Universidade do Minho
Escola de Engenharia
Departamento de Informática

Jorge Francisco Teixeira Bastos da Mota

High Performance Fourier Transforms on GPUs

December 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

December 2022

ACKNOWLEDGEMENTS

Write your acknowledgements here. Do not forget to mention the projects and grants that you have benefited from while doing your research, if any. Ask your supervisor about the specific textual format to use. (Funding agencies are quite strict about this.)

STATEMENT OF INTEGRITY

ABSTRACT

The Fast Fourier Transform is an algorithm or a family of algorithms indespensable for the computation of the Discrete Fourier Transform. Accordingly, this transforms are the core of many applications in several areas and are required to be computed efficiently in many scenarios.

The continuous evolution of GPUs has increased the popularity of parallelizable algorithm implementations on this type of hardware. Traditionally GPUs were associated to graphics background, however, with the popularization of the compute funcionality of this hardware, most modern GPUs now have this capability, hence, algorithms now are more likely to be implemented in the general purpose compute pipeline of GPUs. As a result, many applications take advantage of compute programming in GPGPU capable frameworks such as GLSL, a high-level shading language recurrently used in the context of computer graphics.

In this dissertation we provide, refine and analyze GPU driven implementations of the family of FFT algorithms in GLSL, with the goal to provide programmers with efficient and simplified compute kernels for this transform, from the classic Cooley-Tukey algorithm to more suitable algorithms for the GPU. Accordingly, we also compare these same algorithms with different GPGPU frameworks with the goal to analyse their significance for different compute APIs. Finally, we demonstrate how all improvements discussed in this dissertation culminate in the performance improvement in a real-time rendering tecnique that heavily depends on this transform, as a case of study.

KEYWORDS FFT, GPGPU, GLSL, cuFFT, analysis, performance, compute.

RESUMO

A Transformada Rápida de Fourier é um algoritmo ou uma família de algoritmos indispensáveis para o cálculo da Transformada Discreta de Fourier. Assim, essas transformadas são o núcleo de muitas aplicações em diversas áreas e precisam ser calculadas de forma eficiente em muitos cenários.

A evolução contínua dos GPUs aumentou a popularidade das implementações de algoritmos paralelizáveis neste tipo de *hardware*. Tradicionalmente, os GPUs eram associadas ao fundo gráfico, no entanto, com a popularização da funcionalidade de *compute* desse hardware, os GPUs mais modernos agora têm essa capacidade, portanto, os algoritmos agora são mais propensos a serem implementados na *compute pipeline* de propósito geral dos GPUs. Como resultado, muitas aplicações aproveitam a programação em *compute* em *frameworks* compatíveis com GPGPU como GLSL, uma linguagem de *shading* de alto nível usada recorrentemente no contexto de computação gráfica.

Nesta dissertação fornecemos, refinamos e analisamos implementações em GPU da família de algoritmos FFT em GLSL, com o objetivo de fornecer aos programadores *compute kernels* eficientes e simplificados para esta transformada, desde o clássico algoritmo de Cooley-Tukey até algoritmos mais adequados para o GPU. Da mesma forma, também comparamos esses mesmos algoritmos com diferentes *frameworks* GPGPU com o objetivo de analisar a sua significância para diferentes APIs de *compute*. Por fim, demonstramos como todas as melhorias discutidas nesta dissertação culminam na melhoria de desempenho em uma técnica de renderização em tempo real que depende fortemente dessa transformação, como caso de estudo.

PALAVRAS-CHAVE FFT, GPGPU, GLSL, cuFFT, análise, performance, compute.

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 real-time 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 **?**, 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 real-time 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.

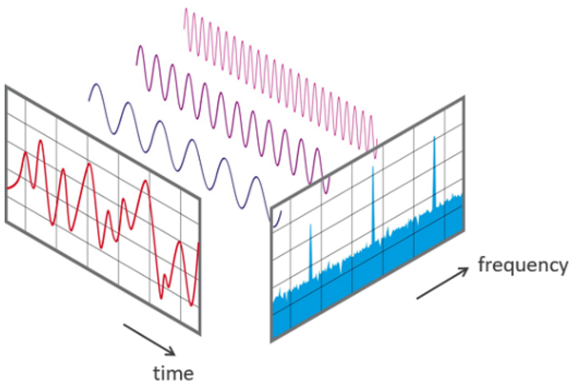


Figure 1: Time to frequency signal decomposition **Source:** NTiAudio

Since 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 is the focus in this dissertation, in Section 2.2. The chapter ends 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 decompose a function into frequency components. 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:

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

where

- $X(f), \forall f \in \mathbb{R} \rightarrow$ 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 $i = \sqrt{-1}$.

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 Euler's formula (Equation 2), we can rewrite the Fourier Transform 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.

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

This model of the Fourier Transform applied to infinite domain functions is called Continuous Fourier Transform.

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 on computers.

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

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

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 expand this formula as:

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

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 $x_n \rightarrow X_k$, 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 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.

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

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

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, since 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 several 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 approaches 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 computation of a DFT of size N into two DFTs of size N/2, 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 $N/2$.

$$X_k = \sum_{n=0}^{N-1} x_n \cdot \omega_N^{kn}$$

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

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

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

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 factor called twiddle 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.

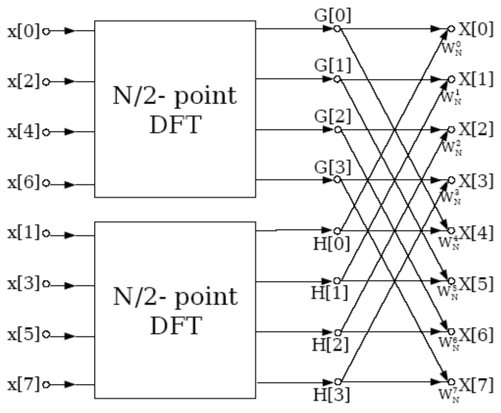


Figure 2: Radix-2 Decimation-in-Time FFT **Source:** Jones (2014)

Effectively, this smaller DFT's are recursively reduced by this algorithm until there's only the computation of a length-2 DFT. On each stage it is applied the butterfly operation with a shifted element according to the substransform size. This butterfly, without any reuse of the twiddle factor, corresponds to:

$$A = a + b \cdot \omega_N^k \tag{9}$$
$$B = a + b \cdot \omega_N^{k+N/2} \tag{10}$$

where $A = x_k$and $B = x_{k+N/2}$

Yet, since the roots of unity for k and $k + N/2$ are always symmetrical, we may re-express this butterfly in terms of the same twiddle factor ω_N^k, such that $\omega_N^{k+N/2}$ will be used as $-\omega_N^k$ (Jones (2014)). With this said, this Cooley-Tukey butterfly (Chu and George (1999)) reuses the same twiddle factor, hence, the reuse of complex multiplications and the FFT computational savings. as 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). 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 $O(N \log(N))$.

The splitting of the DFT into two smaller half sized DFTs causes the original input sequence to require a special reordering to pass the even and odd numbers into, and when this algorithm is applied recursively, this reordering is always needed, so in the end we need a special order for the input elements, fortunately, as noted by **?** this order corresponds to the bit reversed elements of the sequence, therefore, we need to bit reverse all elements of the input sequence.

The bit reversal of the input sequence corresponds to the permutation of swapping the elements position to its bit reversed index, as illustrated in Figure 4. 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 $b_1b_2b_3b_4$, which corresponds to the bit reversed index as $b_4b_3b_2b_1$.

Both the DIT and DIF FFT algorithms require this bit reversal permutation step, but in the case of the DIT we bit reverse the input sequence and on the DIF we apply it on the output, to result in a natural order sequence.

There are many implementations of the bit reversal, and since it is quite simple, any decent version can be used in regards of this FFT algorithm since it is not the main bottleneck. An algorithm such as Algorithm 1 can be used for the $bit\_reverse$ function or any other efficient alternatives (Prado (2004)).

Algorithm 1: Bit reverse

```
Data: Integer $i$
Result: Bit reversed integer $i$
$n \leftarrow 0$ foreach $i = 0$ to $\log(N) - 1$ do
    $n \leftarrow n << 1;$
    $n \leftarrow n|(i\&1);$
    $i \leftarrow i >> 1;$
end
return $i$;
```

In practice, Algorithm 2 demonstrates the aforesaid as an iterative 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-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 11.

$$\begin{aligned} X_{2k} &= \sum_{n=0}^{\frac{N}{2}-1} (x_n + x_{n+\frac{N}{2}}) \cdot \omega_{N/2}^{kn} \\ X_{2k+1} &= \sum_{n=0}^{\frac{N}{2}-1} ((x_n - x_{n+\frac{N}{2}}) \cdot \omega_{N/2}^{kn}) \cdot \omega_N^n \end{aligned} \tag{11}$$

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 5 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.

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. On each stage it is applied the Gentleman-Sande butterfly operation (Chu and George (1999)) with a shifted element according to the subtransform size, as illustrated in Figure 6.

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 6 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 2.

As mentioned in Section 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 3 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 3: 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 $out$;
```

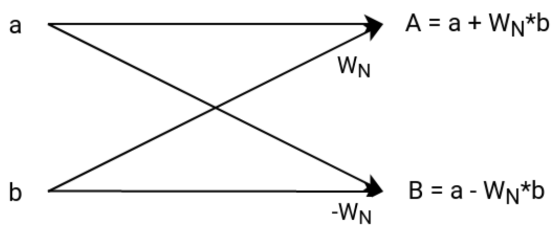


Figure 3: Cooley-Tukey butterfly

img/bit_reverse_permutation.png

Figure 4: Bit reverse permutation

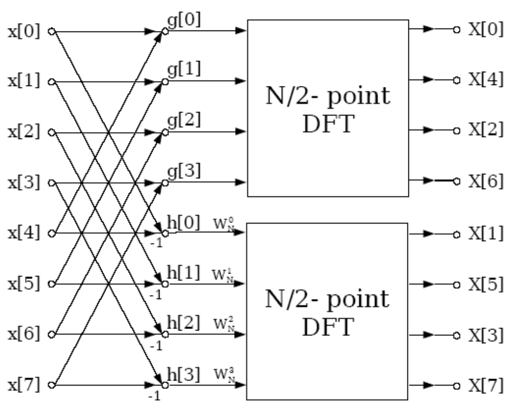


Figure 5: Radix-2 Decimation-in-Frequency FFT **Source:** Jones (2014)

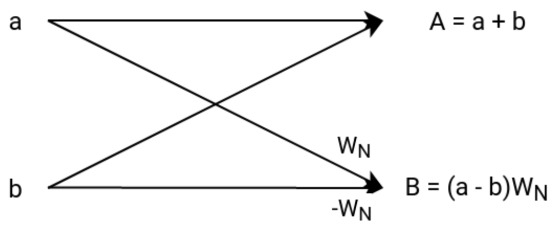


Figure 6: Gentleman-Sande butterfly

3

COMPUTATION OF THE FOURIER TRANSFORM

Nowadays, there is a lot more to the computation of FFT's than just the basic Cooley-Tukey algorithm described in Section 2.3.1. As seen by the large amount of current literature, there are more algorithms, variations and improvements that enhance the computation in many aspects. Choosing the right conditions enhances the calculation of the performance of these primitives, especially for specific hardware (**?**).

One could optimize the FFT in many ways, and in this section we introduce more power of 2 input algorithms such as the Stockham algorithm, a natural order algorithm without explicit bit reversal permutation in Section 3.1. Furthermore, after presenting the radix-2 version of this algorithm, in Section 3.2 we expand the application of this algorithm for higher radix such as radix-4 and consequently reflect on its advantages and disadvantages.

3.1 STOCKHAM ALGORITHM

Despite existing already existing fast solutions for index bit reversal (Prado (2004)), this *shuffle* step still weights the algorithms with extra overhead. The Stockham algorithm is an auto sort algorithm that eliminates the need to have the bit reversal permutation to output a natural order result. It does this by taking advantage of a reordering of the elements (Govindaraju et al. (2008)) in between stages, as illustrated in Figure 7.

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 a GPU programmable language.

When the butterflies are performed, the even and odd elements are composed in such a way that the elements will be in natural order. This composition follows the indexing scheme described in Equation 12.

$$x[q + 2 * p] = y[q + p] \tag{12}$$

$$x[q + 2 * p + 1] = y[q + p + m] \tag{13}$$

Where x and y are alternated sequences for read and 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 m corresponds to the size of the sub FFT divided by 2.

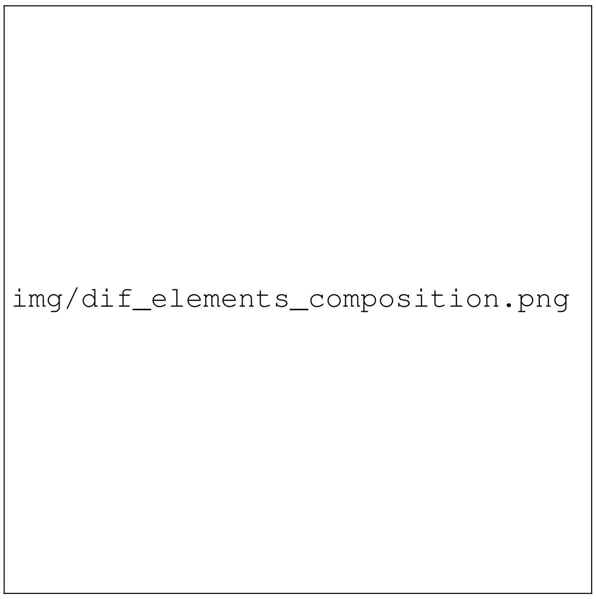


Figure 7: Chain of even and odd compositions over each stage for a natural order DIF

This algorithm requires the use of two alternating sequences for reading and writing the elements for each stage since there is the possibility of reading from a position where it has already been written. That said, it is ensured that at each stage no values are read to which the result of a butterfly has already been saved. Finally, as a consequence of using these alternate sequences, the final result will be in the last sequence we wrote in the last stage, hence, we can deduce which sequence the result is based on the number of stages $\log{(N)}$.

This algorithm is described in Algorithm 4. However, this version may seem visually different from the Cooley Tukey, it preserves the algorithms logic and gets rid of the bit reversal permutation. One main consequence of this algorithm is the requirement of additional space complexity for the alternated read and write access for each stage.

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

```
Data: Sequence pingpong0 with size $N$ power of 2
Result: Sequence $out$ with size $N$ with the DFT of the input
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 $stride$;
        $w_p = \exp(-2 * \pi * i * p / gs)$;
        if $stage\ mod\ 2 == 0$ then
            $a \leftarrow pingpong0[q + s * (p + 0)]$;
            $b \leftarrow pingpong0[q + s * (p + gs/2)]$;
            /* Perform butterfly                                  */
            $pingpong1[q + s * (2 * p + 0)] = a + b$;
            $pingpong1[q + s * (2 * p + 1)] = (a - b) * w_p$;
        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;
else
    return pingpong0;
end
```

This algorithm description will give us a solid code base to use as reference when implementing it on the GPU, mainly due to the way we are indexing and the usage of alternating read write sequences.

3.2 RADIX-4 INSTEAD OF RADIX-2

We've discussed about multiple radix-2 approaches, however, we can explore a wide range of alternatives when we get into higher radices other than just radix-2. These FFT algorithms can use higher radix for better performance and even mixed radix Singleton (1969) for wider range of input sequence sizes.

Rearranging the Stockham algorithm to radix-4 upgrades the computation of the butterflies while reducing the number of stages which will be crucial in later sections. The radix-4 performs the work of two radix-2 iterations with less memory accesses (**?**).

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 more complexity in the butterflies which are sometimes called dragonflies. Additionally, we can use less multiplications with better factorizations (Marti-Puig and Bolano (2009)).

The radix-4 Stockham splits the FFT of size N into four subtransforms of size $N/4$ each stage, therefore this algorithm only features $log(N)/2$ stages and requires the size to be power of 4. Since this is a natural order algorithm the computation of the dragonfly includes the reordering of the elements in natural order every stage, as illustrated in Figure 8.

tese/img/radix4_stockham.png

Figure 8: Illustration of a stage in radix-4 Stockham with each color representing a radix-4 butterfly computation

The forward dragonfly for this algorithm is presented in Figure 9 and its computation involves 4 elements.

With each stage the dragonflies are calculated and the elements are reordered around, therefore, in the end we get Algorithm 5 and its inverse Algorithm 6.

Figure 9: Radix-4 FFT butterfly structure **Source:** Marti-Puig and Bolano (2009)

Algorithm 5: Stockham Radix-4 Decimation-in-Frequency Forward FFT

```
Data: Sequence pingpong0 with size $N$ power of 4
Result: Sequence $out$ with size $N$ with the DFT of the input
foreach $stage = 0$ to $\log(N)/2 - 1$ do
    $n \leftarrow 1 \ll (((\log(N)/2) - stage) * 2)$;
    $s \leftarrow 1 \ll (stage * 2)$;
    $n0 \leftarrow 0$;
    $n1 \leftarrow n/4$;
    $n2 \leftarrow n/2$;
    $n3 \leftarrow n1 + n2$;
    foreach $i = 0$ to $N - 1$ do
        $p \leftarrow i$ div $s$;
        $q \leftarrow i$ mod $s$;
        $w_{1p} = \exp(-2 * \pi * i * p/n)$;
        $w_{2p} = w_{1p} * w_{1p}$;
        $w_{3p} = w_{1p} * w_{2p}$;
        if $stage \bmod 2 == 0$ then
            $a \leftarrow pingpong0[q + s * (p + n0))]$;
            $b \leftarrow pingpong0[q + s * (p + n1)]$;
            $c \leftarrow pingpong0[q + s * (p + n2)]$;
            $d \leftarrow pingpong0[q + s * (p + n3)]$;
            /* Perform dragonfly                                   */
            $pingpong1[q + s * (4 * p + 0)] = a + c + b + d$;
            $pingpong1[q + s * (4 * p + 1)] = w_{1p} * ((a - c) - (b - d) * \sqrt{-1})$;
            $pingpong1[q + s * (4 * p + 2)] = w_{2p} * ((a + c) - (b + d))$;
            $pingpong1[q + s * (4 * p + 3)] = w_{3p} * ((a - c) + (b - d) * \sqrt{-1})$;
        else
```

Algorithm 6: Stockham Radix-4 Decimation-in-Time Inverse FFT

```
Data: Sequence pingpong0 with size $N$ power of 4
Result: Sequence $out$ with size $N$ with the DFT of the input
foreach $stage = 0$ to $\log(N)/2 - 1$ do
    $n \leftarrow 1 \ll (((\log(N)/2) - stage) * 2)$;
    $s \leftarrow 1 \ll (stage * 2)$;
    $n0 \leftarrow 0$;
    $n1 \leftarrow n/4$;
    $n2 \leftarrow n/2$;
    $n3 \leftarrow n1 + n2$;
    foreach $i = 0$ to $N - 1$ do
        $p \leftarrow i$ div $s$;
        $q \leftarrow i$ mod $s$;
        $w_{1p} = \exp(2\pi * i * p/n)$;
        $w_{2p} = w_{1p} * w_{1p}$;
        $w_{3p} = w_{1p} * w_{2p}$;
        if $stage\ mod\ 2 == 0$ then
            $a \leftarrow pingpong0[q + s * (p + n0))]$;
            $b \leftarrow pingpong0[q + s * (p + n1)]$;
            $c \leftarrow pingpong0[q + s * (p + n2)]$;
            $d \leftarrow pingpong0[q + s * (p + n3)]$;
            /* Perform dragonfly                                  */
            $pingpong1[q + s * (4 * p + 0)] = a + c + b + d$;
            $pingpong1[q + s * (4 * p + 1)] = w_{1p} * ((a - c) + (b - d) * \sqrt{-1})$;
            $pingpong1[q + s * (4 * p + 2)] = w_{2p} * ((a + c) - (b + d))$;
            $pingpong1[q + s * (4 * p + 3)] = w_{3p} * ((a - c) - (b - d) * \sqrt{-1})$;
        else
            $a \leftarrow pingpong1[q + s * (p + n0))]$;
            $b \leftarrow pingpong1[q + s * (p + n1)]$;
            $c \leftarrow pingpong1[q + s * (p + n2)]$;
            $d \leftarrow pingpong1[q + s * (p + n3)]$;
            /* Perform dragonfly                                  */
            $pingpong0[q + s * (4 * p + 0)] = a + c + b + d$;
            $pingpong0[q + s * (4 * p + 1)] = w_{1p} * ((a - c) - (b - d) * \sqrt{-1})$;
            $pingpong0[q + s * (4 * p + 2)] = w_{2p} * ((a + c) - (b + d))$;
            $pingpong0[q + s * (4 * p + 3)] = w_{3p} * ((a - c) + (b - d) * \sqrt{-1})$;
        end
    end
end
if $\log(N)\ mod\ 2 == 0$ then
    return pingpong1;
else
    return pingpong0;
end
```

3.3 TWO REAL INPUTS WITHIN ONE COMPLEX INPUT

It is possible that in the case of an application there is a requirement to compute multiple FFTs at once. Although we may simply invoke another FFT computation, we can optimize this step by computing multiple FFTs in the context of the same transform.

We can take advantage of the complex input of our implementation to encode multiple real values to avoid extra explicit FFTs. Since the input has elements in a complex format, but the values are real, we only use the real component of the element, therefore, the output of the inverse also contains the complex with only the real component, as illustrated in Figure 10.

Based on this, we can reuse the imaginary part of the complex with a meaningful value other than 0, as illustrated in Figure 11.

The result of the forward pass will be a mixed frequency value for the two original real values. Additionally, the frequency domain of both input sequences can be extracted from the mixed frequency value by performing some calculations, as demonstrated by **?**. These calculations exploit the symmetry property of the frequency domain when the input sequence fills the real part of the complex and when it fills the imaginary part. With this said we may extract each FFT sequence from the mixed transform with the presented pack and unpack formulas in Equation 14 and Equation 15.

$$Y(k) = X_1(k) + X_2(k) \tag{14}$$

$$\begin{aligned} X_1(k) &= \frac{Y(k) + (Y_r(N-k) - Y_i(N-k))}{2} \\ X_2(k) &= \frac{Y(k) - (Y_r(N-k) - Y_i(N-k))}{2} \end{aligned} \tag{15}$$

- $X_1(k)$, is the k'th complex valued element of the Fourier transform of a sequence with real numbers;
- $X_2(k)$, is the k'th complex valued element of the transform of a sequence with imaginary numbers;
- $Y(k)$, is the mixed frequency transform.
- $Y_r(N-k)$, is the real component of an element of the mixed frequency transform at position $N-k$;
- $Y_r(N-k)$, is the imaginary component of an element of the mixed frequency transform at position $N-k$;

tese/img/results/fft_1_real_complex.png

Figure 10: FFT for an element without an imaginary part

tese/img/results/fft_2_real_complex.png

Figure 11: FFT for an element with an imaginary part

4

IMPLEMENTATION ON THE GPU

The previous chapters introduced and expanded upon FFT algorithms to compute the DFT, which provides enough background to back up the GPU implementations presented in this chapter.

As a result, in this chapter we apply this background and implement these algorithms in GLSL compute shaders as a two-dimensional transform. For this reason, we first describe how the 2D FFT is applied in Section 4.1. Furthermore, in Section 4.2 is described in detail how we implemented and improved the algorithms to run on the GPU.

4.1 2D FOURIER TRANSFORM

Up until now, we only described transforming one-dimensional sequences, however, 2D FFTs are not that different from applying multiple 1D FFTs. Calculating the forward 2D FFT of a two-dimensional sequence produces the frequency domain result. For images, this frequency domain result corresponds to the change of pixel intensities in the original image (**?**).

Calculating a 2D Fourier Transform requires a two-dimensional input sequence that the Forward 2D FFT converts numerical elements from real to complex domain and complex to real domain on its Inverse.

When dealing with images, we might have multiple values per pixel element, we may use it as a greyscale image if we derive the relative 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.

The 2D FFT is computed by performing single dimension FFTs horizontally and vertically, in our case we implemented this by first performing the 1D FFTs for every row and then every column, that is, 1 2D FFT of size MxM is equivalent to $2 * M$ 1D FFT of size M. This is called the row–column decomposition (**?**). In later sections, we refer to this decomposition in the implementation as the horizontal and vertical pass, with the vertical pass being executed after the horizontal.

In Nau3D, we implemented this in two separate compute shader passes, the horizontal pass and the vertical pass, as illustrated in Figure 12.

tese/img/2d_fft.png

Figure 12: High-level illustration of horizontal and vertical passes

Additionally, the implementation of the 1D FFTs for every row and column is completely independent, hence the is freedom to choose any algorithm. In the next section we implement multiple FFT algorithms while reusing the same architecture for the horizontal and vertical passes.

4.2 GLSL IMPLEMENTATION

The implementations were made using GLSL, a high-level shader language for graphics APIs such as OpenGL, and we 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 iterative process that required researching and testing to compose an optimal solution. One of the main targets was to keep the code generalized so that it could be used as the base for other implementations using FFTs with ease.

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 (Section 4.2.1) then progressively improving to the Stockham algorithm (Section 4.2.2 and Section 4.2.3), all this while implementing good GPGPU programming strategies.

4.2.1 *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 4.1: 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 on 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 4.2: Complex multiplication

Due to the adoption of a different programming paradigm the FFT segment iteration loop doesn't exist such as in Algorithm 2, 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 4.3: 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 4.4: 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 4.5: 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.

The ideal solution here is to port this implementation synchronization step to be kernel-wise and this detail changes 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 are functions that make use of barriers that synchronize all the threads within a work group. However, there is a lot of literature on behalf of GPU compute execution synchronization 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 13.

`tese/img/inv_space_2d.png` `tese/img/inv_space_1d.png`

Figure 13: 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. This constraint may seem like a penalty on the size of the work group if the FFT size is very high, but this isn't the case for the sizes we tested (see **??** in Section 5.2), the GPU is an extremely parallel hardware, therefore, its better to dispatch smaller programs with more instances than overloading local threads in a work group to decrease its.

When fitting all stages in one kernel, the algorithm stays the same but we provide all information needed, such as the size and log size of the FFT for the compiler to unroll the loop, since the for loop feature in GLSL requires to be used with constant expressions that allow the loop to be in lined in the compiled code.

```
#define FFT_SIZE 256
#define LOG_SIZE 8

layout (local_size_x = FFT_SIZE/2, local_size_y = 1) in;

layout (binding = 0, rg32f) uniform image2D pingpong0;
layout (binding = 1, rg32f) uniform image2D pingpong1;
uniform int fft_dir;
// ... Auxiliar functions

void main() {
    // ...
    int pingpong = 0;

    for(int stage = 0; stage < LOG_SIZE; ++stage) {
        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));
        // ... Perform butterflies

        pingpong = (pingpong + 1) % 2;
        barrier();
    }
}
```

Listing 4.6: Unique pass structure for Cooley-Tukey

Aside from the variable `fft_dir` that allows the re usage of this pass for the inverse, all needed data for the FFT lives inside the code, hence, there are more opportunities for optimization when the kernel is compiled since most data is within the code itself. For full code for horizontal and vertical passes see Appendix A.

4.2.2 *Radix-2 Stockham*

Since we wanted to get rid of the bit reversal permutation in the shader code, we implemented the Stockham algorithm (see Section 3.1). What differs from the previous code is mainly the reordering of the elements when writing the results of the butterfly. It is also important to note that this algorithm is in DIF instead of DIT like the previous ones.

When performing the butterfly in Listing 4.7 we read the elements from the image according to the thread identifier, however we store it in such way that the output has its elements in natural order (see Algorithm 4).

```
for(int stage = 0; stage < LOG_SIZE; ++stage) {
    int n = 1 << (LOG_SIZE - stage); // group_size
    int m = n >> 1; // shift
    int s = 1 << stage;

    int p = line / s;
    int q = line % s;
    vec2 wp = euler(fft_dir * 2 * (M_PI / n) * p);

    if(pingpong == 0) {
        vec2 a = imageLoad(pingpong0, ivec2(q + s*(p + 0), column)).rg;
        vec2 b = imageLoad(pingpong0, ivec2(q + s*(p + m), column)).rg;

        vec2 res = (a + b);
        imageStore(pingpong1, ivec2(q + s*(2*p + 0), column), vec4(res,0,0));
        res = complex_mult(wp,(a - b));
        imageStore(pingpong1, ivec2(q + s*(2*p + 1), column), vec4(res,0,0));
    }
    else {
        // ... Read pingpong1, write pingpong0
    }

    // ... Sync
}
```

Listing 4.7: Radix-2 Stockham DIF

Consequently, we got rid of the conditional read with bit reversed index, and the compute shader code for the horizontal Listing A.5 and vertical passes Listing A.6 got simpler as a result of this.

4.2.3 *Radix-4 Stockham*

In Section 3.2 we introduced how higher radix factorizations could improve the performance with the cost of size constraints, hence, here we change the code with ease to implement the radix-4 factorization described previously.

First we update the stage control variables and compute the multiple twiddle factors used in the radix-4 butterfly. Note that now the for loop only iterates $\log(N)/2$ times and the number of local threads is reduced by half.

```
#define FFT_SIZE 256
#define LOG_SIZE 8 // log2(FFT_SIZE)
#define HALF_LOG_SIZE 4 // log2(FFT_SIZE) / 2

layout (local_size_x = FFT_SIZE/4, local_size_y = 1) in;

// ...

void main() {
    // ...

    for(int stage = 0; stage < HALF_LOG_SIZE; ++stage) {
        int n = 1 << (HALF_LOG_SIZE - stage)*2;
        int s = 1 << stage*2;

        int n0 = 0;
        int n1 = n/4;
        int n2 = n/2;
        int n3 = n1 + n2;

        int p = line / s;
        int q = line % s;

        vec2 w1p = euler(2*(M_PI / n) * p * fft_dir);
        vec2 w2p = complex_mult(w1p,w1p);
        vec2 w3p = complex_mult(w1p,w2p);

        // ... Radix-4 butterfly
    }
}
```

Listing 4.8: Radix-4 Stockham stage control variables

Then, we compute the radix-4 butterfly and store the results in natural order, as described in Figure 9.

```
if(pingpong == 0) {
    vec2 a = imageLoad(pingpong0, ivec2(q + s*(p + n0), column)).rg;
    vec2 b = imageLoad(pingpong0, ivec2(q + s*(p + n1), column)).rg;
    vec2 c = imageLoad(pingpong0, ivec2(q + s*(p + n2), column)).rg;
    vec2 d = imageLoad(pingpong0, ivec2(q + s*(p + n3), column)).rg;

    vec2 apc = a + c;
    vec2 amc = a - c;
    vec2 bpd = b + d;
    vec2 jbmd = complex_mult(vec2(0,1), b - d);

    imageStore(pingpong1, ivec2(q + s*(4*p + 0), column), vec4(apc + bpd, 0,0));
    imageStore(pingpong1, ivec2(q + s*(4*p + 1), column), vec4(complex_mult(w1p,
    amc - jbmd), 0,0));
    imageStore(pingpong1, ivec2(q + s*(4*p + 2), column), vec4(complex_mult(w2p,
    apc - bpd ), 0,0));
    imageStore(pingpong1, ivec2(q + s*(4*p + 3), column), vec4(complex_mult(w3p,
    amc + jbmd), 0,0));
}
else {
    // ...
}
```

Listing 4.9: Radix-4 Stockham butterfly

We take advantage of branchless programming to flip the signal of the butterfly to avoid unnecessary if statements for the inverse.

```
imageStore(pingpong1, ivec2(q + s*(4*p + 0), column), vec4(apc + bpd, 0,0));
imageStore(pingpong1, ivec2(q + s*(4*p + 1), column), vec4(complex_mult(w1p, amc
    + jbmd*fft_dir), 0,0));
imageStore(pingpong1, ivec2(q + s*(4*p + 2), column), vec4(complex_mult(w2p, apc
    - bpd ), 0,0));
imageStore(pingpong1, ivec2(q + s*(4*p + 3), column), vec4(complex_mult(w3p, amc
    - jbmd*fft_dir), 0,0));
```

Listing 4.10: Radix-4 Stockham dragonfly inverse arithmetic

With these changes, we now have the radix-4 Stockham shader complete (see Listing A.7 and Listing A.8).

Furthermore, we may expand the implementation of this radix-4 compute shader, in such a way that it supports power of 2 sizes instead of just power of 4. For example, when a power of 2 size input is used in the radix-4 Stockham algorithm it performs half of $\log(N)$ stages, meaning that it will miss one final stage for a sub-transform with a size that is not multiple of 4. In this final stage, we may apply the radix-2 Stockham algorithm stage as an additional step to grant support for power of 2 sizes.

With this said, at the end of the compute shader in the radix-4 implementation we add a conditional last step implementing the radix-2 Stockham stage butterflies. Likewise, this stage corresponds to the same code as in

Section 4.2.2, but partially more hardcoded for a last stage. For this reason, the presented code doesn't use a twiddle factor, since it will always correspond to 1 due to parameter p being equal to 0 for the last stage.

Finally, since we only have $FFT\_SIZE/4$ local threads, therefore, 2 butterflies need to be computed at a time, as demonstrated in Listing 4.11.

```
#define FFT_SIZE 512
#define LOG_SIZE 9 // log2(FFT_SIZE)
#define HALF_LOG_SIZE 4 // log2(FFT_SIZE / 2) / 2


int main() {
    // ... radix-4 Stockham stages

    if(LOG_SIZE % 2 == 1) {
        int s = FFT_SIZE >> 1;
        int q = 2*line;

        if(pingpong == 0) {
            vec2 a = imageLoad(pingpong0, ivec2(q + 0, column)).rg;
            vec2 b = imageLoad(pingpong0, ivec2(q + s, column)).rg;
            imageStore(pingpong1, ivec2(q + 0, column), vec4(a + b, 0,0));
            imageStore(pingpong1, ivec2(q + s, column), vec4(a - b, 0,0));

            // There's only SIZE/4 local threads therefore we compute 2 values
            q = 2*line + 1;

            a = imageLoad(pingpong0, ivec2(q + 0, column)).rg;
            b = imageLoad(pingpong0, ivec2(q + s, column)).rg;
            imageStore(pingpong1, ivec2(q + 0, column), vec4(a + b, 0,0));
            imageStore(pingpong1, ivec2(q + s, column), vec4(a - b, 0,0));
        }
        else {
            vec2 a = imageLoad(pingpong1, ivec2(q + 0, column)).rg;
            vec2 b = imageLoad(pingpong1, ivec2(q + s, column)).rg;
            imageStore(pingpong0, ivec2(q + 0, column), vec4(a + b, 0,0));
            imageStore(pingpong0, ivec2(q + s, column), vec4(a - b, 0,0));

            // There's only SIZE/4 local threads therefore we compute 2 values
            q = 2*line + 1;

            a = imageLoad(pingpong1, ivec2(q + 0, column)).rg;
            b = imageLoad(pingpong1, ivec2(q + s, column)).rg;
            imageStore(pingpong0, ivec2(q + 0, column), vec4(a + b, 0,0));
            imageStore(pingpong0, ivec2(q + s, column), vec4(a - b, 0,0));
        }
    }
}
```

Listing 4.11: Radix-2 stage for the radix-4 Stockham code

5 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 tests 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.

5.1 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 efficiently on NVIDIA GPUs.

It has been proven multiple times that cuFFT is one of the fastest available tools for computing FFT such as in **?** and many other resources.

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;
- Power of 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 are presented in Figure 15.

5.2 GLSL IMPLEMENTATION RESULTS

The implementations discussed in Section 4.2, 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.

The benchmark results in this section 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 Section 4.2.1 we discussed how the implementation would benefit by having a unique pass that synchronized by stage instead of dispatching multiple stage passes. Accordingly, to prove this, we evaluated the difference between the Cooley-Tukey algorithm in the unique pass and in a stage-per-pass approach. Hence, in Figure 14 are presented the CPU and GPU results of the benchmarks for this test case. The GPU time corresponds to the total time spent in the GPU executing the compute shader.

These benchmarks were tested in Nau3D, a generic 3D rendering engine, therefore, the CPU side of the stage-per-pass implementation takes into account the execution of a *Lua* script to update the state and control variables at the end of each stage. For this reason, the CPU time besides the preparation of the compute dispatch also includes some overhead for using this *Lua* script.

The results in Figure 14 demonstrate the approximate expected overhead of using the stage-per-pass in an application. Since there is a need for the update of the following stages, the implementation synchronizes with the GPU immediately over each stage, however, the compute shader is reusable, therefore, the same shader module can be used for multiple sizes. In contrast, the CPU time of the unique pass kernel is mostly constant and this approach is highly optimized for its own size so most calculations are inlined by the GLSL compiler. The stage synchronization is kept inside the GPU until the kernel completed the execution. Since this type of synchronization is on the work group level, there are much fewer actors involved in the synchronization step. In addition, these actors only correspond to the 1D FFT region of each row/column and not the entire 2D FFT.

Due to the difference in the performance of these two approaches, the following comparisons will exclusively correspond to unique pass implementations of the investigated algorithms.

As we can see in Figure 15 we plot the performance of the benchmarks of cuFFT and the unique pass algorithms in GLSL. These implementations are the single-pass approaches of the discussed algorithms in Section 4.2. Based on the findings of these benchmarks, the GLSL radix-2 implementation of the Stockham algorithm has an overall better performance comparing it to the Cooley-Tukey version. Additionally, it was also implemented with a smaller kernel. Consequently, this happens due to the removal of the data reordering process of the bit reversal done in the Cooley-Tukey version only in the first stage. Effectively, this change improves the results consistently within the test size ranges.

Undoubtedly, the results that come closer to the cuFFT are the ones of the radix-4 Stockham. It drastically improved the performance halfway close to the cuFFT, as presented in Figure 15. Yet, the complexity of the shader code was sharply increased, not only due to the higher radix alternative but also due to the additional support step in the last stage for the power of 2 sizes.

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. Since this dragonfly represents the operations required in an equivalent radix-2 Stockham factorization. Although the work complexity remains the same, there are much fewer barrier synchronization events and radix-4 iterations perform the work of two radix-2 iterations with only one memory access.

For each stage there's a synchronization barrier in each local thread inside a work group, so fewer stages means fewer sync points, hence, compensating for the 70% kernel size increase of the radix-4 version.

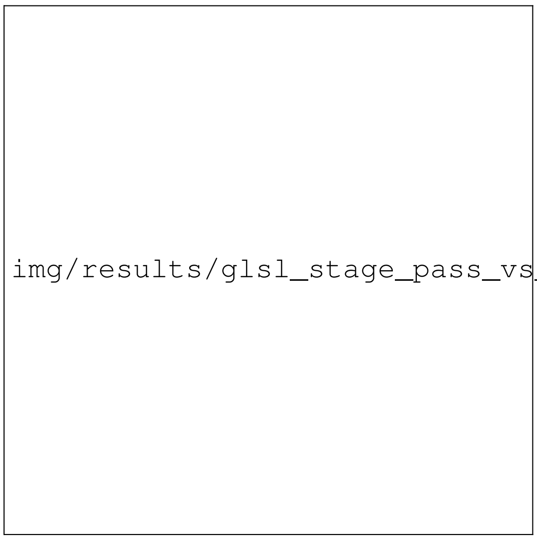
In the context of the same need for multiple FFTs, we reach a point where the we need to batch executions together for cuFFT. While this can also be achieved using our GLSL implementations, we can take a step further on the usage of the same pass for double the FFTs. Accordingly, we adapted the radix-4 Stockham implementation to use `vec4` instead of `vec2` for the elements of the input and output buffers.

Code integrity is conserved since changing to `vec4` does not change operators in the compute kernel.

To compare with an equivalent, the computation in cuFFT used a special setup of a 2D FFT plan to compute with a batch size of 2. The cuFFT framework doesn't have an explicit setup for multiple transforms in the same kernel, hence, we use batched execution for comparison.

In table Table 2, the benchmarks of Figure 16 are presented together with the results of calculating a single transform in one step. To analyze the potential gain that the implementation with twice the transforms gives, we also present the percentage of the performance gain per individual FFT.

As a result, we can clearly notice a larger gain for size 128 inputs, both for cuFFT and for GLSL. From there, the gain for the rest of the tested sizes is notably smaller, the most relevant gain being that of inputs of size 256 for GLSL since cuFFT did not show any difference to twice the time of a single FFT for this input size. Finally,



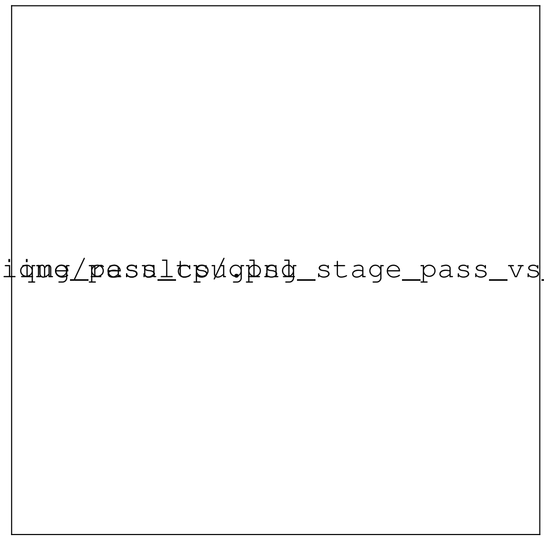


Figure 14: CPU and GPU time for using stage per pass approach and unique pass for the Radix-2 Cooley-Tukey algorithm

Sizes		**128**	**256**	**512**	**1024**
cuFFT	Single FFT	0.035	0.049	0.133	0.560
	Double FFT	0.038	0.098	0.23	1.032
	Gain per single FFT	45.71%	0.0%	13.53%	7.85%
GLSL radix-4 Stockham	Single FFT	0.042	0.087	0.389	1.363
	Double FFT	0.044	0.132	0.704	2.702
	Gain per single FFT	47.61%	24.13%	9.51%	0.88%

Table 2: Benchmarks values of the Forward 2D FFT for cuFFT and GLSL radix-4 Stockham for single and double transforms in the same pass, together with the percentage of gain over individual transforms.

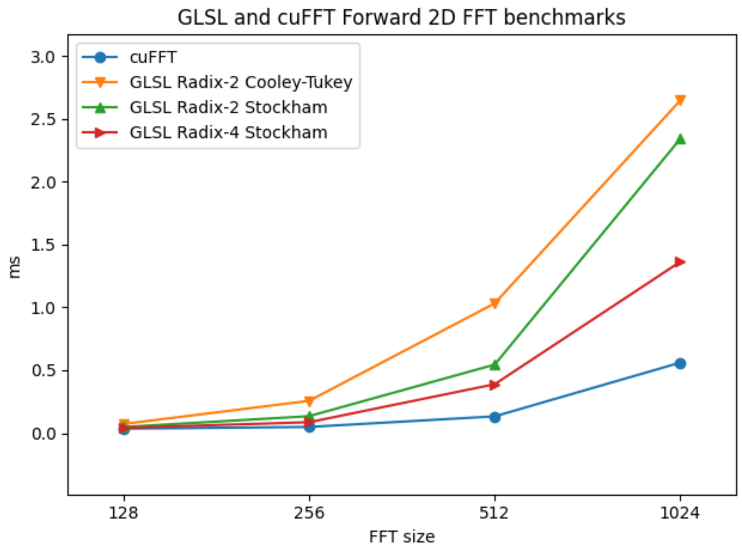


Figure 15: Forward 2D FFT benchmarks in milliseconds of out-of-place cuFFT and unique pass algorithms in GLSL

tese/img/results/cufft_glsl_multiple_fft_benchmarks.png

Figure 16: Forward 2D FFT benchmarks of double the transforms in milliseconds of out-of-place cuFFT and radix-4 Stockham GLSL

for input sizes 512 and 1024 the cuFFT batched execution demonstrated a valuable gain while the GLSL one continued to decline.

Overall, we can conclude that it is worth including multiple FFTs in the same *pass* for the tested sizes when multiple independent transforms are necessary.

With the presented results, we can clearly notice better performance on the GPU regarding the implementation in GLSL of radix-4 Stockham with a unique pass approach that reduces the number of synchronization stages and calculates per dragonfly the equivalent 2 butterflies of two stages of radix-2 Stockham.

5.3 CASE OF STUDY

Based on the findings of Section 5.2 we measured how the GLSL implementations behaved, by analyzing the performance of a test application that converted a 2D image to the frequency domain and then reversed it to its original look. Yet, with the goal of highlighting the importance of the performance increase of these algorithms, in this section, we provide an overview of the impact of the implementation within a more realistic scenario by using an ocean rendering technique demo that heavily relies on the usage of FFT (Figure 17).

In this section we brief the Tensendorf waves demo in Section 5.3.1 where we describe in which way FFTs are relevant for the implementation of this rendering technique, and how we improved an existing implementation for Nau3D that used a pass-per-stage Cooley-Tukey implementation. After that we present results and how the FFT implementation improves the demo performance and by how much in Section 5.3.2.

5.3.1 *Tensendorf waves*

The rendering of the oceans demo we used as a starting point was a real-time implementation in Nau3D of the popular article published by **?**. In this demo there are two main stages, the generation of the height map and the actual rendering, the FFTs come into place in the generation of the height map since we need to generate it and the additional vectors used for shading. In total, there are 4 2D inverse FFTs computed for each frame, which translates to $8 * FFT\_SIZE$ 1D FFTs in total.

Regarding the results in Section 5.2, we first changed the pipeline from a pass-per-stage radix-2 Cooley-Tukey algorithm to implement radix-2 and radix-4 Stockham with synchronization within the kernel for the horizontal and vertical passes. Each pass computes multiple FFTs at a time and we take advantage of the same kernel to compute all required FFTs at the same time. Additionally, it is worth noticing that the inverse FFTs produce from the frequencies components, two usable real values. Finally, use used `vec4` as described in **??** to compute multiple FFTs in the context of the SIMT instructions produced for the GPU.

Although the FFTs take a big role in this demo, it also renders the ocean waves that have around 2 million vertices, hence, the performance does not only depend on the FFTs computation.

For this demo, we tested its performance with sizes 512 and 1024 for the ping pong buffers. We used size 512 to test a balance of performance and wave quality and 1024 to test for better wave quality as suggested by **?**.

The size of 512 allowed us to test for good enough wave quality while maintaining a Similarly to Section 5.2, the radix-4 Stockham implementation for the size 512 integrates a final stage with radix-2 butterflies to be able to support the computation of this size.

5.3.2 *Results*

In Figure 18 we can note the performance difference the radix-2 Stockham gives when performing the horizontal and vertical pass comparing it to the radix-2 Cooley-Tukey with a pass-per-stage. Not only is there an improvement in time and resources used by eliminating CPU steps, but there is also better performance using unique pass versions.

When running the application, the performance increase is noticeable and the frame rate is improved by up to 20%.

Testing the demo for higher quality waves the radix-4 Stockham performance stands out in Figure 19, as predicted. When running the application, the difference in performance is noticeable, with the frame rate using Stockham radix-2 being improved by up to 20%, while the radix-4 delivers a frame rate of up to 60%.

These results proved the necessity of implementing adequate algorithms and appropriate GPGPU programming practices.

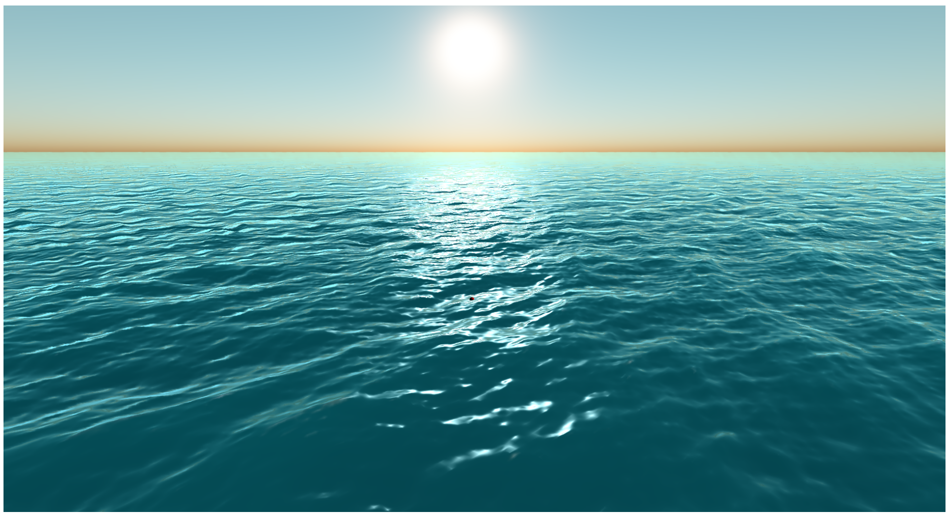


Figure 17: Tensendorf waves in Nau3D Engine

img/results/demo_512_cpu.png

img/results/demo_512_gpu.png

Figure 18: Time spent in the CPU and GPU for the size 512 FFT horizontal and vertical passes

img/results/demo_1024_cpu.png

img/results/demo_1024_gpu.png

Figure 19: Total time spent in the CPU and GPU for the size 1024 FFT horizontal and vertical passes

6

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.

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.

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.

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

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 Radix-2 Cooley-Tukey Horizontal stage pass, see Section 4.2.1

```
#version 440

#define M_PI 3.14159265358979323846264338327950

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 Radix-2 Cooley-Tukey Vertical stage pass, see Section 4.2.1

```
#version 440

#define M_PI 3.1415926535897932384626433832795
#define FFT_SIZE 256
#define LOG_SIZE 8

layout (local_size_x = FFT_SIZE/2, local_size_y = 1) in;

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

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_SIZE, LOG_SIZE));
}

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

void main() {
    int line = int(gl_GlobalInvocationID.x);
    int column = int(gl_WorkGroupID.y);
    int pingpong = 0;

    for(int stage = 0; stage < LOG_SIZE; ++stage) {
        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));

        // alternate between textures
        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));
        }

        pingpong = (pingpong + 1) % 2;
        barrier();
    }
}
```

Listing A.3: FFT Radix-2 Cooley-Tukey Horizontal unique pass, see Section 4.2.1

```
#version 440

#define M_PI 3.1415926535897932384626433832795
#define FFT_SIZE 256
#define LOG_SIZE 8

layout (local_size_x = FFT_SIZE/2, local_size_y = 1) in;

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

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_SIZE, LOG_SIZE));
}

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

void main() {
  int line = int(gl_WorkGroupID.y);
  int column = int(gl_GlobalInvocationID.x);
    int pingpong = LOG_SIZE % 2;

    for(int stage = 0; stage < LOG_SIZE; ++stage) {
        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_SIZE - 1) && fft_dir == 1) {
            mult_factor = 1.0 / (FFT_SIZE*FFT_SIZE);
        }

        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));
        }

        pingpong = ((pingpong + 1) % 2);
        barrier();
    }
}
```

Listing A.4: FFT Radix-2 Cooley-Tukey Vertical unique pass, see Section 4.2.1

```
#version 440

#define M_PI 3.1415926535897932384626433832795
#define FFT_SIZE 256
#define LOG_SIZE 8

layout (local_size_x = (FFT_SIZE/2)/NUM_BUTTERFLIES, local_size_y = 1) in;

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

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);
}

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

void main() {
    int line = int(gl_GlobalInvocationID.x);
    int column = int(gl_WorkGroupID.y);
    int pingpong = 0;

    for(int stage = 0; stage < LOG_SIZE; ++stage) {
        int n = 1 << (LOG_SIZE - stage);
        int m = n >> 1;
        int s = 1 << stage;

        int p = line / s;
        int q = line % s;

        vec2 wp = euler(fft_dir * 2 * (M_PI / n) * p);
        if(pingpong == 0) {
            vec2 a = imageLoad(pingpong0, ivec2(q + s*(p + 0), column)).rg;
            vec2 b = imageLoad(pingpong0, ivec2(q + s*(p + m), column)).rg;

            vec2 res = (a + b);
            imageStore(pingpong1, ivec2(q + s*(2*p + 0), column), vec4(res,0,0));
            res = complex_mult(wp,(a - b));
            imageStore(pingpong1, ivec2(q + s*(2*p + 1), column), vec4(res,0,0));
        }
        else {
            vec2 a = imageLoad(pingpong1, ivec2(q + s*(p + 0), column)).rg;
            vec2 b = imageLoad(pingpong1, ivec2(q + s*(p + m), column)).rg;

            vec2 res = (a + b);
            imageStore(pingpong0, ivec2(q + s*(2*p + 0), column), vec4(res,0,0));
            res = complex_mult(wp,(a - b));
            imageStore(pingpong0, ivec2(q + s*(2*p + 1), column), vec4(res,0,0));
        }

        pingpong = (pingpong + 1) % 2;
        barrier();
    }
}
```

Listing A.5: FFT Radix-2 Stockham Horizontal unique pass, see Section 4.2.2

```
#version 440

#define M_PI 3.1415926535897932384626433832795
#define FFT_SIZE 256
#define LOG_SIZE 8

layout (local_size_x = FFT_SIZE/2, local_size_y = 1) in;

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

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);
}

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

void main() {
  int line = int(gl_WorkGroupID.y);
  int column = int(gl_GlobalInvocationID.x);
    int pingpong = LOG_SIZE % 2;

    for(int stage = 0; stage < LOG_SIZE; ++stage) {
        int n = 1 << (LOG_SIZE - stage);
        int m = n >> 1;
        int s = 1 << stage;

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

        int p = column / s;
        int q = column % s;

        vec2 wp = euler(fft_dir * 2 * (M_PI / n) * p);
        if(pingpong == 0) {
            vec2 a = imageLoad(pingpong0, ivec2(line, q + s*(p + 0))).rg;
            vec2 b = imageLoad(pingpong0, ivec2(line, q + s*(p + m))).rg;

            vec2 res = (a + b) * mult_factor;
            imageStore(pingpong1, ivec2(line, q + s*(2*p + 0)), vec4(res,0,0));
            res = complex_mult(wp,(a - b)) * mult_factor;
            imageStore(pingpong1, ivec2(line, q + s*(2*p + 1)), vec4(res,0,0));
        }
        else {
            vec2 a = imageLoad(pingpong1, ivec2(line, q + s*(p + 0))).rg;
            vec2 b = imageLoad(pingpong1, ivec2(line, q + s*(p + m))).rg;

            vec2 res = (a + b) * mult_factor;
            imageStore(pingpong0, ivec2(line, q + s*(2*p + 0)), vec4(res,0,0));
            res = complex_mult(wp,(a - b)) * mult_factor;
            imageStore(pingpong0, ivec2(line, q + s*(2*p + 1)), vec4(res,0,0));
        }

        pingpong = (pingpong + 1) % 2;
        barrier();
    }
}
```

Listing A.6: FFT Radix-2 Stockham Vertical unique pass, see Section 4.2.2

```
#version 440

#define M_PI 3.1415926535897932384626433832795
#define FFT_SIZE 256
#define LOG_SIZE 8 // log2(FFT_SIZE)
#define HALF_LOG_SIZE 4 // log2(FFT_SIZE) / 2

layout (local_size_x = FFT_SIZE/4, local_size_y = 1) in;
layout (binding = 0, rg32f) uniform image2D pingpong0;
layout (binding = 1, rg32f) uniform image2D pingpong1;

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);
}

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

void main() {
    int line = int(gl_GlobalInvocationID.x);
    int column = int(gl_WorkGroupID.y);
    int pingpong = 0;

    for(int stage = 0; stage < HALF_LOG_SIZE; ++stage) {
        int n = 1 << (HALF_LOG_SIZE - stage)*2;
        int s = 1 << stage*2;

        int n0 = 0;
        int n1 = n/4;
        int n2 = n/2;
        int n3 = n1 + n2;

        int p = line / s;
        int q = line % s;

        vec2 w1p = euler(2*(M_PI / n) * p * fft_dir);
        vec2 w2p = complex_mult(w1p,w1p);
        vec2 w3p = complex_mult(w1p,w2p);

        if(pingpong == 0) {
            vec2 a = imageLoad(pingpong0, ivec2(q + s*(p + n0), column)).rg;
            vec2 b = imageLoad(pingpong0, ivec2(q + s*(p + n1), column)).rg;
            vec2 c = imageLoad(pingpong0, ivec2(q + s*(p + n2), column)).rg;
            vec2 d = imageLoad(pingpong0, ivec2(q + s*(p + n3), column)).rg;

            vec2 apc = a + c;
            vec2 amc = a - c;
            vec2 bpd = b + d;
            vec2 jbmd = complex_mult(vec2(0,1), b - d);
            imageStore(pingpong1, ivec2(q + s*(4*p + 0), column), vec4(apc + bpd,
     0,0));
            imageStore(pingpong1, ivec2(q + s*(4*p + 1), column), vec4(
    complex_mult(w1p, amc + jbmd*fft_dir), 0,0));
            imageStore(pingpong1, ivec2(q + s*(4*p + 2), column), vec4(
    complex_mult(w2p, apc - bpd ), 0,0));
            imageStore(pingpong1, ivec2(q + s*(4*p + 3), column), vec4(
    complex_mult(w3p, amc - jbmd*fft_dir), 0,0));
        }
        else {
            vec2 a = imageLoad(pingpong1, ivec2(q + s*(p + n0), column)).rg;
            vec2 b = imageLoad(pingpong1, ivec2(q + s*(p + n1), column)).rg;
            vec2 c = imageLoad(pingpong1, ivec2(q + s*(p + n2), column)).rg;
            vec2 d = imageLoad(pingpong1, ivec2(q + s*(p + n3), column)).rg;

            vec2 apc = a + c;
            vec2 amc = a - c;
            vec2 bpd = b + d;
            vec2 jbmd = complex_mult(vec2(0,1), b - d);

            imageStore(pingpong0, ivec2(q + s*(4*p + 0), column), vec4(apc + bpd,
     0,0));
            imageStore(pingpong0, ivec2(q + s*(4*p + 1), column), vec4(
    complex_mult(w1p, amc + jbmd*fft_dir), 0,0));
            imageStore(pingpong0, ivec2(q + s*(4*p + 2), column), vec4(
    complex_mult(w2p, apc - bpd ), 0,0));
            imageStore(pingpong0, ivec2(q + s*(4*p + 3), column), vec4(
    complex_mult(w3p, amc - jbmd*fft_dir), 0,0));
        }


        pingpong = (pingpong + 1) % 2;
        barrier();
    }
}
```

Listing A.7: FFT Radix-4 Stockham Horizontal unique pass, see Section 4.2.3

```
#version 440

#define M_PI 3.1415926535897932384626433832795
#define FFT_SIZE 256
#define LOG_SIZE 8 // log2(FFT_SIZE)
#define HALF_LOG_SIZE 4 // log2(FFT_SIZE) / 2

layout (local_size_x = (FFT_SIZE/4)/NUM_BUTTERFLIES, local_size_y = 1) in;
layout (binding = 0, rg32f) uniform image2D pingpong0;
layout (binding = 1, rg32f) uniform image2D pingpong1;

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);
}

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

void main() {
  int line = int(gl_WorkGroupID.y);
  int column = int(gl_GlobalInvocationID.x);
    int pingpong = HALF_LOG_SIZE % 2;

    for(int stage = 0; stage < HALF_LOG_SIZE; ++stage) {
        int group_size = 2 << stage;
        int shift = 1 << stage;

        int n = 1 << ((HALF_LOG_SIZE - stage)*2);
        int s = 1 << (stage*2);

        int n0 = 0;
        int n1 = n/4;
        int n2 = n/2;
        int n3 = n1 + n2;

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

        int p = column / s;
        int q = column % s;

        vec2 w1p = euler(2*(M_PI / n) * p * fft_dir);
        vec2 w2p = complex_mult(w1p,w1p);
        vec2 w3p = complex_mult(w1p,w2p);

        if(pingpong == 0) {
            vec2 a = imageLoad(pingpong0, ivec2(line, q + s*(p + n0))).rg;
            vec2 b = imageLoad(pingpong0, ivec2(line, q + s*(p + n1))).rg;
            vec2 c = imageLoad(pingpong0, ivec2(line, q + s*(p + n2))).rg;
            vec2 d = imageLoad(pingpong0, ivec2(line, q + s*(p + n3))).rg;

            vec2 apc = a + c;
            vec2 amc = a - c;
            vec2 bpd = b + d;
            vec2 jbmd = complex_mult(vec2(0,1), b - d);

            imageStore(pingpong1, ivec2(line, q + s*(4*p + 0)), vec4(mult_factor
    * (apc + bpd), 0,0));
            imageStore(pingpong1, ivec2(line, q + s*(4*p + 1)), vec4(mult_factor
    * (complex_mult(w1p, amc + jbmd*fft_dir)), 0,0));
            imageStore(pingpong1, ivec2(line, q + s*(4*p + 2)), vec4(mult_factor
    * (complex_mult(w2p, apc - bpd )), 0,0));
            imageStore(pingpong1, ivec2(line, q + s*(4*p + 3)), vec4(mult_factor
    * (complex_mult(w3p, amc - jbmd*fft_dir)), 0,0));
        }
        else {
            vec2 a = imageLoad(pingpong1, ivec2(line, q + s*(p + n0))).rg;
            vec2 b = imageLoad(pingpong1, ivec2(line, q + s*(p + n1))).rg;
            vec2 c = imageLoad(pingpong1, ivec2(line, q + s*(p + n2))).rg;
            vec2 d = imageLoad(pingpong1, ivec2(line, q + s*(p + n3))).rg;

            vec2 apc = a + c;
            vec2 amc = a - c;
            vec2 bpd = b + d;
            vec2 jbmd = complex_mult(vec2(0,1), b - d);

            imageStore(pingpong0, ivec2(line, q + s*(4*p + 0)), vec4(mult_factor
    * (apc + bpd), 0,0));
            imageStore(pingpong0, ivec2(line, q + s*(4*p + 1)), vec4(mult_factor
    * (complex_mult(w1p, amc + jbmd*fft_dir)), 0,0));
            imageStore(pingpong0, ivec2(line, q + s*(4*p + 2)), vec4(mult_factor
    * (complex_mult(w2p, apc - bpd )), 0,0));
            imageStore(pingpong0, ivec2(line, q + s*(4*p + 3)), vec4(mult_factor
    * (complex_mult(w3p, amc - jbmd*fft_dir)), 0,0));
        }

        pingpong = (pingpong + 1) % 2;
        barrier();
    }
}
```

Listing A.8: FFT Radix-4 Stockham Vertical unique pass, see Section 4.2.3

B

CUFFT

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

#define FFT_SIZE 256

#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 5.1