Universidade do Minho
Escola de Engenharia
Departamento de Informática

Jorge Francisco Teixeira Bastos da Mota

High Performance Fourier Transforms on GPUs with GLSL

February 2023

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Jorge Francisco Teixeira Bastos da Mota

High Performance Fourier Transforms on GPUs with GLSL

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
António José Borba Ramires Fernandes

February 2023

ACKNOWLEDGEMENTS

First of all, I would like to thank my supervisor, Professor António Ramires, who accepted to guide me in this project, presenting on several occasions fundamental suggestions for the improvement of this dissertation, making the realization of this dissertation a true learning process.

Furthermore, I am extremely grateful to my family for their unconditional support, affection, and interest in pushing me in the right direction throughout my life.

I would also like to thank all my friends, who accompanied me along this path, who listened to my concerns and advised me when necessary.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ABSTRACT

The Fast Fourier Transform is a family of algorithms indispensable for the computation of the Discrete Fourier Transform. As a result, these 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 functionality 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 frequently used in the context of computer graphics.

In this dissertation we provide, refine and compare 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 such as the Stockham algorithm with higher radix.

Accordingly, we also use the cuFFT NVIDIA framework for reference in the comparisons of the GLSL algorithms implementations with the goal to analyse their significance on the tradeoff of using specialized implementations of the FFT algorithms or integrating dedicated software tools for any case of application.

Finally, we demonstrate how all improvements discussed in this dissertation culminate in performance improvement in a real-time rendering technique that heavily depends on multiple of these transforms in the Nau3D engine as a case of study.

KEYWORDS FFT, GPGPU, GLSL, cuFFT, performance, compute, Cooley-Tukey, Stockham.

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 usamos a *framework* cuFFT NVIDIA como referência nas comparações das implementações dos algoritmos em GLSL com o objetivo de analisar a sua importância no *tradeoff* entre usar implementações especializadas dos algoritmos FFT ou integrar ferramentas de *software* dedicadas para qualquer caso de aplicação.

Por fim, demonstramos como todas as melhorias discutidas nesta dissertação culminam na melhoria de desempenho numa técnica de renderização em tempo real que depende de FFTs na *engine* Nau3D como caso de estudo.

PALAVRAS-CHAVE FFT, GPGPU, GLSL, cuFFT, performance, compute, Cooley-Tukey, Stockham.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACRONYMS

CPU Central Processing Unit. 32, 39, 40
CUDA Compute Unified Device Architecture. 38

DFT Discrete Fourier Transform. 5, 12, 13, 17, 19, 28
DIF Decimation-in-Frequency. 19, 34, 46
DIT Decimation-in-Time. 16, 19, 34, 46

FFT Fast Fourier Transform. 5, 6, 7, 11, 12, 13, 14, 20, 21, 26, 27, 28, 38, 42

GLSL OpenGL Shading Language. 5, 6, 28, 40, 42, 43, 46
GPU Graphics Processing Unit. 5, 21, 28, 29, 32, 38, 39, 40, 43, 44

SIMT Single Instruction, Multiple Threads. 44

1

INTRODUCTION

1.1 CONTEXTUALIZATION

The Fast Fourier Transform (FFT) have been present in our surroundings for a long time. They are used extensively in digital signal processing and many other areas. This transform often needs to be used in a real-time context, where the computations must be performed fast enough to ensure the quality of the application. Fast Fourier Transforms essentially are optimized algorithms to compute the Discrete Fourier Transform (DFT) 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.

In computer graphics, the FFT can be implemented on the GPU to take advantage of the parallel nature of this algorithm and boost the performance for various application cases. Traditionally, this transform used to be implemented in the fragment shader within the graphics pipeline Moreland and Angel (2003), however, with the popularization of the GPUs general purpose compute functionality, this primitive can be implemented using the compute pipeline as of OpenGL 4.3.

1.2 MOTIVATION

Although there are already famous and efficient GPU frameworks such as cuFFT (Nvidia) and Intel Math Kernel Library (Wang et al. (2014)), integrating these Fast Fourier Transforms tools into an application can become a difficult process to manage and might introduce dependencies and unused functionality when the application just requires a specific case of the Fourier Transform.

When programmers need to implement features or applications that require the use of FFT, it is common to find custom implementations that offer fast enough Fast Fourier Transforms (Flügge (2017)), however, they are often left with the implementations of the most common algorithms such as Cooley-Tukey. Knowing the differences in algorithms and implementations, their advantages and disadvantages is essential to ensure the performance of the application in question.

This work focuses on studying the comparison of FFT implementations and improvements in GLSL compute shaders that make this a viable option instead of adopting the support of FFT frameworks for the GPU.

1.3 AIM OF THE WORK

The main objective of this dissertation is to provide, refine and compare efficient FFT implementations in GLSL in the context of computer graphics for power of 2 sizes. This dissertation achieves this by first studying the FFT and providing the required background on this transform and its properties, then proceeding to describe the Cooley-Tukey algorithm and its variants, and finally presenting different algorithms to improve the Cooley-Tukey variants.

Following that, will be an implementation stage, where the algorithms will be incorporated into compute shaders in GLSL and we study how we can compute and dispatch them more efficiently. To wrap things up, these implementations will be compared with each other under the same conditions for multiple sizes and using the cuFFT implementation benchmarks as a reference. The best results of the testing benchmarks will then be used for the analysis of the performance impact on a case study applied to computed graphics that heavily relies on multiple FFT to deliver quality performance.

1.4 DOCUMENT STRUCTURE

This dissertation has the following structure:

- Chapter 1 exposes an introduction to the subject of this dissertation with the contextualization, motivation, and aim of the work.
- Chapter 2 will have a state-of-the-art overview of the theory and practice associated with Fourier Transforms. This chapter also elaborates on other algorithms that improve the Cooley-Tukey algorithm and presents a way to perform the FFT of two real sequences in the same transform. The chapter ends with a description of how to apply the FFT in two dimensions.
- Chapter 3 describes in detail how to implement the previously discussed algorithms in GLSL.
- Chapter 4 analyzes and compares the implementations described in the previous chapter and demonstrates how the results apply to a more real case of application.
- Chapter 5 will discuss a conclusion to the work done and the results obtained and presents possible future work.

2

THE FOURIER TRANSFORM

It is 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 (2020), numerical integration, time-domain interpolation, and x-ray diffraction. Moreover, it is present in several fields of study such as Applied Mechanics, Signal Processing, Sonics and Acoustics, Biomedical Engineering, Instrumentation, Radar, Numerical Methods, Electromagnetics and more (Shakshi (2016), Lee et al. (2017), Brigham (1988)). As already mentioned before, this dissertation focuses more on the application of FFT in the context of Computer Graphics, where this transform is applied in several cases such as image processing, image filtering, and ocean waves simulation.

In Signal Analysis when representing a signal with amplitude as a 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.

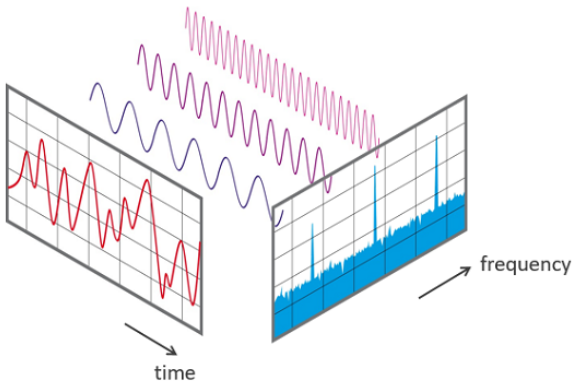

Figure 1: Time to frequency signal decomposition **Source:** NTiAudio

Sine and cosine waves are in simple waveforms, they can then be manipulated with relative ease. This process is constantly present in communications due to 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 preface on the Fourier Transform in Section 2.1 and describe the discrete version of the Fourier Transform in Section 2.2, which is the focus of this dissertation. Furthermore, we present the state of the art of the most popular algorithms in Section 2.3 and progressively demonstrate how

to improve them in the following sections Section 2.4 and Section 2.5. Regarding optimization of the FFT, we then proceed to detail how a complex-to-complex FFT computation can simultaneously transform two distinct real input sequences in Section 2.6. At the end of this chapter, in Section 2.7 we finalize with an explanation of how one-dimensional algorithms can be applied in the context of 2D input sequences such as an image

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 (Adams (2020)) can be described as Equation 1.

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

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* or *space* domain representation;
- $i \rightarrow$ imaginary unit $i = \sqrt{-1}$.

This formulation shows the usage of a 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 called the Discrete Fourier Transform (DFT). It converts a finite set of values in *time*/*space* domain to *frequency* domain representation. It is an usefull version of the Fourier Transform since it deals with a discrete amount of data. This transform is described in Equation 5.

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

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 simpler sum of real and imaginary numbers.

$$X_k = x_0 + x_1(\cos b_1 + i\sin b_1) + ... + x_{N-1}(\cos b_{N-1} + i\sin b_{N-1}) \tag{6}$$

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

Finally, the result will be a complex number.

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, therefore we 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 the frequency domain.

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

2.2.1 *Matrix multiplication*

The example shown above is done sequentially as if each frequency component is computed individually, but there is a way to express the DFT using matrix multiplication (Rao and Yip (2018)). Since the operations are done equally without any extra step we can group all analyzing function sinusoids ($e^{-\frac{i2\pi}{N}kn}$), also referred 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 (Rao and Yip (2018)), also called the DFT matrix, and its inverse can be defined as.

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

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

We separate the twiddle factors in a matrix and express the DFT as a matrix multiplication between the twiddle factors and the input sequence. When the size is known before the transformation, it is possible to pre-calculate and reuse the matrix of the factors

$$X = W \cdot x$$

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

It is 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 since it is more simple and efficient to normalize the inverse by N, than the forward and the inverse by $\sqrt{N}$.

EXAMPLE Continuing the previous example, we can adapt the application 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 is conspicuous that the complexity time for each complex 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 with the sequence length. Therefore, over time new algorithms and techniques were 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 compute the transform.

There are many algorithms and approaches on the FFT family such as the well known Cooley-Tukey Cooley and Tukey (1965), 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 size N in one DFT of size $N/2$ and two DFTs of size $N/4$.

Computing the DFT directly requires N^2 complex multiplies and $N(N-1)$ complex additions, by using an FFT algorithm to compute the DFT, it only requires $(N/2)\log(N)$ complex multiplies and complex additions.

When we consider a DFT of a sequence of length 4, we can notice that some of the operations involved repeat themselves for the computation of the whole sequence.

In Equation 7 we list the required operations to compute the DFT of a sequence with 4 elements.

$$\begin{aligned} X_0 &= x_0 + x_1 + x_2 + x_3 \\ X_1 &= x_0 - i \cdot x_1 - x_2 + i \cdot x_3 \\ X_2 &= x_0 - x_1 + x_2 - x_3 \\ X_3 &= x_0 + i \cdot x_1 - x_2 - i \cdot x_3 \end{aligned} \tag{7}$$

When the operations are grouped together, we have a clear perception of the repetition of the operations, as presented in Equation 8.

$$
\begin{aligned}
X_0 &= (x_0 + x_2) + (x_1 + x_3) \\
X_1 &= (x_0 - x_2) - i \cdot (x_1 - x_3) \\
X_2 &= (x_0 + x_2) - (x_1 + x_3) \\
X_3 &= (x_0 - x_2) + i \cdot (x_1 - x_3)
\end{aligned} \tag{8}
$$

The FFT takes advantage of the periodic roots of unity values and the repetitive operations to generalize the grouping in Equation 8 for a sequence of size N. This algorithm itself is a multiplication of the input sequence by a sparse matrix (Heckbert (1995)). It achieves this by computing the butterfly operations at each stage and storing them to allow the reuse of the intermediate values in the next stage until the DFT is computed for every element.

It is worth noting that for power of 2 sizes, the roots of unity are symmetrical and periodic over N, therefore, any twiddle factor ω_N^k is equal to ω_N^{k+N} and $-\omega_N^{k+N/2}$. For example, in figure Figure 2 we have the roots of unity for $N = 8$, where $\omega_8^0 = -\omega_8^4$ and $\omega_8^0 = \omega_8^8$.

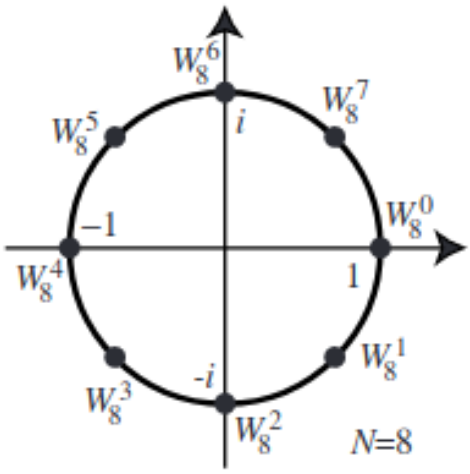

Figure 2: Roots of unity with $N = 8$ **Source:** Heckbert (1995)

Section 2.3.1 and Section 2.3.2 focus on the Cooley–Tukey algorithm for sequences with power of 2 sizes. In Section 2.3.1 we describe how to generalize the FFT algorithm with decimation in time, and explain where this algorithm reduces the computational cost of the DFT. Moreover, in Section 2.3.2 we present another well-known variation of this algorithm with decimation in frequency.

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 the other as a sum over the odd indexed elements.

Cooley and Tukey proved this possibility of dividing the DFT computation into two smaller DFTs by exploiting this division (Cooley and Tukey (1965)), as presented in Equation 10. Hence, it is hinted the recursive definition of this formulation on both DFT of size $N/2$.

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

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

This formulation segments the full-sized DFT into two $N/2$ sized DFTs of the even and odd indexed elements where the latter is multiplied by a factor called twiddle ω_N^k. We can notice that in the $N/2$ sub-transform, the twiddle factors of $\omega_{N/2}^k$ will be periodic for $k = 0, 1, ..., N-1$. This periodicity can also be noticed in Equation 8 within the sub-transforms of the even and odd elements.

This algorithm is a Radix-2 Decimation-in-Time in the sense that elements are regrouped into 2 sub-transforms, and the decomposition reduces the time values to the frequency domain. The understanding of this algorithm comes from the recursive application of Equation 10, to compute the FFT for all the values of the input sequence. Figure 3 illustrates the composition of the sub-transforms, where the DFTs of size $N/2$ can be replaced with the same decomposition of even and odd sub-transforms.

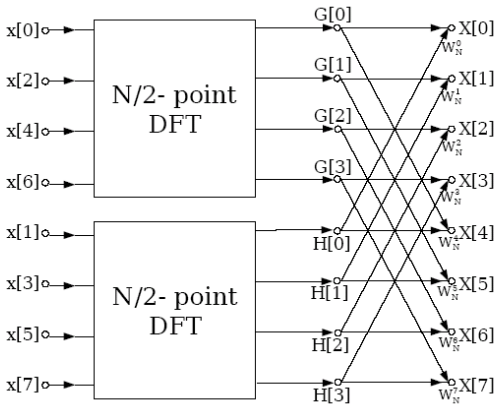

Figure 3: Radix-2 Decimation-in-Time FFT **Source:** Jones (2014)

Effectively, the main building block in the FFT is the butterfly operation of 2 elements since it calculates intermediate values at each stage to compute the FFT of the whole sequence. This butterfly operation resembles the computation of a length-2 DFT but with a shifted element according to the sub-transform size

This butterfly, without any reuse of the twiddle factor, corresponds to:

$$\begin{aligned} A &= a + b \cdot \omega_N^k \\ B &= a + b \cdot \omega_N^{k+N/2} \end{aligned} \tag{10}$$

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

Yet, since the roots of unity for k and $k + N/2$ over N are symmetrical, as illustrated in Figure 2, 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 equal to $-\omega_N^k$ (Jones (2014)). With this said, Cooley-Tukey butterfly (Chu and George (1999)) reuses the same twiddle factor, as illustrated in Figure 4.

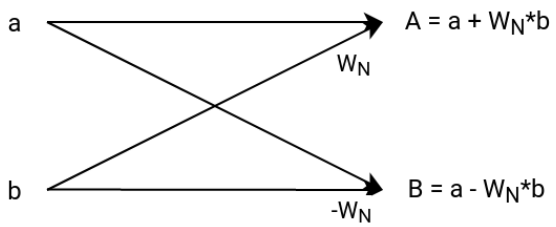

Figure 4: Cooley-Tukey butterfly

The complexity work of the algorithm is distributed within 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 Thong (1981) this order corresponds to the bit reversed indices of the sequence, therefore, we need to apply a bit reverse permutation to the elements of the input sequence.

The bit reversal of the input sequence corresponds to the permutation of swapping the position of the elements to its bit reversed index, as illustrated in Figure 5. The $bit\_reverse$ of an index depends directly on the indexing domain of the input sequence, therefore, it needs the size N, or more precisely the $\log(N)$ value, to use as a reference to reverse the bit order while maintaining the value within the sequence range.

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

For the DIT algorithm, we apply the bit reversal permutation to the input sequence to return a natural order result.

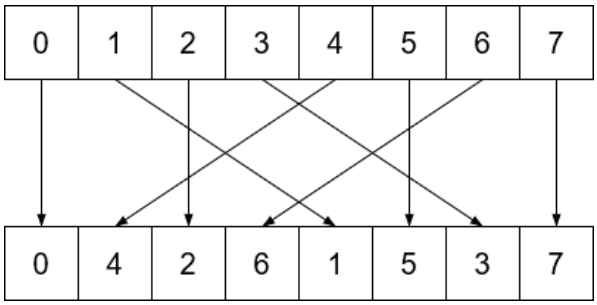

Figure 5: Bit reverse permutation

There are many implementations of the bit reversal, and since it is quite simple, any suitable version can be used in regard to 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 with an iterative DIT implementation for the forward FFT. To compute its inverse the algorithm is the same, we just need to use the twiddle factor with a positive exponent $w_m \leftarrow \exp(2\pi i/m)$.

Although this algorithm is congruent with a code implementation, it is 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 innermost 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, it is based on the same principle of divide-and-conquer, however, it rearranges the original DFT into the computation of two transforms, one with the even indexed elements and other with the odd indexed elements, as demonstrated in Equation 12.

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

$$
\begin{aligned}
X_{2k} &= \sum_{n=0}^{N/2-1} (x_n + x_{n+\frac{N}{2}}) \cdot \omega_N^{2kn} = \sum_{n=0}^{N/2-1} (x_n + x_{n+\frac{N}{2}}) \cdot \omega_{N/2}^{kn} \\
X_{2k+1} &= \sum_{n=0}^{N/2-1} (x_n + x_{n+\frac{N}{2}}) \cdot \omega_N^{(2k+1)n} = \sum_{n=0}^{N/2-1} ((x_n - x_{n+\frac{N}{2}}) \cdot \omega_{N/2}^{kn}) \cdot \omega_N^{n}
\end{aligned}
\tag{12}
$$

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

As explained in the previous section, the computational savings for this algorithm come from exploiting the DFT to reuse repeated calculations by storing intermediate terms in between stages.

This algorithm is a Radix-2 Decimation-in-Frequency since the DFT is decimated into two distinct smaller DFT's and the frequency samples will be computed separately in different groups as if the regrouping of the DFTs would reduce directly to the frequency domain. The understanding of this algorithm comes from the recursive application of Equation 12, to compute the FFT for all the values of the input sequence. Figure 6 illustrates the composition of the transforms, where the DFTs of size $N/2$ can be replaced with the same decomposition in two transforms for the even and odd elements.

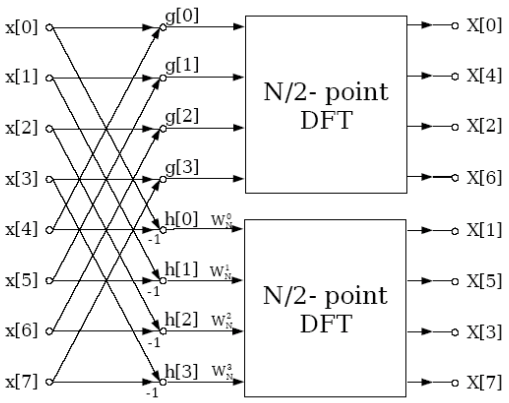

Figure 6: Radix-2 Decimation-in-Frequency FFT **Source:** Jones (2014)

Similarly to the DIT version, the DFT can be recursively reduced by the DIF algorithm until there is only the computation of a length-2 DFT. At each stage, it is applied the Gentleman-Sande butterfly operation (Chu and George (1999)) with a shifted element according to the sub-transform size, as illustrated in Figure 7.

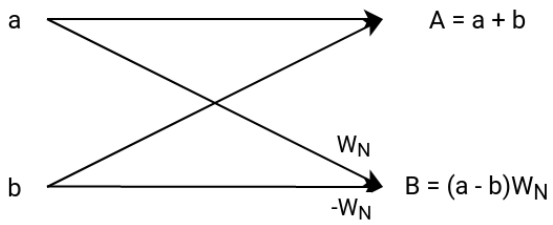

Figure 7: Gentleman-Sande butterfly

Since this algorithm has similarities with the DIT, its complexity also lives to this similarity, maintaining the same $O(N \log(N))$ for the number of complex multiplications. Although they look different, Figure 7 and Figure 4 have the same number of operations 1 addition 1 subtraction, and 1 complex multiplication.

Unlike DIT, the DIF algorithm applies bit reversal permutation (Figure 5) to the output sequence to produce a natural order result.

In practice, Algorithm 3 demonstrates the aforesaid with an iterative representation of a possible implementation. To compute its inverse the algorithm is the same, we just need to use the twiddle factor with a positive exponent $w_m \leftarrow \exp(2\pi i/m)$. Although this algorithm is congruent with a code implementation, it is 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 innermost 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;

Nowadays, there is a lot more to the computation of FFTs than just the basic radix-2 Cooley-Tukey algorithm described previously. As seen by a large amount of current literature, there are more algorithms, variations, and improvements that enhance the computation of this transform in many aspects. Choosing the right conditions enhances the calculation of the performance of this primitive, especially for specific systems (Bengtsson (2020) and Mermer et al. (2003)).

One could optimize the FFT in many ways, and in the next section, we introduce a power of 2 sizes algorithm called radix-2 Stockham, a natural order algorithm without explicit bit reversal permutation.

2.4 STOCKHAM ALGORITHM

Despite existing fast solutions for index bit reversal (Prado (2004)), this *shuffle* step still weighs 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 8.

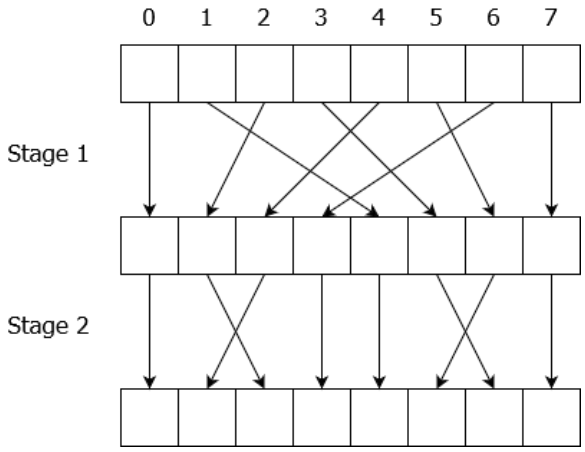

Figure 8: Chain of even and odd compositions over each stage for a natural order DIF

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 13.

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

$$x[q + 2 * p + 1] = y[q + p + m] \tag{14}$$

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.

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 that 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 determine which sequence the result is based on the number of stages $\log(N)$.

This algorithm is described in Algorithm 4. To compute its inverse the algorithm is the same, we just need to use the twiddle factor with a positive exponent $w_p = \exp(2 * \pi * i * p/gs)$. This version may seem visually different from the Cooley Tukey, however, it preserves the algorithm 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 within every 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 $stage = 0$ ***to*** $\log(N) - 1$ **do**
  $gs \leftarrow N \gg stage$;
  $s \leftarrow 1 \ll stage$;
  foreach $i = 0$ ***to*** $N - 1$ **do**
    $p \leftarrow i$ div s;
    $q \leftarrow i$ mod s;
    $w_p = \exp(-2 * \pi * i * p/gs)$;
    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.

2.5 RADIX-4 INSTEAD OF RADIX-2

After presenting the radix-2 version of the Stockham algorithm, in this section we expand the application of this algorithm for higher radix such as radix-4 and consequently reflect on its advantages and disadvantages.

We've discussed 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 a 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 (Bailey (1988)).

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 sub-transforms 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 at every stage, as illustrated in Figure 9.

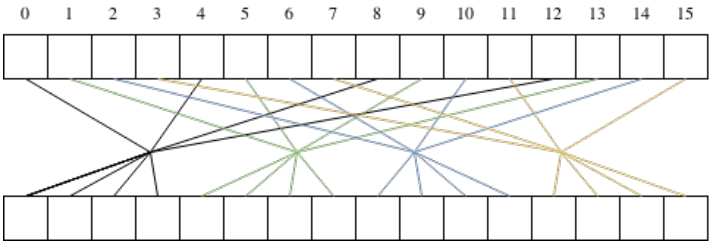

Figure 9: Illustration of a stage in radix-4 Stockham with each color representing a radix-4 butterfly computation

The dragonfly for this algorithm is illustrated in Figure 10 and its computation involves 4 elements that compute 2 stages of 2 radix-2 butterflies.

In Equation 15 is presented the forward radix-4 dragonfly operations and in Equation 16 its inverse.

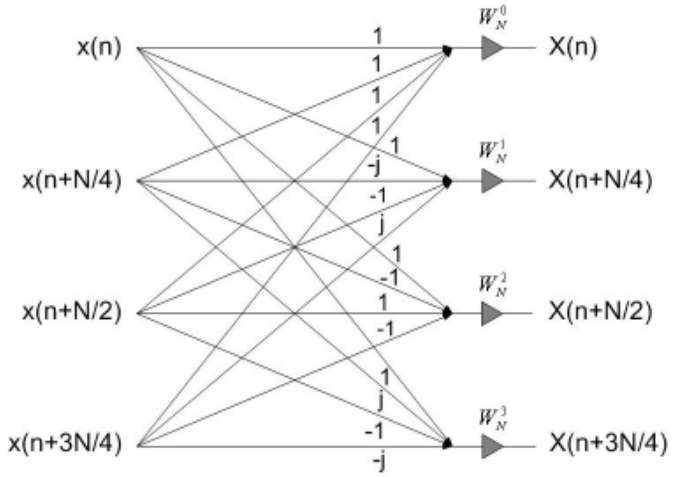

Figure 10: Radix-4 FFT butterfly structure **Source:** Marti-Puig and Bolano (2009)

$$\begin{aligned} A &= (a+c)+(b+d) \\ B &= ((a-c)-i(b-d)) \times \omega_N^{p} \\ C &= ((a+c)-(b+d)) \times \omega_N^{2p} \\ D &= ((a-c)+i(b-d)) \times \omega_N^{3p} \end{aligned} \tag{15}$$

$$\begin{aligned} A &= (a+c)+(b+d) \\ B &= ((a-c)+i(b-d)) \times \omega_N^{-p} \\ C &= ((a+c)-(b+d)) \times \omega_N^{-2p} \\ D &= ((a-c)-i(b-d)) \times \omega_N^{-3p} \end{aligned} \tag{16}$$

With each stage, the dragonflies are calculated and the elements are reordered similarly to the radix-2 Stockham algorithm. In the end, we get the forward radix-4 Stockham algorithm Algorithm 5. The inverse algorithm is similar, but instead uses the inverse dragonfly as presented in Equation 16.

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 $stage \; mod \; 2 == 0$ **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

2.6 TWO REAL INPUTS WITHIN ONE COMPLEX

It is possible that in the case of an application there is a requirement to compute multiple real 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 11.

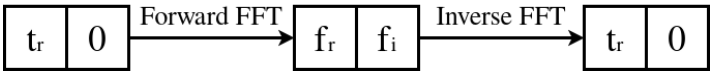

Figure 11: FFT for an element without an imaginary part

Based on this, we can reuse the imaginary part of the complex with a meaningful value other than 0, as illustrated in Figure 12.

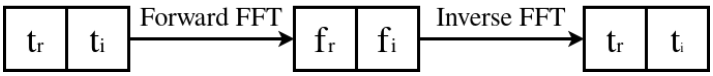

Figure 12: FFT for an element with an imaginary part

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 Shima (2000). 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 17 and Equation 18 respectively.

When considering two real valued sequences x_1 and x_2, whose transforms are X_1 and X_2, and having $y = x_1 + ix_2$, we can get the mixed frequencies in Y by packing the separated transforms as in Equation 17.

$$Y(k) = X_1(k) + X_2(k) \tag{17}$$

When we have the mixed frequency components Y, we may unpack X_1 and X_2 as in Equation 18, where Y_r and Y_i are the real and imaginary parts of the complex-valued element.

$$\begin{aligned} X_1(k) &= \frac{Y(k) + (Y_r(N-k) - Y_i(N-k))}{2} \\ X_2(k) &= \frac{Y(k) - (Y_r(N-k) - Y_i(N-k))}{2} \end{aligned} \tag{18}$$

2.7 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 the real domain on its inverse.

When dealing with images, we might have multiple values per pixel element, therefore, we have the possibility of using it as a greyscale image if we derive the relative luminance via quantized RGB signals of the image (ITU (2002)), using only the values of one channel, or compute multiple FFTs for each channel values. With this said, when transforming the input in the forward FFT we will obtain a complex-valued output, and when applying the inverse we obtain the real-valued result.

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 $N \times N$ is equivalent to $2 * N$ 1D FFT of size N, as presented in Equation 19. This is called the row–column decomposition (Mermer et al. (2003)).

$$X_{r,c} = \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} x_{n,m} \cdot \omega_N^{rn} \cdot \omega_N^{cn} \tag{19}$$

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 (Figure 13).

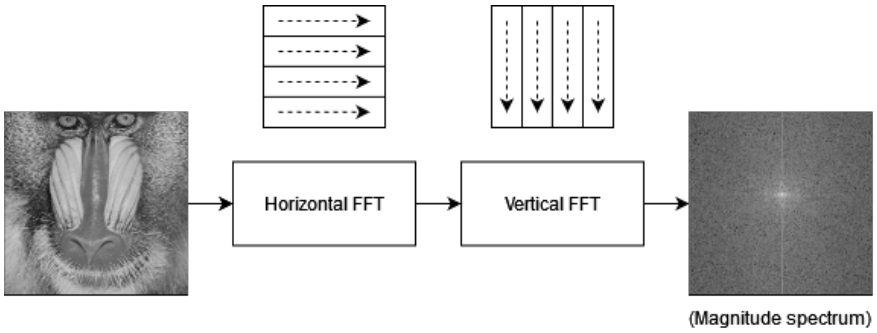

Figure 13: High-level illustration of horizontal and vertical passes

Additionally, the implementation of the 1D FFTs for every row and column is independent of the 2D FFT structure, hence, there is the freedom to use any of the discussed algorithms. In Chapter 3, we implement multiple FFT algorithms while reusing the same architecture of the horizontal and vertical passes.

3

IMPLEMENTATION IN GLSL

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.

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 an FFT implementation using compute shaders, which are general-purpose programmable shaders.

As there are many aspects that can impact performance, the implementation was an iterative process that required research and testing to compose a more suitable solution.

The next sections go into detail about the way every major iteration evolved into the next one and why there was the need to do it, starting with the Cooley-Tukey algorithm (Section 3.1) and then progressively improving to the Stockham algorithm (Section 3.2 and Section 3.3)

3.1 COOLEY-TUKEY

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

Since this implementation runs multiple threads in parallel we separated 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 3.1: Input buffer bindings

The read/write control for these 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 is a synchronization step at the end of the pass it is 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, hence, the local threads can be dispatched with two dimensions. Each work group will have a total of 32 local threads, this number may vary depending on the GPU for optimal performance.

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 are some advantages to complex value operations since the addition and subtraction for vector types already have operator overloadings that work 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 3.2: 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 process 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 3.3: Invocation indices

Since this first approach is a dynamic implementation that invokes a pass-per-stage some stage control variables need to be fed 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 3.4: Uniform control variables

Effectively, we use these shader uniform input variables and obtain the actual index we are 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 3.5: FFT element index

To calculate the twiddle factor we use Euler's formula (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));
    // ...
}
```

Listing 3.6: Euler's formula

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 texture we will use to look up 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 presented in Section 2.3.1.

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

    // Compute and store
    vec2 wb = complex_mult(w, b);
    vec2 raux = a + wb;
    imageStore(pingpong1, ivec2(idx, column), vec4(raux, 0, 0));
    raux = a - wb;
    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 wb = complex_mult(w, b);
    vec2 raux = a + wb;
    imageStore(pingpong0, ivec2(idx, column), vec4(raux,0,0));
    raux = a - wb;
    imageStore(pingpong0, ivec2(idx + shift, column), vec4(raux,0,0));
}
```

Listing 3.7: Computation of the Cooley-Tukey butterfly

Finally, there is 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` (Kessenich et al.).

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

Listing 3.8: Auxiliary function that takes advantage of GLSL's predefined utilities

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, it is ensured that the first stage will always read from `pingpong0` reinforcing 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
}
```

Listing 3.9: Computation of the Cooley-Tukey butterfly with bit reversal

With all these 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 Equation 5, this is demonstrated in Listing A.2.

3.1.1 *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 stage itself, moreover, it requires 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.

There is a lot of literature on behalf of GPU compute execution synchronization (Stuart and Owens (2011)). In our case, we use execution barriers that synchronize all the threads within a work group (Kessenich et al.).

The current grouping of threads doesn't allow the use of these barrier synchronizations since the computation of one-dimensional FFT is distributed between multiple work groups, thus, 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 a way that each 1D FFT threads fit in one work group as illustrated in Figure 14.

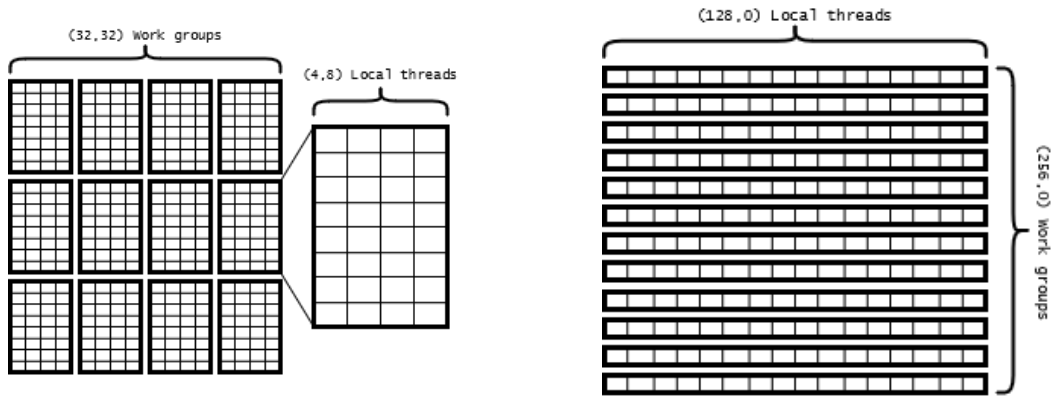

Figure 14: Difference in invocation spaces for size 256 FFT to allow barrier synchronization between local threads

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's local size must now be restricted to half the size of the FFT. This restriction does not affect our results, where the tests were performed on images with a relatively small size. The GPU is extremely parallel hardware, therefore, it is 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 be able to unroll the loop, since the for loop feature in GLSL requires to be used with constant expressions that allow the loop to be inlined 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 3.10: Unique pass structure for Cooley-Tukey

Aside from the variable `fft_dir` that allows the 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.

3.2 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 2.4). What differs from the previous code is mainly the reordering of the elements when writing the results of the butterfly.

Note that this algorithm is in DIF instead of DIT like the previous ones, although this change does not have a performance implication.

When performing the butterfly in Listing 3.11 we read the elements from the image according to the thread identifier, however, we store it in such a 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 3.11: 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.

3.3 RADIX-4 STOCKHAM

In Section 2.5 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 3.12: Radix-4 Stockham stage control variables

Then, we compute the radix-4 butterfly and store the results in natural order, as described in Figure 10.

```
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 3.13: 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 3.14: 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 3.2, but partially more hardcoded for the 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 3.15.

```
#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));

            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));

            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 3.15: Radix-2 stage for the radix-4 Stockham code

4

ANALYSIS AND COMPARISON

Finally, in 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 the CUDA Fast Fourier Transform library from NVIDIA.

4.1 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 Sörman (2016) and other resources.

The library is acknowledged as one of the most efficient FFT GPU frameworks for the flexibility it provides and it is "*de-facto a standard GPU implementation for developers using CUDA*" (Střelák and Filipovič (2018)). 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 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 better performance than higher prime factors (Nvidia), thus power of 2 sizes have the best performance overall.

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

- Out-of-place 2D FFT, the 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 taken into account since takes extra time to set up things on the GPU.

The results of the cuFFT out-of-place benchmarks are presented in Figure 16.

4.2 GLSL IMPLEMENTATION RESULTS

The implementations discussed in Chapter 3 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 (Ramires) and profiled it using an internal pass profiler.

The benchmark results in this section were tested with the following hardware and software configurations:

- **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 3.1.1 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 15 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.

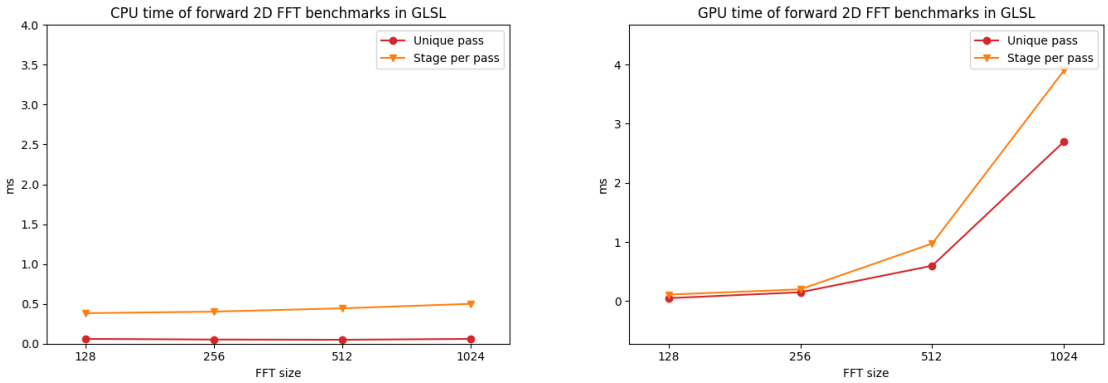

Figure 15: CPU and GPU benchmarks of 2D forward FFT using stage-per-pass approach and unique pass for the Radix-2 Cooley-Tukey algorithm

Sizes		128	256	512	1024
Unique pass	CPU	0.06	0.052	0.049	0.06
	GPU	0.05	0.15	0.598	2.698
stage-per-pass	CPU	0.383	0.402	0.444	0.5
	GPU	0.11	0.199	0.973	3.899

Table 1: CPU and GPU benchmarks of 2D forward FFT using stage-per-pass approach and unique pass for the Radix-2 Cooley-Tukey algorithm

The results in Figure 15 and Table 1 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 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 completes the execution. Since this type of synchronization is on the work group level, much fewer actors are 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 16 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 Chapter 3. 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.

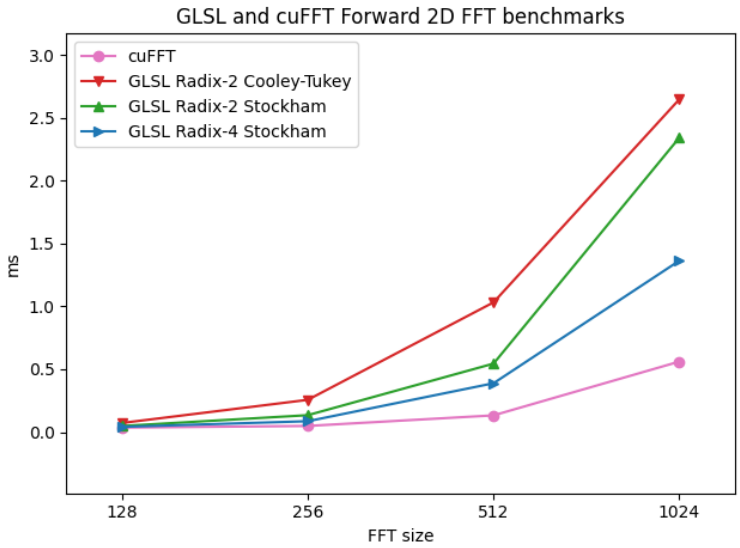

Figure 16: Forward 2D FFT benchmarks in milliseconds of out-of-place cuFFT and unique pass algorithms in GLSL

Sizes	128	256	512	1024
cuFFT	0.0359	0.0494	0.1335	0.5609
GLSL radix-2 Cooley-Tukey	0.073	0.257	1.032	2.646
GLSL radix-2 Stockham	0.049	0.135	0.545	2.341
GLSL radix-4 Stockham	0.042	0.087	0.389	1.363

Table 2: Benchmarks of the forward 2D FFT benchmarks in milliseconds of out-of-place cuFFT and unique pass algorithms in GLSL

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 16 and Table 2. 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 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.

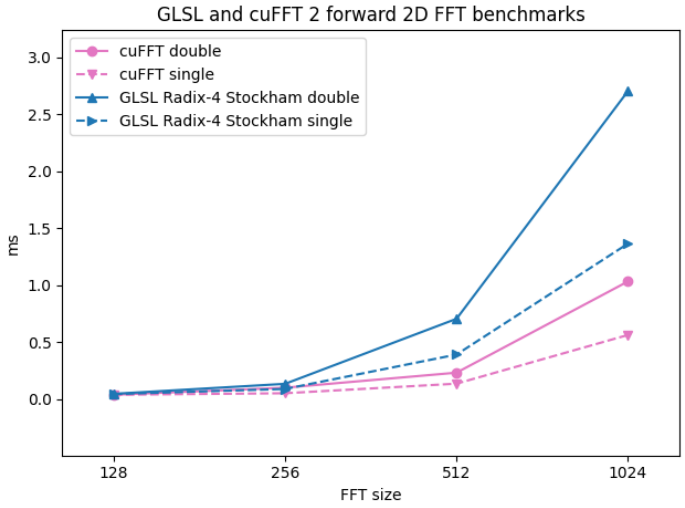

Figure 17: Forward 2D FFT benchmarks of double the transforms in milliseconds of out-of-place cuFFT and radix-4 Stockham GLSL

In table Table 3, the benchmarks of Figure 17 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.

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 3: Benchmarks 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.

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, 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.

4.3 CASE OF STUDY

Based on the findings of Section 4.2 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 demanding scenario by using an ocean rendering technique demo that heavily relies on the usage of FFT (Figure 18).

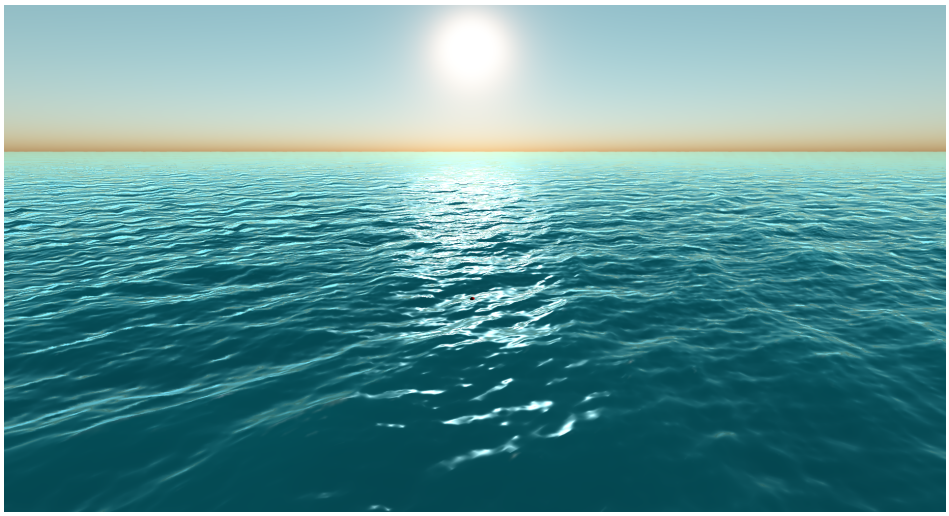

Figure 18: Tensendorf waves in Nau3D Engine

In this section we brief the Tensendorf waves demo in Section 4.3.1 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 4.3.2.

4.3.1 *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 Tessendorf et al. (2001). 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 4.2, 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, we used `vec4` as described in Section 4.2 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 Tessendorf et al. (2001).

Similarly to Section 4.2, 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.

4.3.2 *Results*

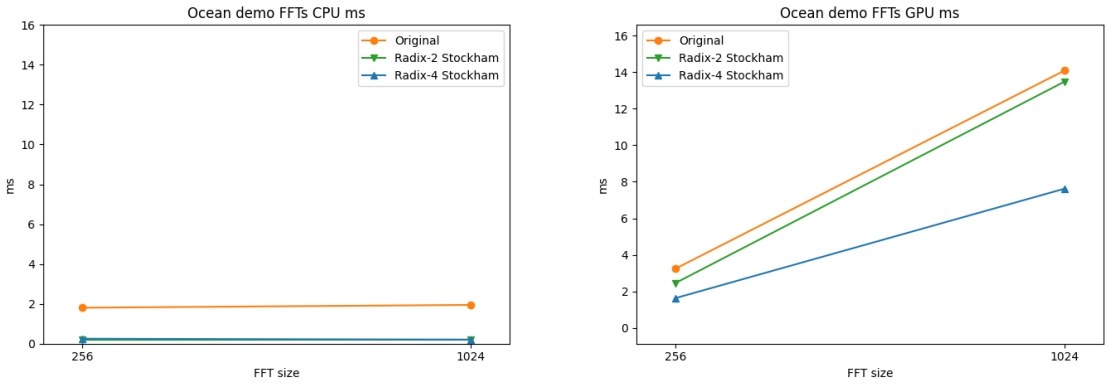

Figure 19: Time spent in the CPU and GPU for the size 512 FFT horizontal and vertical passes

Sizes		512	1024
Original	CPU	1.805	1.946
	GPU	3.241	14.086
Radix-2 Stockham	CPU	0.191	0.203
	GPU	2.457	13.48
Radix-4 Stockham	CPU	0.249	0.2
	GPU	1.632	7.621

Table 4: Time spent in the CPU and GPU for the size 512 FFT horizontal and vertical passes

In Figure 19 and Table 4 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 need to implement adequate algorithms to ensure the quality and performance of the target application.

5

CONCLUSIONS AND FUTURE WORK

In this thesis, we provided background on the Discrete Fourier Transform, with the goal of introducing the DIT and DIF Cooley-Tukey algorithms. From there, we described how the Stockham algorithm removes the need for the bit reversal permutation and what is the radix-4 of this algorithm. Moreover, we also discussed how a complex-valued FFT can be used to compute 2 real-valued input sequences without any additional cost, and how this transformation can be applied to the spatial domain of a 2D image.

After presenting the Fast Fourier Transform and its algorithms, we processed implementing the Cooley-Tukey algorithm with a stage-per-pass approach followed by unique pass implementations of the radix-2 Cooley-Tukey and Stockham algorithm. Additionally, we also implemented the radix-4 Stockham algorithm, however, with the intent of having this algorithm support sizes of 2 instead of just power of 4, we altered the final stage of the code to conditionally implement a final stage of radix-2 Stockham.

Finally, to analyze these algorithms, we conducted a comparison of the performance of the implementations with benchmarks in a test environment and also measured implementation details such as the advantages and disadvantages of using stage-per-pass versus unique-pass and double the FFTs in GLSL by reusing the same `vec4` operations. From the compared algorithms, the best results were those of radix-4 Stockham, which we used as a model to test the possible benefits in the case study. The case study demonstrated positive results, by applying the previously discussed knowledge in the real-time rendering technique with radix-4 Stockham multiple FFTs.

5.1 RESULTS

Based on the findings of Chapter 4, we were able to conclude several aspects of the tests carried out. When it comes to the tested unique-pass approach, it was demonstrated to be faster than the stage-per-pass, yet, we concluded multiple advantages and disadvantages:

Advantages:

- Synchronization only on the GPU, and not with the CPU until the 2D FFT is fully calculated;
- The work synchronizes only on the 1D FFT, and not on the whole 2D FFT, hence, it finishes the work earlier than the stage-per-pass approach;

- Unique-pass shader has more constant expressions and information compared to the stage-per-pass shader, thus it is more prone to compiler optimizations.

Disadvantages:

- Higher kernel size;
- The implementation is specialized for a specific size, therefore, not as flexible to reuse for other sizes. While the stage-per-pass implementation is reused for multiple power of 2 sizes, the unique-pass strategies require changing the provided macro constants and recompiling as an additional shader;
- Mixed radix stages do not have special invocation spaces. This was the case of the radix-4 Stockham for power of 2 sizes, we were required to use duplicate the amount of radix-2 Stockham stage performed by each thread since there were only $FFT\_SIZE/4$ threads.

Furthermore, when it comes to comparing implementations, it was found that radix-4 Stockham obtained the best results, up to $2\times$ faster than radix-2 Cooley-Tukey. This was mainly due to the elimination of the bit reversal permutation and the reduction in the number of memory accesses by half due to the computation of 2 radix-2 butterflies of 2 radix-2 stages at once.

When the acquired knowledge was applied to the case study, as expected, there was an increase in performance similar to that of the tests carried out. This case study applied strategies that were described and tested, such as the use of double FFTs by using the `vec4` operations and the multiple real values from the FFT frequencies. The results improved the frames per second of the application by up to 60%.

5.2 FUTURE WORK

With the obtained results, other questions arise, which may lead to future studies to expand the work carried out.

Although the radix-4 reduced the memory accesses by half, these memory loads still weigh on the performance of the compute shaders, other strategies for this problem could be studied to reduce the overhead this introduces.

Since the results of the radix-4 Stockham were the fastest when using radix higher than 4, what is the most suitable configuration for dispatching a higher radix implementation while still supporting power of 2 sizes? With the size restriction of the invocation space until the kernel is done executing, the unique-pass approach needs to duplicate the number of butterflies for each lower radix stage of the implementation. Therefore, we could study the implementation of these higher radix algorithms to verify if the usage of stage passes for the lower radix stages would be more beneficial than strictly having a unique-pass approach. A study could be carried out to explore these conditions for multiple GPUs, and come to a conclusion on the best configuration on average for the tested hardware.

BIBLIOGRAPHY

Michael D Adams. *Signals and Systems (Edition 3.0)*. Michael Adams, 2020.

David H Bailey. A high-performance fft algorithm for vector supercomputers. *The International Journal of Supercomputing Applications*, 2(1):82–87, 1988.

Gabriel Bengtsson. Development of stockham fast fourier transform using data-centric parallel programming, 2020.

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.

James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

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

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.

Paul Heckbert. Fourier transforms and the fast fourier transform (fft) algorithm. *Computer Graphics*, 2:15–463, 1995.

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.

ITURBT ITU. Parameter values for the hdtv standards for production and international programme exchange. *Recommendation ITU-R BT*, pages 709–5, 2002.

Yan-Bin Jia. Polynomial multiplication and fast fourier transform. *Com S*, 477:577, 2020.

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

John Kessenich, Dave Baldwin, and Randy Rost. The opengl® shading language. 2016. *URL: https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.4.60.pdf*, 4.

YC Lee, VC Koo, and YK Chan. Design and development of fpga-based fft co-processor for synthetic aperture radar (sar). In *2017 Progress in Electromagnetics Research Symposium-Fall (PIERS-FALL)*, pages 1760–1766. IEEE, 2017.

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.

Coskun Mermer, Donglok Kim, and Yongmin Kim. Efficient 2d fft implementation on mediaprocessors. *Parallel Computing*, 29(6):691–709, 2003.

Kenneth Moreland and Edward Angel. The fft on a gpu. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119, 2003.

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.

António Ramires. Nau 3D Engine. https://github.com/Nau3D/nau. Accessed at 2022-11-21.

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

Ramavtar Jaswal Shakshi. Brain wave classification and feature extraction of eeg signal by using fft on lab view. *Int. Res. J. Eng. Technol*, 3:1208–1212, 2016.

J Shima. Computing the fft of two real signals using a single fft. 2000.

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.

Torbjörn Sörman. Comparison of technologies for general-purpose computing on graphics processing units, 2016.

David Střelák and Jiří Filipovič. Performance analysis and autotuning setup of the cufft library. In *Proceedings of the 2nd Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems*, pages 1–6, 2018.

Jeff A Stuart and John D Owens. Efficient synchronization primitives for gpus. *arXiv preprint arXiv:1110.4623*, 2011.

Jerry Tessendorf et al. Simulating ocean water. *Simulating nature: realistic and interactive techniques. SIGGRAPH*, 1(2):5, 2001.

Tran Thong. Algebraic formulation of the fast fourier transform. *IEEE Circuits & Systems Magazine*, 3(2):9–19, 1981.

Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*, pages 167–188. Springer, 2014.

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 3.14159265358979323846264338327795

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

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

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

Listing A.1: FFT Radix-2 Cooley-Tukey Horizontal stage pass, see Section 3.1

```
#version 440

#define M_PI 3.14159265358979323846264338327795

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

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

        raux = (a - wb) * 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 3.1

```
#version 440

#define M_PI 3.14159265358979323846264338327795
#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 wb = complex_mult(w, b);
            vec2 raux = a + wb;
            imageStore(pingpong1, ivec2(idx, column), vec4(raux, 0, 0));
            raux = a - wb;
            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 wb = complex_mult(w, b);
            vec2 raux = a + wb;
            imageStore(pingpong0, ivec2(idx, column), vec4(raux,0,0));
            raux = a - wb;
            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 3.1.1

```
#version 440

#define M_PI 3.14159265358979323846264338327795
#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 wb = complex_mult(w, b);
            vec2 raux = (a + wb) * mult_factor;
            imageStore(pingpong1, ivec2(line, idx), vec4(raux,0,0));
            raux = (a - wb) * 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 wb = complex_mult(w, b);
            vec2 raux = (a + wb) * mult_factor;
            imageStore(pingpong0, ivec2(line, idx), vec4(raux,0,0));
            raux = (a - wb) * 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 3.1.1

```
#version 440

#define M_PI 3.14159265358979323846264338327795
#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 3.2

```
#version 440

#define M_PI 3.14159265358979323846264338327795
#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 3.2

```
#version 440
```

```
#define M_PI 3.14159265358979323846264338327795
#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 3.3

```
#version 440

#define M_PI 3.14159265358979323846264338327795
```

```
#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 3.3

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 4.1