CUFFT

The cuFFT library is the NVIDIA framework designed to provide high performance FFT exclusively on its own GPUs that supports a wide range of FFT inputs and settings that compute FFTs effitiently on NVIDIA GPUs.

The cuFFT library is acknowledged as one of the most efficient FFT GPU framework for the flexibility it provides and it is "*de-facto a standard GPU implementation for developers using CUDA*" (**?**). Furthermore, it offers all kinds of settings needed for most use cases, such as multidimensional transforms, complex and real-valued input and output, support for half, single and double floating point precision, execution of multiple transforms simultaneously and finally since all this is implemented using CUDA we can take advantage of streamed execution, enabling asynchronous computation and data movement.

Unfortunately, as mentioned before the main downside of cuFFT is the unavailability of this library for GPUs from other vendors.

The cuFFT library uses algorithms highly optimized for input sizes that can be written in the form $2^a \times 3^b \times 5^c \times 7^d$, so it factorizes the input size to allow arbitrary sized FFT sequences furthermore sizes with lower prime factors have intuitively better performance.

To use the results of the cuFFT library as a reference point, we need to establish equivalent conditions to that of the GLSL implementations:

- Out-of-place 2D FFT, input buffer is different from the output buffer;
- Base 2 input sizes, such as 128, 256, 512 and 1024;

- Complex to complex FFT, input and output buffer are complex valued;
- The benchmarks are average milliseconds of multiple executions, however the first dispatch is not taking into account since takes extra time to setup things on the GPU.

The results of the cuFFT out-of-place benchmarks can be found in Figure 13 and Figure 14.

6.2 IMPLEMENTATION ANALYSIS IN GLSL

The implementations discussed in section 5.3, as said before, were studied and benchmarks were made to come to a conclusion about the advantages and disadvantages of using each one and how do they perform. With this in mind we prepared an interactive test environment using Nau 3D engine **?** and profiled it using an internal pass profiler.

All benchmarks, the ones in this section and section 6.3 were tested with the following hardware and software configuration:

- **CPU:** Intel(R) Core(TM) i7-8750H @ 2.20GHz;
- **GPU:** NVIDIA GeForce GTX 1050 Ti Max-Q;
- **NVIDIA driver:** 511.65;
- **CUDA version:** V11.6.124;
- **GLSL version:** 4.60.

In subsubsection 5.3.1 we discussed how the implementation would benefit by having a unique pass that synchronized by stage instead of dispatching multiple stage passes, accordingly we can clearly notice this difference in Figure 12.

The results in Figure 12 show us how a real time application would behave by adopting both strategies. The pass per stage approach introduces a lot of runtime overhead in between stages, since it needs to update the stage for the next iteration dispatch, on the other hand the unique pass kernel is highly optimized for its own size so most calculation are inlined by the GLSL compiler and the synchronization is kept inside the GPU until the kernel is done executing.

As we can see in Figure 13 the GLSL radix-2 implementation of the Stockham algorithm has overall better performance comparing it to the Cooley-Tukey version. This happens due to the removal of the data reordering process of the bit reversal done in the Cooley-Tukey version only on the first stage. Effectively, this change improves the results consistently within the test size ranges, however, it is worth noting that the Stockham algorithm has worst data access locality than the Cooley-Tukey algorithm since it accesses data arbitrarily within the FFT instead of performing the sorting right away, therefore these results may not hold this way for larger sizes.

Figure 12: Frame time difference between using stage per pass approach and unique pass for the Radix-2 Cooley-Tukey algorithm

Even more important, the results for the radix-4 version of the Stockham algorithm outperform radix-2 as demonstrated in Figure 14, it drastically improved the performance halfway close to the cuFFT.

By choosing a radix-4 approach the number of stages reduces to half but with a lot more complex operations per dragonfly on each stage, although the work complexity remains the same, there are much less barrier synchronization events.

For each stage there's a synchronization barrier for each local thread inside a work group, so less stages means less sync points, hence finishing the work earlier.

6.3 IMPLEMENTATION ANALYSIS IN CUDA

img/results/cufft_vs_glsl_radix2.png

Figure 13: 2D FFT benchmarks in milliseconds of out-of-place cuFFT and GLSL Radix-2 algorithms

img/results/cufft_vs_glsl_radix4.png

Figure 14: 2D FFT benchmarks of in milliseconds out-of-place cuFFT and GLSL Radix-4 Stockham algorithm

img/results/cuda_vs_glsl_radix2.png

Figure 15: 2D FFT benchmarks in milliseconds of CUDA and GLSL Radix-2 algorithms

img/results/cuda_vs_glsl_radix4.png

Figure 16: 2D FFT benchmarks in milliseconds of CUDA and GLSL Radix-4 Stockham algorithm

7

CONCLUSIONS AND FUTURE WORK

BIBLIOGRAPHY

Leo Bluestein. A linear filtering approach to the computation of discrete fourier transform. *IEEE Transactions on Audio and Electroacoustics*, 18(4):451–455, 1970.

E Oran Brigham. *The fast Fourier transform and its applications*. Prentice-Hall, Inc., 1988.

Eleanor Chu and Alan George. *Inside the FFT black box: serial and parallel fast Fourier transform algorithms*. CRC press, 1999.

Randima Fernando et al. *GPU gems: programming techniques, tips, and tricks for real-time graphics*, volume 590. Addison-Wesley Reading, 2004.

Fynn-Jorin Flügge. Realtime gpgpu fft ocean water simulation. Technical report, 2017.

Matteo Frigo and Steven G Johnson. Fftw: Fastest fourier transform in the west. *Astrophysics Source Code Library*, pages ascl–1201, 2012.

Naga K Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete fourier transforms on graphics processors. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12. Ieee, 2008.

Stijn Heldens, Pieter Hijma, Ben Van Werkhoven, Jason Maassen, and Rob V Van Nieuwpoort. Lightning: Scaling the gpu programming model beyond a single gpu. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 492–503. IEEE, 2022.

Roger A Horn and Charles R Johnson. *Matrix analysis*. Cambridge university press, 2012.

Waqar Hussain, Fabio Garzia, and Jari Nurmi. Evaluation of radix-2 and radix-4 fft processing on a reconfigurable platform. In *13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, pages 249–254. IEEE, 2010.

Yan-Bin Jia. Polynomial multiplication and fast fourier transform. *Com S*, 477:577, 2014.

Douglas L Jones. Digital signal processing: A user's guide. 2014.

Pere Marti-Puig and Ramon Reig Bolano. Radix-4 fft algorithms with ordered input and output data. In *2009 16th International Conference on Digital Signal Processing*, pages 1–6. IEEE, 2009.

NTiAudio. Fast Fourier Transformation FFT - Basics. `https://www.nti-audio.com/en/support/know-how/fast-fourier-transform-fft`. Accessed at 2022-01-28.

NVIDIA. Cuda c++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#scalable-programming-model. Accessed: 2022-07-27.

CUDA Nvidia. Cufft library (2018). *URL developer. nvidia. com/cuFFT*.

OK Ojisan (Takuya OKAHISA). Introduction to the stockham fft. http://wwwa.pikara.ne.jp/okojisan/otfft-en/stockham1.html. Accessed: 2022-07-22.

J Prado. A new fast bit-reversal permutation algorithm based on a symmetry. *IEEE Signal Processing Letters*, 11 (12):933–936, 2004.

Charles M Rader. Discrete fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56(6):1107–1108, 1968.

Kamisetty Ramam Rao and Patrick C Yip. *The transform and data compression handbook*. CRC press, 2018.

RC Singleton. An algorithm for computing the mixed radix fast fourier transform. *IEEE Transactions on audio and electroacoustics*, 17(2):93–103, 1969.

Julius Orion Smith. *Mathematics of the discrete Fourier transform (DFT): with audio applications*. Julius Smith, 2007.

R Yavne. An economical method for calculating the discrete fourier transform. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 115–125, 1968.

Part I

APPENDICES

A

GLSL FFT

```
#version 440

#define M_PI 3.1415926535897932384626433832795

layout (local_size_x = 4, local_size_y = 8) in;

layout (binding = 0, rg32f) uniform image2D pingpong0;
layout (binding = 0, rg32f) uniform image2D pingpong1;

uniform int pingpong;
uniform int log_width;
uniform int stage;
uniform int fft_dir;

vec2 complex_mult(vec2 v0, vec2 v1) {
 return vec2(v0.x * v1.x - v0.y * v1.y,
       v0.x * v1.y + v0.y * v1.x);
}

int bit_reverse(int k) {
    uint br = bitfieldReverse(k);
    return int(bitfieldExtract(br, 32 - log_width, log_width));
}

vec2 euler(float angle) {
 return vec2(cos(angle), sin(angle));
}

void main() {
 int line = int(gl_GlobalInvocationID.x);
 int column = int(gl_GlobalInvocationID.y);

 int group_size = 2 << stage;
 int shift = 1 << stage;
```

```
  vec2 a, b;

     int idx = (line % shift) + group_size * (line / shift);
     vec2 w = euler(fft_dir * 2 * (M_PI / group_size) * ((idx % group_size) %
    shift));

     if (pingpong == 0) {
         if (stage == 0) {
             a = imageLoad(pingpong0, ivec2(bit_reverse(idx), column)).rg;
             b = imageLoad(pingpong0, ivec2(bit_reverse(idx + shift), column)).rg
    ;
         }
         else {
             a = imageLoad(pingpong0, ivec2(idx, column)).rg;
             b = imageLoad(pingpong0, ivec2(idx + shift, column)).rg;
         }

         vec2 raux = a + complex_mult(w, b);
         imageStore(pingpong1, ivec2(idx, column), vec4(raux, 0, 0));

         raux = a - complex_mult(w, b);
         imageStore(pingpong1, ivec2(idx + shift, column), vec4(raux, 0, 0));
     }
     else {
         a = imageLoad(pingpong1, ivec2(idx, column)).rg;
         b = imageLoad(pingpong1, ivec2(idx + shift, column)).rg;

         vec2 raux = a + complex_mult(w, b);
         imageStore(pingpong0, ivec2(idx, column), vec4(raux,0,0));

         raux = a - complex_mult(w, b);
         imageStore(pingpong0, ivec2(idx + shift, column), vec4(raux,0,0));
     }
 }
```

Listing A.1: FFT Cooley-Tukey Horizontal stage pass, see subsection 5.3.1

```
#version 440

#define M_PI 3.1415926535897932384626433832795

layout (local_size_x = 8, local_size_y = 4) in;

layout (binding = 0, rg32f) uniform image2D pingpong0;
layout (binding = 1, rg32f) uniform image2D pingpong1;

uniform int pingpong;
uniform int log_width;
uniform int stage;
uniform int fft_dir;

int iter = 1 << log_width;
int shift = (1 << stage);

vec2 complex_mult(vec2 v0, vec2 v1) {
  return vec2(v0.x * v1.x - v0.y * v1.y,
        v0.x * v1.y + v0.y * v1.x);
}

int bit_reverse(int k) {
    uint br = bitfieldReverse(k);
    return int(bitfieldExtract(br, 32 - log_width, log_width));
}

vec2 euler(float angle) {
  return vec2(cos(angle), sin(angle));
}

void main() {
  int line = int(gl_GlobalInvocationID.x);
  int column = int(gl_GlobalInvocationID.y);

  int group_size = 2 << stage;
  int shift = 1 << stage;

  vec2 a, b;

    int idx = (column % shift) + group_size * (column / shift);
    vec2 w = euler(fft_dir * 2 * (M_PI / group_size) * ((idx % group_size) %
    shift));

    float mult_factor = 1.0;
    if ((stage == log_width - 1) && fft_dir == 1) {
        mult_factor = 1.0 / (iter*iter) ;
    }

    if (pingpong == 0) {
        if (stage == 0) {
            a = imageLoad(pingpong0, ivec2(line, bit_reverse(idx))).rg;
            b = imageLoad(pingpong0, ivec2(line, bit_reverse(idx + shift))).rg;
        }
        else {
            a = imageLoad(pingpong0, ivec2(line, idx)).rg;
            b = imageLoad(pingpong0, ivec2(line, idx + shift)).rg;
        }

        vec2 raux = (a + complex_mult(w, b)) * mult_factor;
        imageStore(pingpong1, ivec2(line, idx), vec4(raux,0,0));

        raux = (a - complex_mult(w, b)) * mult_factor;
        imageStore(pingpong1, ivec2(line, idx + shift), vec4(raux,0,0));
    }
    else {
        if (stage == 0) {
            a = imageLoad(pingpong1, ivec2(line, bit_reverse(idx))).rg;
            b = imageLoad(pingpong1, ivec2(line, bit_reverse(idx + shift))).rg;
        }
        else {
            a = imageLoad(pingpong1, ivec2(line, idx)).rg;
            b = imageLoad(pingpong1, ivec2(line, idx + shift)).rg;
        }

        vec2 raux = (a + complex_mult(w, b)) * mult_factor;
        imageStore(pingpong0, ivec2(line, idx), vec4(raux,0,0));

        raux = (a - complex_mult(w, b)) * mult_factor;
        imageStore(pingpong0, ivec2(line, idx + shift), vec4(raux,0,0));
    }
}
```

Listing A.2: FFT Cooley-Tukey Vertical stage pass, see subsection 5.3.1

B

CUFFT

```
#include <cstdio>
#include <cufft.h>
#include <cuda.h>

#define FFT_SIZE 2048

#define CU_ERR_CHECK_MSG(err, msg) {        \
    if(err != cudaSuccess) {                \
        fprintf(stderr, msg);               \
        exit(1);                            \
    }                                       \
}

#define CU_CHECK_MSG(res, msg) {            \
    if(res != CUFFT_SUCCESS) {              \
        fprintf(stderr, msg);               \
        exit(1);                            \
    }                                       \
}

int main() {
    const size_t data_size = sizeof(cufftComplex)*FFT_SIZE*FFT_SIZE;
    cufftComplex* data = reinterpret_cast<cufftComplex*>(malloc(data_size));
    cufftComplex* gpu_data_in;
    cufftComplex* gpu_data_out;
    cudaError_t err;

    // Initializing input sequence
    for(size_t i = 0; i < FFT_SIZE*FFT_SIZE; ++i) {
        data[i].x = i;
        data[i].y = 0.00;
    }

    // Allocate Input GPU buffer
    err = cudaMalloc(&gpu_data_in, data_size);
    CU_ERR_CHECK_MSG(err, "Cuda error: Failed to allocate\n");

    // Allocate Output GPU buffer
    err = cudaMalloc(&gpu_data_out, data_size);
    CU_ERR_CHECK_MSG(err, "Cuda error: Failed to allocate\n");

    // Copy data to GPU buffer
    err = cudaMemcpy(gpu_data_in, data, data_size, cudaMemcpyHostToDevice);
    CU_ERR_CHECK_MSG(err, "Cuda error: Failed to copy buffer to GPU\n");

    // Setup cufft plan
    cufftHandle plan;
    cufftResult_t res;
    res = cufftPlan2d(&plan, FFT_SIZE, FFT_SIZE, CUFFT_C2C);
    CU_CHECK_MSG(res, "cuFFT error: Plan creation failed\n");

    // Execute Forward 2D FFT
    res = cufftExecC2C(plan, gpu_data_in, gpu_data_out, CUFFT_FORWARD);
    CU_CHECK_MSG(res, "cuFFT error: ExecC2C Forward failed\n");

    // Await end of execution
    err = cudaDeviceSynchronize();
    CU_ERR_CHECK_MSG(err, "Cuda error: Failed to synchronize\n");

    // Execute Inverse 2D FFT
    res = cufftExecC2C(plan, gpu_data_in, gpu_data_out, CUFFT_FORWARD);
    CU_CHECK_MSG(res, "cuFFT error: ExecC2C Forward failed\n");

    // Await end of execution
    err = cudaDeviceSynchronize();
    CU_ERR_CHECK_MSG(err, "Cuda error: Failed to synchronize\n");

    // Retrieve computed FFT buffer
    err = cudaMemcpy(data, gpu_data_in, data_size, cudaMemcpyDeviceToHost);
    CU_ERR_CHECK_MSG(err, "Cuda error: Failed to copy buffer to GPU\n");

    // Destroy Cuda and cuFFT resources
    cufftDestroy(plan);
    cudaFree(gpu_data_in);

    return 0;
}
```

Listing B.1: cuFFT, see section 6.1