Delft University of Technology
Faculty Electrical Engineering, Mathematics & Computer Science
Delft Institute of Applied Mathematics

GPU Acceleration Of The PWTD Algorithm For Application In High-Frequency Communication And Fotonics

Thesis Report for the
Delft Institute of Applied Mathematics
as part of

the degree of

MASTER OF SCIENCE
in
APPLIED MATHEMATICS

by

Rory Gravendeel

Delft, the Netherlands
23 August 2020

TU Delft

MSc Thesis report APPLIED MATHEMATICS

"GPU Acceleration Of The PWTD Algorithm For Application In High-Frequency Communication And Fotonics"

RORY GRAVENDEEL
4076494

Delft University of Technology

Thesis advisor

Dr. K. Cools

Members of the graduation committee

Prof.dr.ir. C. Vuik

Prof.dr.ir. H.X. Lin

R. van Driel, MSc

23 August 2020 Delft

Abstract

When creating electronic devices, it is essential to model what happens when an electromagnetic field hits the device and it scatters. Conventionally, this can be modelled using the Marching-on-in-Time algorithm. This can become computationally expensive for complex systems. To speed up the algorithm, the Plane-Wave Time-Domain algorithm is combined with the MOT algorithm. To accelerate the process even more, part of the algorithm is implemented using a Graphics Processing Unit, or GPU.

To test if using GPUs for this type of problem is actually beneficial, three experiments are set up. The first one tests the basic operations of addition and multiplication on matrices and vectors of various sizes, to determine if and when the computation time of the GPU is lower than that of a CPU. The second experiment tests the use of Fast Fourier Transform planner functionality and compares the CPU computation time with that of the GPU for the FFT of matrices of various sizes. The third experiment compares an example of the PWTD algorithm on the CPU and the GPU. These experiments are performed on three different devices.

The results from experiment 1 and 2 show that, after a certain point, the GPU is almost always faster, no matter the operation. Experiment 3 shows that the current GPU implementation is currently not as fast as the regular PWTD algorithm, though one of the devices is only 0.003% slower.

In conclusion, theoretically a decrease in computation time is expected. From experiment 3 it follows that it is not the case yet, though with more optimisation the GPU implementation would almost certainly become faster.

Contents

Chapter 1

Introduction

GPUs have become ubiquitous in the current data science world of Artificial Intelligence and Machine Learning. They are capable tools that use parallel computation to significantly speed up processes. Due to the lowering costs for capable hardware and the availability of easy to use programming languages that make use of GPUs, parallelisation has seen an expansion in both research and modelling in many major scientific fields, including in the field of mathematics. To see what kind of an effect parallelisation can have, this research aims to speed up a current complex algorithm by implementing parts of it on a GPU.

The algorithm in question uses the Plane-Wave Time-Domain algorithm to enhance a Marching-on-in-Time scheme. The main component of this algorithm uses convolutions, which can be mathematically difficult to solve. Fourier transforms can be used to solve the convolutions more efficiently; Fourier transforms can then be parallelised using various algorithms, which is why they are the prime candidates for GPU enhancements. Fourier transforms are also one of the first examples major companies such as Nvidia use to tout the computational gains of using GPUs.

This research aims to find out by how much the computation time of the Plane-Wave Time-Domain algorithm can be reduced by implementing it partly on a GPU. This includes research on the use of GPUs for basic operations but also for the Fast Fourier Transform, an implementation of the discreet version of the Fourier transform. Moreover, a more in-depth look is taken at FFTs as much research has been done into optimizing FFTs, including parallelisation. Furthermore, data has to be transferred from the CPU to the GPU and back, which adds additional computation time. It is therefore essential to also minimise the number of transferals.

The structure of this report is as follows. In chapter 2, the main problem is described for which the MOT and PWTD algorithms will be used. It also discusses the main research questions of the thesis. Chapter 3 discusses the MOT and PWTD algorithms themselves and discusses two related implementations for using the algorithms on the main problem described in chapter 2. Chapter 4 examines additional background information, which includes a closer look at Fourier transforms, GPU programming and the programming language Julia that was used for this thesis.

Chapter 5 then discusses the methodology for the experiments. This includes a look at CUDA programming in general and in Julia, a review of benchmarking in Julia and a quick discussion of the PWTD package created by thesis advisor Dr. Kristof Cools. Chapter 6 then describes the setup of the experiments for the research, with the results following in

chapter 7. A conclusion is drawn in the following chapter; recommendations and future work are discussed in chapter 9. Any additional information, such as programming code, has been added to the appendix.

Chapter 2

Problem Description

Many devices, such as antennas and aeroplanes, experience electromagnetic fields whilst they are operative. It is imperative for developers of such devices to know what happens when these fields interact with their devices and how the field might scatter after collision. The scattering of these fields can be modelled and solved mathematically.

Assume there is a scatterer bounded by a surface S as below, which corresponds to the aforementioned device. Though in this case it is portrayed as a two-dimensional flat surface, it can be any three-dimensional object. Next, consider an incident field $u^i(\mathbf{r}, t)$ that is fired upon the scatterer, which can for instance correspond to an electromagnetic field. It is assumed that u^i is temporally bandlimited by $\omega_{\max}$ (i.e. it is bandlimited in the time-domain). Figure 2.1 below sketches the described situation.

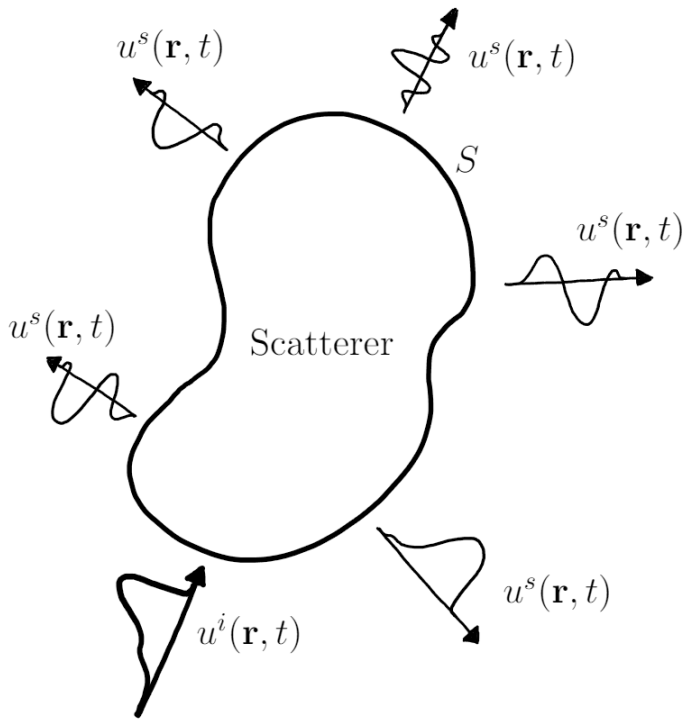


Figure 2.1: Surface scattering problem.

Here $\mathbf{r}$ is the directional vector in the xyz-domain and t the time. When u^i interacts

with S it creates a scattered field denoted by $u^s(\mathbf{r}, t)$. The total field is then $u(\mathbf{r}, t) = u^i(\mathbf{r}, t) + u^s(\mathbf{r}, t)$, the sum of the incident and scattered fields.

The goal is to determine the total field for all t. Note that the total field adheres to the wave equation

$$\nabla^2 u(\mathbf{r}, t) - \frac{\partial^2}{\partial t^2} \frac{1}{c^2} u(\mathbf{r}, t) = 0 \tag{2.1}$$

Using the boundary condition, which is assumed to be a Dirichlet boundary condition on S, u^s can be represented in terms of surface sources $q(\mathbf{r}, t)$ on S such that

$$u^s(\mathbf{r}, t) = \int_S d\mathbf{r}' \frac{\delta(t - R/c)}{4\pi R} * q(\mathbf{r}', t) \tag{2.2}$$

This can be solved numerically with a Marching-On-In-Time algorithm. Also denoted as the MOT algorithm, MOT can be accelerated by applying the Plane-Wave Time-Domain algorithm, which will be discussed in the next chapter. Both MOT and PWTD can become computationally expensive for large systems. To address this, parts of the PWTD algorithm will be accelerated by using GPUs to reduce the computation time needed to model scattering.

To narrow the scope of this research, the following thesis question has been determined:

> *By how much can the computational time of the Plane-Wave Time-Domain algorithm be reduced by using GPUs?*

To help answer this question, three sub-questions have been defined:

- *How can the performance of the FFT on GPUs best be optimised?*
- *How can the number of (and concurrently the computational costs and time from) transferals of data between CPU and GPU best be minimised?*
- *By what other means can the PWTD algorithm be optimised?*

Chapter 3

Plane-Wave Time-Domain Algorithm

This chapter discusses the Plane-Wave Time-Domain algorithm itself and how it can be used in conjunction with the problem from chapter 2. The chapter is built-up as follows: in the first section, the Finite Element Method is described together with a simple example. Section two expands upon this by introducing the Marching-on-in-Time algorithm. In section three, the PWTD algorithm for the problem from chapter 2 is introduced. Sections four and five discuss a two-level, respectively a multilevel implementation of the PWTD algorithm. The chapter concludes with a short note on Fourier transforms.

3.1 Finite Element Method

The Finite Element Method is a widely-used algorithm that can be applied in a wide range of problems, especially in for instance problems with complex geometries. In this section, FEM is discussed shortly to denote its main building blocks. This is combined with an example to illustrate the process of the algorithm. In general, a FEM problem can be sketched as follows:

Assume there is a surface with a boundary over which one wishes to determine the solution or evaluate a system of partial differential equations with a boundary value problem. The surface is denoted by Ω, its boundary by $\partial\Omega$. A vector $\mathbf{n}$ is the normal vector perpendicular to the boundary, with $||\mathbf{n}|| = 1$.

To work through the algorithm, a Boundary Value Problem (BVP) example is introduced for the function $u(x, y)$:

$$\begin{cases} -\Delta u + u = f(x,y) & \text{in } \Omega \\ \frac{\partial u}{\partial n} = g(x,y) & \text{on } \partial\Omega \end{cases}$$

First multiply the system by a test function φ, where φ is chosen as a continuous function. The following equation is then obtained:

$$(-\Delta u + u)\varphi = \varphi f$$

Next, integrate over Ω to obtain

$$\int_\Omega \varphi(-\Delta u + u)d\Omega = \int_\Omega \varphi f d\Omega$$

By applying Integration by Parts [19], the Divergence Theorem of Gauss [19] and the natural boundary condition, the weak formulation is obtained:

$$\int_\Omega \nabla u \nabla \varphi + u\varphi d\Omega = \int_\Omega f\varphi d\Omega + \int_{\partial\Omega} g\varphi d\Gamma$$

Galerkin's Method [20] is then applied to the weak formulation. Set

$$u(x,y) = \sum_{j=1}^{\infty} c_j\varphi_j(x,y) \simeq \sum_{j=1}^{n} c_j\varphi_j(x,y) = u^n(x,y)$$

Assume $\{\varphi_j(x,y)\}_{j=1}^n$ is a basis, so the φ_j are linearly independent.

Next, divide Ω and $\partial\Omega$ into meshpoints, as figure 3.1 below illustrates. The meshpoints are numbered.

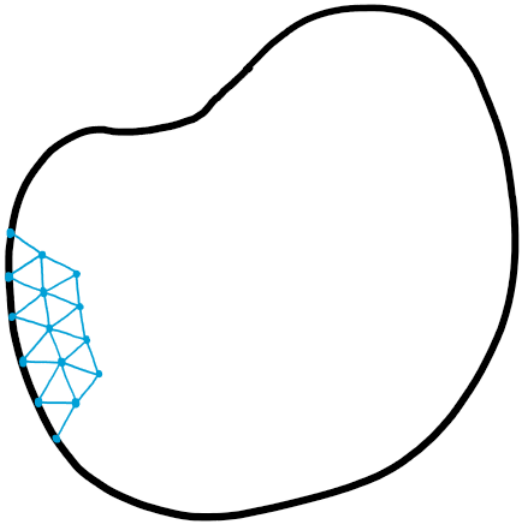


Figure 3.1: Surface divided into meshpoints

For each gridnode, assume that φ_i belongs to node i and φ_i is piecewise polynomial. Furthermore:

$$\varphi_i(x_j, y_j) = \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

The u determined above for the Galerkin method is then added into the weak formulation, where it follows that:

$$\sum_{j=1}^{n} c_j \left[\int_\Omega \nabla\varphi_i \nabla\varphi_j + \varphi_i\varphi_j d\Omega \right] = \int_\Omega \varphi_i f d\Omega + \int_\Omega \varphi_i g d\Gamma \qquad \forall i \in \{1, \ldots, n\}$$

The right-hand side can be set as b_i and the part between the []-brackets as S_{ij}; the system can then be rewritten as:

$$\sum_{j=1}^{n} S_{ij}c_j = b_i \tag{3.1}$$

Depending on the chosen elements, S_{ij} is then evaluated over each element before the final system is determined.

There are various methods that can help in the last step, such as the Holand and Bell Theorem over triangular elements, combined with Newton-Côtes. The type of element can also be varied; one could opt for extra nodes per triangle or for quadratic elements such as Taylor-Hood elements. The choice of basis functions also plays an important role. Note that this walk-through of the Finite Element Method is based upon notes for the courses WI4205 Applied Finite Elements and WI4450 Special Topics at the University of Technology Delft, both given by Fred Vermolen in 2018. More information can be found in [19].

3.2 Marching-On-In-Time Algorithm

The Marching-On-In-Time algorithm, or MOT-algorithm as it is commonly abbreviated to, works in the same way as the Finite Element Method, only then with added temporal basis functions to accommodate the time-variable in equations. These extra time functions are added to the Galerkin part of FEM. When compared to the previous section, the notation is changed so it is more aligned to the notation of the original problem. See also section 18.2 of [3].

Let $q(\mathbf{r}, t)$ be an unknown source density. $u^i(\mathbf{r}, t)$ is denoted as the field incident on the scatterer, which is temporally bandlimited by $\omega_{\max}$. The scatterer is bounded by a surface S as described in the problem description chapter. When the incident field interacts with the surface S, a scattered field is generated, denoted by $u^s(\mathbf{r}, t)$. The total field can then be denoted as $u(\mathbf{r}, t) = u^i(\mathbf{r}, t) + u^s(\mathbf{r}, t)$, which adheres to the wave equation:

$$\nabla^2 u(\mathbf{r}, t) - \frac{\partial^2}{\partial t^2} \frac{1}{c^2} u(\mathbf{r}, t) = 0$$

The incident and scattered fields can be represented in terms of similar surface sources $q(\mathbf{r}, t)$ that are located on S, such that

$$\begin{aligned} u^s(\mathbf{r}, t) &= \int_S d\mathbf{r}' \frac{\delta(t - R/c)}{4\pi R} * q(\mathbf{r}', t) \\ -u^i(\mathbf{r}, t) &= \int_S d\mathbf{r}' \frac{\delta(t - R/c)}{4\pi R} * q(\mathbf{r}', t) \qquad \forall \mathbf{r} \in S \end{aligned}$$

where

- $\delta(.)$ is the Dirac delta-function, such that for f on $\mathbb{R}^n$, we have for $a \in \mathbb{R}^n$ that $\int_{\mathbb{R}^n} \delta(x - a) f(x) dx = f(a)$
- $\mathbf{r}$ is the directional vector in the xyz-domain
- $R = |\mathbf{r} - \mathbf{r}'|$
- c is the wave speed in the medium of field S
- $*$ is the convolution operation, in the temporal domain

The convolution operation [25] can be defined as follows:
For functions $f, g : [0, \infty) \to \mathbb{R}$ the operator $*$ is defined as the integral of the product of two functions after one is reversed and shifted, such that

$$(f * g)(t) = \int_0^t f(\tau)g(t-\tau)d\tau$$

Equivalently

$$(f * g)(t) = \int_0^t f(t-\tau)g(\tau)d\tau$$

To solve this system numerically, represent $q(\mathbf{r}, t)$ in terms of the basis functions $f_n(\mathbf{r}), n = 1, \ldots, N_s$, which are spatial functions, and $T_i(t), i = 0, \ldots, N_t$, which are temporal functions. As with FEM, rewrite q as

$$q(\mathbf{r}, t) = \sum_{n=1}^{N_s} \sum_{i=0}^{N_t} q_{n,i} f_n(\mathbf{r}) T_i(t)$$

Here $q_{n,i}$ represent unknown expansion coefficients, like the c_j in the Finite Element Method.

Next, substitute $q(\mathbf{r}, t)$ in its basis functions form into the equation for $u^i(\mathbf{r}, t)$. Test the resulting equation at time $t = t_j = j\Delta t$ with test function $\tilde{f}_m(\mathbf{r})$ for $m = 1, \ldots, N_s$. In the same way as for equation 3.1, a matrix equation can be obtained for the system of the form:

$$\bar{Z}_0 Q_j = U_j^i - \sum_{k=1}^{j-1} \bar{Z}_k Q_{j-k} \tag{3.2}$$

where

- The m-th element of vector Q_j is given by $q_{m,j}$
- The m-th element of vector U_j^i is given by $-\int_S d\mathbf{r} \tilde{f}_m(\mathbf{r}) u^i(\mathbf{r}, t_j)$
- $$\bar{Z}_{k,mn} = \int_S d\mathbf{r} \tilde{f}_m(\mathbf{r}) \int_S d\mathbf{r}' f_n(\mathbf{r}') \left[\frac{\delta(t - R/c)}{4\pi R} * T_{j-k}(t) \right] |_{t=t_j} \tag{3.3}$$

With this equation the coefficients $q_{n,j}$ can be determined by starting at the first time step $j = 0$, from which the next value of the above equation can be determined per time step.

The setup is similar to that of the Finite Element Method, but with an added temporal basis function. The evaluation process does differ somewhat and can be computationally expensive. The operation on the right-hand side of equation 3.2 requires evaluation of field $u_s(\mathbf{r}, t)$ at $\mathcal{O}(N_s)$ points on S due to all the prior sources. Furthermore, it requires $\mathcal{O}(N_s^2)$ operations to complete. As this operation is repeated for all N_t timesteps, the final complexity is equal to $\mathcal{O}(N_t N_s^2)$.

To reduce the computational costs the Plane-Wave Time-Domain algorithm is applied. Do note that, unlike with FEM, only the boundary is turned into a mesh since in this case the problem is a Boundary Element problem, where only the boundary of the scatterer affects the system.

3.3 Plane-Wave Time-Domain Algorithm

The Plane-Wave Time-Domain algorithm is at the heart of this thesis research, as it is the algorithm to be accelerated by using GPUs. This section discusses the regular algorithm by first examining the necessary notation and subsignal division. After that, the general definition of a plane-wave is considered as well as a plane-wave representation for the scatter-problem. Finally, the PWTD decomposition is implemented and reviewed. The steps in [3] are followed since it is the main source for the algorithm; additional explanation has been added to certain parts for clarification. The same notation of [3] is also used here so the source material can easily be referenced.

3.3.1 Subsignals

The Plane-Wave Time-Domain algorithm calls for all source signals to be of a finite duration. This works for the original problem from chapter 2, as the source signal $q(\mathbf{r}, t)$ can be divided into subsignals $q_l(\mathbf{r}, t)$ of an equal duration, such that

$$q(\mathbf{r}, t) = \sum_{l=1}^{L} q_l(\mathbf{r}, t). \tag{3.4}$$

Each subsignal q_l is zero outside of an interval $(l-1)T_s \leq t \leq lT_s$, where T_s is the duration of the subsignals. The field radiated by source q_l can then be denoted by $u_l(\mathbf{r}, t)$; the total field can then be expressed by summing up over all the intervals:

$$u_l(\mathbf{r}, t) = \sum_{l=1}^{L} u_l(\mathbf{r}, t) \tag{3.5}$$

The relation between the subsignal and subfield can then be denoted as

$$u_l(\mathbf{r}, t) = \int_S d\mathbf{r}' \frac{\delta(t - R/c)}{4\pi R} * q_l(\mathbf{r}, t) \tag{3.6}$$

3.3.2 Plane-Wave Decomposition

The first step of the PWTD algorithm is to determine a fitting plane wave representation for the transient wave fields in 3.6. This step is motivated by the use of frequency domain fast multipole schemes, which are discussed at the end of this section. Before that, the definition of a plane wave must be considered.

A wave can be described with the following wavefunction [13]:

$$\psi(x, t) = A\cos(kx - \omega t + \phi)$$

where A is the amplitude, k the wavenumber, ω the angular frequency and ϕ the phase angle of the wave. This type of wavefunction is a one-dimensional plane wave. It is one-dimensional because it only depends on the Cartesian coordinate x. Moreover, the wave maxima are located at $kx - \omega t + \phi = 2j\pi$ where j is an integer. These wave maxima are a series of parallel planes normal to the x-axis and spaced a distance $\lambda = 2\pi/k$ apart. They propagate along the x-axis at a velocity $v = \omega/k$.

In the same way a three-dimensional plane wave can be described. Consider then the wavefunction

$$\psi(x, y, z, t) = A\cos(\mathbf{k} \cdot \mathbf{r} - \omega t + \phi)$$

where $\mathbf{r} = (x, y, z)$ and $\mathbf{k} = (k_x, k_y, k_z) \cdot \mathbf{n}$, where $\mathbf{n}$ is an arbitrary unit vector. The planes are normal to $\mathbf{n}$ and are located at the wave maxima $\mathbf{k} \cdot \mathbf{r} - \omega t + \phi = 2j\pi$.

Using this, a plane wave decomposition for 3.6 can be explored. [3] posits the following plane wave representation for $u_l(\mathbf{r}, t)$, denoted as $\tilde{u}_l(\mathbf{r}, t)$:

$$\tilde{u}_l(\mathbf{r}, t) = -\frac{\partial_t}{8\pi^2 c} \int_0^{\theta_{\text{int}}} d\theta \sin\theta \int_0^{2\pi} d\phi \int_S d\mathbf{r}' \delta \left[t - \hat{\mathbf{k}} \cdot (\mathbf{r} - \mathbf{r}') \Big/ c \right] * q_l(\mathbf{r}', t) \tag{3.7}$$

Here $\hat{\mathbf{k}} = \hat{\mathbf{x}} \sin\theta\cos\phi + \hat{\mathbf{y}} \sin\theta \sin\phi + \hat{\mathbf{z}} \cos\theta$ is a unit direction vector. Via the integral over S the source distributions q_l are projected onto a plane wave travelling in the $\hat{\mathbf{k}}$ direction. θ_{int} in the first integral is the upper limit of θ.

Next, if θ_{int} is set equal to π, then $\tilde{u}_l$ can be conveyed as a superposition of plane waves that move in all directions. The necessary relation between u_l and $\tilde{u}_l$ can then be found by integrating over θ and ϕ in equation 3.7. To do this efficiently, equation 3.7 is transformed into a different coordinate system $(x', y', z') \equiv (\rho', \theta', \phi')$ where the z'-axis is aligned with $\mathbf{R} = \mathbf{r} - \mathbf{r}'$. This coordinate system transformation is motivated by the slant stack transform operation [7]. Figure 18.3 of [3] shows how the translation to the new coordinate system works in a 2D system; it has been added below for reference.

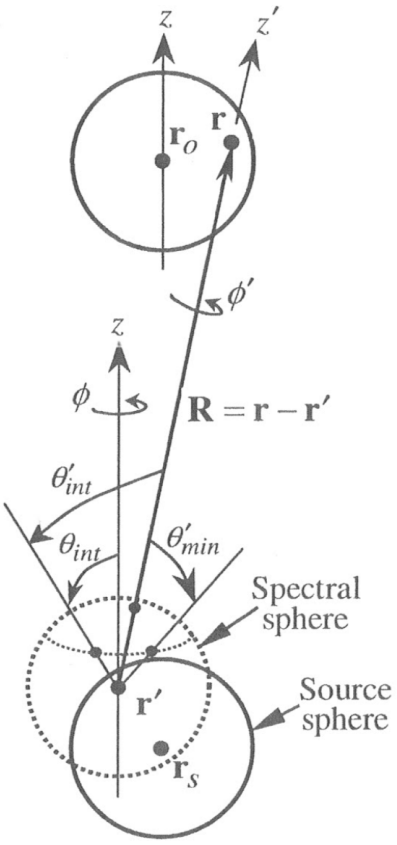


Figure 3.2: Figure 18.3a of [3]: Translation into new coordinate system.

In this new coordinate system the upper limit on θ', θ'_{int}, is a function of $\phi', \mathbf{r}$ and $\mathbf{r}'$, so $\theta'_{\text{int}}(\phi', \mathbf{r}, \mathbf{r}')$. Then

$$\tilde{u}_l(\mathbf{r}, t) = -\frac{\partial_t}{8\pi^2 c} \int_S d\mathbf{r}' \int_0^{2\pi} d\phi' \int_0^{\theta'_{\text{int}}(\phi', \mathbf{r}, \mathbf{r}')} d\theta' \sin\theta' \delta\left(t - \hat{\mathbf{k}}' \cdot \mathbf{R}' \Big/ c\right) * q_l(\mathbf{r}', t) \tag{3.8}$$

Here $\hat{\mathbf{k}}' = \hat{\mathbf{x}}' \sin\theta' \cos\phi' + \hat{\mathbf{y}}' \sin\theta' \sin\phi' + \hat{\mathbf{z}}' \cos\theta'$ and $\mathbf{R}' = \hat{\mathbf{z}}'|\mathbf{R}|$. Furthermore, assume $\theta_{\text{int}} > \cos^{-1}(\hat{\mathbf{z}} \cdot \mathbf{R}/R)$. By defining $R = |\mathbf{R}|$ and using the fact that $\hat{\mathbf{k}}' \cdot \mathbf{R}' = R\cos\theta'$, one can set $\tau = R\cos\theta'/c$ such that, in conjunction with equation 3.8, it follows that

$$\begin{aligned}
\tilde{u}_l(\mathbf{r}, t) &= -\int_S d\mathbf{r}' \int_0^{2\pi} d\phi' \int_{-(R/c)\cos\theta'_{\text{int}}}^{R/c} d\tau \frac{\partial_t \delta(t-\tau)}{8\pi^2 R} * q_l(\mathbf{r}', t) \\
&= \int_S d\mathbf{r}' \frac{\delta(t - R/c)}{4\pi R} * q_l(\mathbf{r}', t) - \int_S d\mathbf{r}' \frac{\delta(t + R\cos\theta'_{\text{int}}/c)}{4\pi R} * q_l(\mathbf{r}', t) \\
&= u_l(\mathbf{r}, t) - \int_S d\mathbf{r}' \frac{\delta(t + R\cos\theta'_{\text{int}}/c)}{4\pi R} * q_l(\mathbf{r}', t)
\end{aligned} \tag{3.9}$$

If again $\theta_{\text{int}} = \pi$, 3.9 reduces to

$$\tilde{u}_l(\mathbf{r}, t) = u_l(\mathbf{r}, t) - \int_S d\mathbf{r}' \frac{\delta(t + R/c)}{4\pi R} * q_l(\mathbf{r}', t) \tag{3.10}$$

The second term in equations 3.9 and 3.10 is called the ghost term, which is anticausal and thus has outputs and states that do not depend on past inputs. This also means that the ghost term is seen by the observer before the source signal even exists. By using a causality trick, one can time-gate out the ghost signal by setting the subsignal length to $T_s < R/c$, as this ensures the ghost signal and true signal never overlap.

Next, the question of why the derivation of equation 3.7 is need to further the advancement of a speedy algorithm that can calculate the transient fields is discussed. Consider a source distribution restricted to a sphere of radius R_s. Furthermore, consider a collection of observers that are also located in a (different) sphere of the same radius. The centres of the source- and observer-sphere correspond to $\mathbf{r}_s$ and $\mathbf{r}_o$, respectively. The vector connecting the two centres is denoted by $\mathbf{R}_c = \mathbf{r}_o - \mathbf{r}_s$. Assume $R_c = |\mathbf{R}_c| > 2R_s$, so the distance between the two centres is larger than the radii of both spheres added together and thus the spheres don't overlap. Also note that the vector $\mathbf{r} - \mathbf{r}' = (\mathbf{r} - \mathbf{r}_o) - \mathbf{R}_c - (\mathbf{r}' - \mathbf{r}_s)$. Figure 18.5 from [3] visualises the vectors just described; for easy reference, the image has been added in figure 3.3.

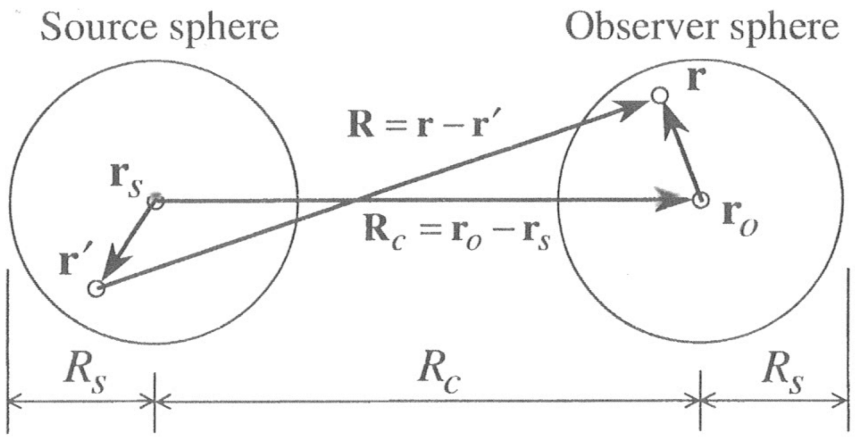


Figure 3.3: Figure 18.5 from [3] showing the vectors of the PWTD algorithm.

Equation 3.7 can then be rewritten into

$$\begin{aligned}\tilde{u}_l(\mathbf{r},t) = &\int d^2\hat{\mathbf{k}}\delta\left[t-\hat{\mathbf{k}}\cdot(\mathbf{r}-\mathbf{r}_o)\Big/c\right] * \mathcal{T}(\hat{\mathbf{k}},\mathbf{R}_c,t)\\ &* \int_S d\mathbf{r}'\delta\left[t+\hat{\mathbf{k}}\cdot(\mathbf{r}'-\mathbf{r}_s)\Big/c\right] * q_l(\mathbf{r}',t)\end{aligned} \tag{3.11}$$

where

$$\int d^2\hat{\mathbf{k}} = \int_0^{\pi} d\theta \sin\theta \int_0^{2\pi} d\phi \qquad \text{integration over the unit sphere}$$

$$\mathcal{T}(\hat{\mathbf{k}},\mathbf{R}_c,t) = -\frac{\partial_t}{8\pi^2 c}\delta\left(t-\hat{\mathbf{k}}\cdot\mathbf{R}_c\Big/c\right) \qquad \text{translation function}$$

Now a three-stage implementation of 3.11 is used to evaluate $\tilde{u}_l$, where each convolution is determined separately.

1. First perform the rightmost integration and convolution of 3.11, which is equal to

$$q_l^{\text{out}}(\hat{\mathbf{k}},t) = \int_S d\mathbf{r}'\delta\left[t+\hat{\mathbf{k}}\cdot(\mathbf{r}'-\mathbf{r}_s)\Big/c\right] * q(\mathbf{r},t)_l \tag{3.12}$$

   This operation is known as the slant stack transform (SST) [7] of q_l. The quantities q_l^{out} can be interpreted as rays leaving the source sphere in direction $\hat{\mathbf{k}}$.

2. Next, perform the second integration and convolution, so evaluate

$$q_l^{\text{in}}(\hat{\mathbf{k}},t) = \mathcal{T}(\hat{\mathbf{k}},\mathbf{R}_c,t) * q_l^{\text{out}}(\hat{\mathbf{k}},t) \tag{3.13}$$

   Each outgoing ray is translated by the operator $\mathcal{T}$ from the source sphere to the observer sphere. The q_l^{in} can be seen as rays entering the observer sphere.

3. Lastly, perform the final integration and convolution:

$$\tilde{u}_l(\mathbf{r},t) = \int d^2\hat{\mathbf{k}}\delta\left[t-\hat{\mathbf{k}}\cdot(\mathbf{r}-\mathbf{r}_o)\Big/c\right] * q_l^{\text{in}}(\hat{\mathbf{k}},t) \tag{3.14}$$

   This action shifts all the projections of the incoming rays correctly to the observer.

Setting $T_s < (R_c - 2R_s)/c$ ensures that

$$u_l(\mathbf{r}, t) = \begin{cases} 0 & \text{for} \quad t < lT_s \\ \tilde{u}_l(\mathbf{r}, t) & \text{for} \quad t \geq lT_s \end{cases}$$

and it ensures that the ghost signal is timed out correctly.

Fast Multipole Method

As a side-note, one recognises the Fast Multipole Method in the above derivation of the plane wave decomposition of the algorithm, per [8]. The method is started from the matrix Z of 3.2. For scattering problems, the matrix elements can be written as

$$Z_{nn'} = -i \int_S d^2\mathbf{x} \int_S d^2\mathbf{x}' f_n(\mathbf{x}) \frac{e^{ik|\mathbf{x}-\mathbf{x}'|}}{4\pi|\mathbf{x}-\mathbf{x}'|} f_n(\mathbf{x}') \tag{3.15}$$

Here f_n are again the basis functions, which is comparable to 3.3. The following elementary identities are used to rewrite 3.15:

$$\frac{e^{ik|\mathbf{X}+\mathbf{d}|}}{|\mathbf{X}+\mathbf{d}|} = ik \sum_{l=0}^{\infty} (-1)^l (2l+1) j_l(kd) h_l^{(1)}(kX) P_l(\hat{d} \cdot \hat{X}) \tag{3.16}$$

$$\int d^2\hat{k} e^{i\mathbf{k}\cdot\mathbf{d}} P_l(\hat{k} \cdot \hat{X}) = 4\pi i^l j_l(kd) P_l(\hat{d} \cdot \hat{X}) \tag{3.17}$$

It follows that

$$Z_{nn'} \approx \frac{k}{(4\pi)^2} \int_S d^2\mathbf{x} f_n(\mathbf{x}) \int_S d^2\mathbf{x}' f_{n'}(\mathbf{x}') \int d^2\hat{k} e^{i\mathbf{k}\cdot(\mathbf{x}-\mathbf{x}'-\mathbf{X})} \mathcal{T}_L(kX, \hat{k} \cdot \hat{X}) \tag{3.18}$$

where

$$\mathcal{T}_L(\kappa, \cos\theta) \equiv \sum_{l=0}^{L} t^l (2l+1) h_l^{(1)}(\kappa) P_l(\cos\theta) \tag{3.19}$$

The Legendre polynomials and Bessel functions found in 3.18 and 3.19 are discussed below.

Bessel Functions

The Bessel functions [34] are the solutions of Bessel's differential equation:

$$z^2 \frac{d^2w}{dz^2} + z\frac{dw}{dz} + (z^2 - m^2)w = 0 \tag{3.20}$$

Here m is an arbitrary complex number, which is also known as the order of the Bessel function.

There are two kinds of Bessel functions, of the first kind and of the second kind. These are, in the case of cylindrical coordinates,

$$J_m(z) = \left(\frac{1}{2}z\right)^m \sum_{k=0}^{\infty} (-1)^k \frac{\left(\frac{1}{4}z^2\right)^k}{k!\,\Gamma(m+k+1)} \tag{3.21}$$

$$Y_m(z) = \frac{J_m(z)\cos(m\pi) - J_{-m}(z)}{\sin(m\pi)} \tag{3.22}$$

Here Γ is the gamma-function, which is equal to

$$\Gamma(n) = (n-1)!$$

In the case of spherical coordinates and assuming that m is a positive integer, the Bessel functions become

$$\mathbf{j}_m(z) = \sqrt{\frac{1}{2}\pi/z} J_{m+\frac{1}{2}}(z) \tag{3.23}$$

$$\mathbf{y}_m(z) = \sqrt{\frac{1}{2}\pi/z} Y_{n+\frac{1}{2}}(z) \tag{3.24}$$

Legendre Polynomials

For a variable x, the Legendre polynomial [2] can be written as

$$P_n(x) = \frac{1}{2^n} \sum_{k=0}^{n} \binom{n}{k}^2 (x-1)^{n-k}(x+1)^k \tag{3.25}$$

Legendre polynomials can be useful for expanding $1/r$-potentials and for multipole expansions.

3.3.3 Implementation Issues

Reference [3] also highlights some implementation issues that need to be addressed before the algorithm can be implemented correctly. These issues are related to spatial integration, spectral integration and subsignal temporal representation.

Before discussing these, another operation to consider is bandlimiting. A bandlimited signal is a signal in which a particular band of frequencies are present. In other words, it is the limiting of a signal's Fourier Transform to zero above a certain finite frequency.

Spatial Integration

Whilst performing the integration of 3.12 and 3.3, make sure the correct quadrature rules are used; this depends entirely on the discretization function used. The corresponding quadrature rules are discussed with Spectral Integration.

Spectral integration

To numerically evaluate the fields using 3.11, the correct quadrature rules have to be used to perform the integration over the unit sphere. Three key observations help when evaluating 3.11 numerically.

1. If the translation function is excluded from the integral in 3.11, it is equivalent to the time-dependent radiation pattern of a source distribution contained in a sphere of radius $2R_s$ [17]. If it is assumed that $q_l(\mathbf{r},t)$ is temporally bandlimited to a $\omega_s > \omega_{\max}$, this piece of the integral can be defined in terms of spherical harmonics (see the previous section) $Y_{km}(\theta,\phi)$ as

$$\begin{aligned} g(\hat{\mathbf{k}},\mathbf{r},t) &= \delta\left[t-\hat{\mathbf{k}}\cdot(\mathbf{r}-\mathbf{r}_o)\Big/c\right] * \int_S d\mathbf{r}'\delta\left[t+\hat{\mathbf{k}}\cdot(\mathbf{r}'-\mathbf{r}_s)\Big/c\right] * q_l(\mathbf{r}',t) \\ &= \int_S d\mathbf{r}'\delta\left[t+\hat{\mathbf{k}}\cdot[(\mathbf{r}'-\mathbf{r}_s)-(\mathbf{r}-\mathbf{r}_o)]\Big/c\right] * q_l(\mathbf{r}',t) \\ &= \sum_{k=0}^{K}\sum_{m=-k}^{k} g_{km}(\mathbf{r},t)Y_{km}(\theta,\phi) \end{aligned} \tag{3.26}$$

   Set $K = \lceil \chi_1 2R_s\omega_s/c \rceil$ where $\chi_1 > 1$ is an excess bandwidth factor that provides fast convergence for the series in 3.26.

2. The translation function $\mathcal{T}$ in 3.11 is only a function of the angle θ', where θ' lies in between the vectors $\hat{\mathbf{k}}$ and $\mathbf{R}_c$. Therefore it can be denoted either in terms of Legendre polynomials in θ' or in terms of spherical harmonics in (θ,ϕ), as

$$\mathcal{T}(\hat{\mathbf{k}},\mathbf{R}_c,t) =$$

$$= \begin{cases} -\frac{\partial_t}{16\pi^2 R_c}\sum_{k=0}^{\infty}(2k+1)P_k(ct/R_c)P_k(\cos\theta') & |t| \le R_c/c \\ 0 & \text{elsewhere} \end{cases} \tag{3.27}$$

$$= \begin{cases} \sum_{k=0}^{\infty}\sum_{m=-k}^{k}\mathcal{T}_{km}(\mathbf{R}_c,t)Y_{km}(\theta,\phi) & |t| \le RC/c \\ 0 & \text{elsewhere} \end{cases} \tag{3.28}$$

3. Spherical harmonics posses an orthogonality property, hence the terms in 3.28 for which $k > K$ do not contribute to the result whilst integrating $g(\hat{\mathbf{k}},\mathbf{r},t) * \mathcal{T}(\hat{\mathbf{k}},\mathbf{R}_c,t)$ over the unit sphere. Because of this property the first sum in 3.28 can be stopped at $k = K$. Furthermore, the translation $\mathcal{T}$ can then be replaced by the truncated version $\bar{\mathcal{T}}(\hat{\mathbf{k}},\mathbf{R}_c,t)$ in all previous expressions, where $\bar{\mathcal{T}}$ is denoted by

$$\bar{\mathcal{T}}(\hat{\mathbf{k}},\mathbf{R}_c,t) = \begin{cases} -\frac{\partial_t}{16\pi^2 R_c}\sum_{k=0}^{K}(2k+1)P_k(ct/R_c)P_k(\cos\theta') & |t| \le R_c/c \\ 0 & \text{elsewhere} \end{cases} \tag{3.29}$$

From these three observations it follows that the integral in 3.11 can be denoted by multiplying two distinct functions. These two functions can be expressed in terms of spherical harmonics $Y_{km}, k = 0,\ldots,K; m = -k,\ldots,k$. The integral can then be evaluated by using a $(2K+1)$-point trapezoidal rule in the ϕ-direction and a $(K+1)$-point Gauss-Legendre quadrature in the θ-direction.

The Trapezoidal Rule [28] for an integral is defined as follows:

$$\int_a^b f(x)dx \approx \sum_{j=1}^{N}\frac{f(x_{j-1})+f(x_j)}{2}\Delta x_j \tag{3.30}$$

where x_j are the quadrature points.

The Gaus-Legendre quadrature is defined as follows:

When integrating over the interval $[-1, 1]$, the Gauss-Legendre quadrature has the form

$$\int_{-1}^{1} f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i) \tag{3.31}$$

Here n is the number of gridpoints, w_i are the quadrature weights and x_i are the roots of the n-th Legendre Polynomial, see 3.25. The weights are then given by the formula

$$w_i = \frac{2}{(1 - x_i^2)[P_n'(x_1)]^2}$$

This together yields the following expression for $\tilde{u}_l$:

$$\begin{aligned}\tilde{u}_l(\mathbf{r}, t) = &\sum_{p=0}^{K} \sum_{q=-K}^{K} w_{pq} \delta \left[t - \hat{\mathbf{k}}_{pq} \cdot (\mathbf{r} - \mathbf{r}_o) \Big/ c \right] * \bar{\mathcal{T}}(\hat{\mathbf{k}}_{pq}, \mathbf{R}_c, t) \\ &* \int_S d\mathbf{r}' \delta \left[t + \hat{\mathbf{k}}_{pq} \cdot (\mathbf{r}' - \mathbf{r}_s) \Big/ c \right] * q_l(\mathbf{r}', t)\end{aligned} \tag{3.32}$$

where

$$\begin{aligned}w_{pq} &= \frac{4\pi(1 - \cos^2 \theta_p)}{(2K+1)[(K+1)P_K(\cos(\theta_p)]^2} \\ \hat{\mathbf{k}}_{pq} &= \hat{\mathbf{x}} \sin\theta_p \cos\phi_q + \hat{\mathbf{y}} \sin\theta_p \sin\phi_q + \hat{\mathbf{z}} \cos\theta_p \\ \phi_q &= 2\pi q/(2K+1) \\ \theta_p &\text{ is the } (p+1)^{\text{th}} \text{ zero of } P_{K+1}(\cos\theta)\end{aligned} \tag{3.33}$$

Subsignal temporal representation

In part 1 of Spectral integration, the assumption was made that the subsignals were temporally bandlimited. However, it is also necessary for each subsignal to be timelimited, which is not possible due to the temporal bandlimiting. Luckily this can easily be fixed. The entire signal is bandlimited to $\omega_{\max}$ and can be divided into subsignals that are both bandlimited to $\omega_s = \chi_0 \omega_{\max}$ where $\chi_0 > 1$ and approximately time limited. This can be done by using the correct local interpolation functions.

It is known that $q(\mathbf{r}, t)$ is temporally bandlimited. It can be locally bandlimited by using temporally bandlimited and approximately time limited functions as

$$q(\mathbf{r}, t) \cong \sum_{k=1}^{N_t} q(\mathbf{r}, k\Delta_t)\psi_k(t). \tag{3.34}$$

Here Δ_t is the time step and $\psi_k(t)$ is a bandlimited interpolant. A near optimal ψ_k is given by

$$\psi_k(t) = \frac{\omega_+}{\omega_s} \frac{\sin(\omega_+(t-k\Delta_t))}{\omega_+(t-k\Delta_t)} \frac{\sinh\left(\omega_- p_t \Delta_t \sqrt{1-[(t-k\Delta_t)/p_t\Delta_t]^2}\right)}{\sinh\left(\omega_- p_t \Delta_t\right)\sqrt{1-[(t-k\Delta_t)/p_t\Delta_t]^2}} \tag{3.35}$$

where

$\omega_s = \pi/\Delta_t = \chi_0 \omega_{\max}$
$\chi_0 > 1$ is the oversampling ratio
$\omega_\pm = (\omega_s \pm \omega_{\max})/2$
p_t integer that defines the approximate duration of the interpolation function.

If this is plotted for $\Delta_t = 0.01, \chi_0 = 3, p_t = 2$ the following two figures follow for $k = 20$ and $k = 50$:

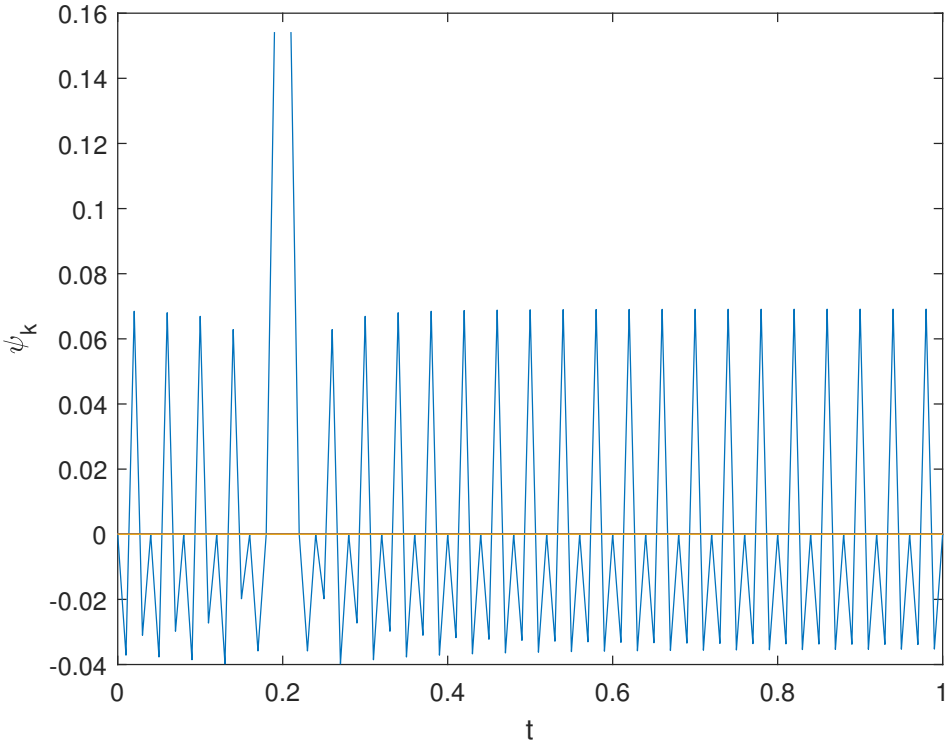


Figure 3.4: Bandlimited interpolant for $k = 20$

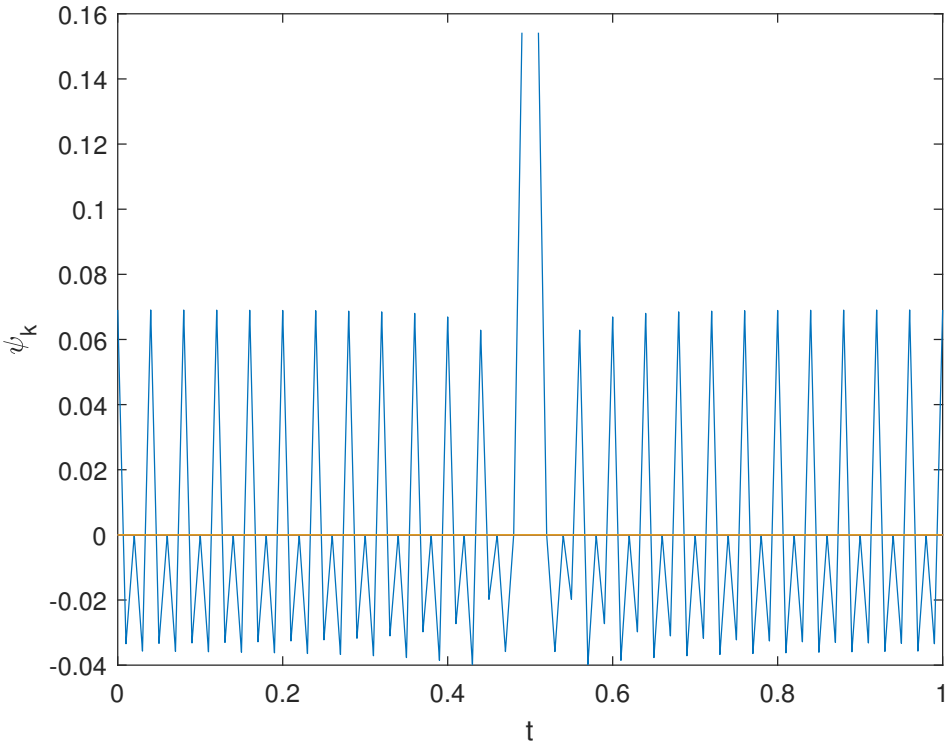


Figure 3.5: Bandlimited interpolant for $k = 50$

The interpolant oscillates closely around the t-axis, where it eventually creates a massive spike at $t/\Delta_t = k$; figure 3.5, for instance, has 100 datapoints and $k = 50$, so naturally its spike is halfway along the axis.

From 3.34 it follows that the signal can be divided into subsignals q_l given by

$$q_l(\mathbf{r}, t) = \sum_{k=l\ M_t}^{(l+1)M_t-1} q(\mathbf{r}, k\Delta_t)\psi_k(t) \tag{3.36}$$

Here each subsignal q_l is defined in terms of M_t samples of the signal q but spans $M_t' = M_t + 2p_t$ time steps. This means that each subsignal $q_l(\mathbf{r}, t)$ is created from the samples of $q(\mathbf{r}, t)$ in an interval of length $T_s = M_t\Delta_t$ but that the duration of each subsignal $q_l(\mathbf{r}, t)$ is of duration $T_s' = M_t'\Delta_t = (M_t + 2p_t)\Delta_t$. Therefore, neighbouring subsignals overlap by $2p_t$ samples. Together with the function ψ_k being bandlimited to ω_s, this results in the required time limitation.

It can be easily seen that χ_0 and χ_1 determine the accuracy of this three-step PWTD algorithm. [3] has run various tests to verify this, the results of these tests can be found at the end of section 18.3.

3.4 Two-Level Algorithm

The most computationally expensive part of the MOT algorithm is the computation of 3.2, especially the sum on the right-hand side. The idea is to combine the MOT algorithm with the PWTD algorithm to create a Plane-Wave Time-Domain-enhanced Marching-on-in-Time algorithm. The main concept behind this algorithm is quite simple: the scatterer is first divided into subscatterers. The contributions of all the subscatterers that are nearby are then determined directly using MOT, the other contributions are determined using the PWTD scheme. This section will discuss a two-level implementation of the PWTD-enhanced MOT algorithm; section five will build upon this by introducing a multilevel scheme.

For both implementations the steps are the same:

1. Create a partitioning of the scatterer and its boundary
2. Transform the waves
3. Perform the algorithm

3.4.1 Partitioning

To describe the partitioning of the two-level algorithm, consider a cubical volume around the scatterer S. This cube is divided into smaller cubes that are all of equal size and each fits into a sphere of radius R_s. Set N_g as the total number of non-empty boxes and N_s as the number of spatial basis functions for the MOT scheme. (Recall from the MOT scheme that N_s is the total number of spatial basis functions, and N_t the total number of temporal basis functions.) The set of spatial basis functions in a non-empty box is called a group. Of importance is the average number of spatial basis functions in each group, which is equal to $M_s = N_s/N_g$, where $M_s \propto (R_s\omega_{\max}/c)^2$. Remember that $\omega_{\max}$ is the bandlimiting factor for $u^i(\mathbf{r}, t)$.

The non-empty boxes are numbered $1, \ldots, N_g$. Define $\gamma, \gamma' = 1, \ldots, N_g$ as corresponding groups in the numbered non-empty boxes, where γ denotes the source group and γ' denotes the observer group. Each group pair (γ, γ') is then either near field or far field. The distinction between the two depends on the distance between the group centers. If this distance is less than a preset distance $R_{c,\min}$ the pair is near field, if it is larger than $R_{c,\min}$ it is far field. Assume that $R_{c,\min} = \xi R_s$, where $3 \leq \xi \leq 6$ is chosen beforehand.

An example of the two-level partitioning can be found in figure 3.6. Note that figures 3.6 and 3.7 are reconstructions of the images in figure 18.9 in [3] where the images have been redrawn and coloured in. Next, assume that the box with the cross in it is box 1 and is the observer box; furthermore, assume that $R_s = 3$. Then group pair $(2, 1)$, where box 2 is the box immediately above box 1, is a near field pair. Group pair $(11, 1)$, where box 11 is the eleventh box after tracing the boundary upwards from box 1, is a far field pair.

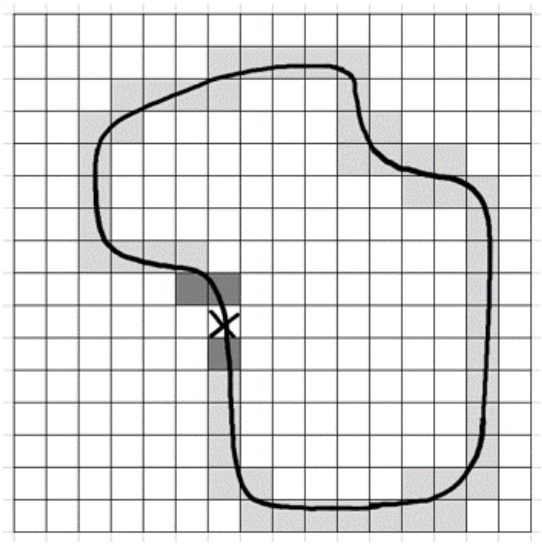


Figure 3.6: Example of two-level subdivision

3.4.2 Transforming

To avoid ghost signals in the PWTD, a correct signal duration must be defined. To that end, the fundamental subsignal duration T_s can be defined as below; subsignal durations that are needed in the determination of fields related to other far field pairs (γ, γ') are denoted as $T_{s,\gamma\gamma'}$:

$$
\begin{aligned}
&T_s = M_t \Delta_t && (3.37)\\
&M_t = \min_{\gamma,\gamma'} \{\lfloor (R_{c,\gamma\gamma'} - 2R_s)/(c\Delta_t) \rfloor\}\\
&R_{c,\gamma\gamma'} \text{ the distance between the centers of group-pair } (\gamma, \gamma')\\
&T_{s,\gamma\gamma'} = M_{t,\gamma\gamma'} \Delta_t && (3.38)\\
&M_{t,\gamma\gamma'} = M_t \lfloor (R_{c,\gamma\gamma'} - 2R_s)/(cT_s) \rfloor
\end{aligned}
$$

T_s is then the maximum duration of a subsignal without obtaining a ghost signal between the nearest groups of far-field pairs of blocks. $T_{s,\gamma\gamma'}$ are integer multiples of T_s per their definition; this allows these values to be reused easily.

To evaluate the convolution for 3.13, define the Fourier Transform (see chapter 4.1) of the translation function:

$$
\begin{aligned}
\mathcal{F}\left\{\bar{\mathcal{T}}(\hat{\mathbf{k}}, \mathbf{R}_{c,\gamma\gamma'}, t)\right\} &= \int_{-\infty}^{\infty} \bar{\mathcal{T}}(\hat{\mathbf{k}}, \mathbf{R}_{c,\gamma\gamma'}, t) e^{-i\omega t} dt\\
&= -\frac{i\omega}{8\pi^2 c} \sum_{k=0}^{K} (2k+1)(-i)^k \mathbf{j}_k(\omega R_{c,\gamma\gamma'}/c) P_k(\cos\theta') \qquad (3.39)
\end{aligned}
$$

Here $\mathbf{j}_k$ is the spherical Bessel function, see also 3.23. P_k is a Legendre polynomial as in 3.25. For any sphere pair (γ, γ') these functions can easily be constructed by using the normalised translation function

$$\tilde{\mathcal{T}}(\theta', \Omega) = R_{c,\gamma\gamma'} \left\{ \bar{\mathcal{T}}(\hat{\mathbf{k}}, \mathbf{R}_{c,\gamma\gamma'}, t) \right\} \tag{3.40}$$

$$= -\frac{i\Omega}{8\pi^2} \sum_{k=0}^{K} (2k+1)(-i)^k \mathbf{j}_k(\Omega) P_k(\cos\theta') \tag{3.41}$$

Here θ' is the ray angle and $\Omega = \omega R_{c,\gamma\gamma'}/c$ the normalised frequency. These normalised translation functions are then saved in a table for later use.

3.4.3 Algorithm

Using everything defined before, a two-level algorithm can now be defined to determine the sum on the right-hand side of equation 3.2. The parts the sum that correspond to the near and far field pairs are determined in separate steps, starting with the near field evaluations.

1. *Evaluation of the near field contributions*
   Evaluate the following sum during every time step:

$$\sum_{k=1}^{j-1} \bar{\mathbf{Z}}_k^{\gamma\gamma'} \mathbf{Q}_{j-k}^{\gamma'} \tag{3.42}$$

   Here $\bar{\mathbf{Z}}_k^{\gamma\gamma'}$ denotes a submatrix of $\bar{\mathbf{Z}}_k$ from 3.2 that relates fields over group γ to sources in group γ'. $\mathbf{Q}_{j-k}^{\gamma'}$ is a vector with values defined by the $q_{n,j-k}$ for all sources n in group γ'.

2. *Evaluation of the far field contributions*
   Follow the three steps of the Plane-Wave Time-Domain algorithm.

   (a) *Construction of outgoing rays*
   A set of outgoing rays is constructed for each group, generated by subsignals of duration T_s, every M_t time steps. By convolving the subsignal related to the spatial basis function $f_n(\mathbf{r})$, the contribution from f_n to the outgoing ray travelling in the direction of $\hat{\mathbf{k}}_{pq}$ can be evaluated with

$$V_n^+(\hat{\mathbf{k}}_{pq}, t) = \int_S d\mathbf{r}' \delta \left[t + \hat{\mathbf{k}}_{pq} \cdot (\mathbf{r}' - \mathbf{r}_c) \Big/ c \right] f_n(\mathbf{r}') \tag{3.43}$$

   Here $\mathbf{r}_c$ corresponds to the center of the group that encapsulates the basis function f_n.

   (b) *Translation*
   The outgoing rays are translated to incoming rays from group γ to group γ'. The translation is described in the following steps:

      i. By using the rays stored in 2a, an outgoing ray can be constructed by connecting these rays. The Fourier transform of the translation function is also computed, in anticipation of the coming convolution.
      ii. The spectrum of the applicable function is determined from the $\tilde{\mathcal{T}}$ table through local interpolation.

iii. The translation function spectrum and outgoing ray spectrum are multiplied.

iv. The inverse Fourier transform is applied to transform the result back into the time domain. The obtained rays are superimposed onto incoming rays.

(c) *Projection of incoming rays onto observers*
By convolving the incoming rays with

$$\tilde{V}_n^-(\hat{\mathbf{k}}_{pq}, t) = \int_S d\mathbf{r}' \delta \left[t - \hat{\mathbf{k}}_{pq} \cdot (\mathbf{r}' - \mathbf{r}_c) \Big/ c \right] \tilde{f}_n(\mathbf{r}') \tag{3.44}$$

the field at the nth observer can be evaluated. The step is completed by performing the spherical integration.

Complexity

One important factor to look at is the complexity of the two-level algorithm. As determined in [3], the cost in step 1 is equal to $\mathcal{O}(N_t N_s M_s)$ and in step 2 it is equal to $\mathcal{O}(N_t N_s^2 M_s^{-1} \log N_s)$. The complexity can be minimised by setting $M_s \propto \sqrt{N_s}$ so the total complexity is then equal to $\mathcal{O}(N_t N_s^{1.5} \log N_s)$.

3.5 Multilevel Algorithm

The multilevel algorithm is built upon the foundations of the two-level algorithm. In principle, small subscatterers are accumulated into larger entities before the translation, on multiple levels. The same three basic steps are discussed as with the two-level algorithm.

3.5.1 Partitioning

A ranked subdivision of the scatterer is created by recursively dividing a cubical box that encapsulates the scatterer into smaller boxes. Assume there is a box that encapsulates the scatterer. Divide this box into eight smaller boxes. A box that is divided into smaller boxes is called the parent, making each smaller box its child. This is done recursively. The smallest/finest box is designated at level 1; this goes all the way up to level N_l. For level $i = 1, \ldots, N_l$ the following are defined:

$N_g(i)$ the number of non-empty boxes
$M_s(i)$ the average number of sources in each group
$R_s(i)$ the radius of the sphere that encloses a level i box
$K(i)$ the number of spherical harmonics for the translation functions

As in the two-level algorithm, a set of near and a set of far field group pairs are constructed. Again, every source/observer combination belongs to only one group pair. However, not all far field pairs have to inhabit the same level. The greater the distance between source and observer, the higher the level they are in. To do so, define $R_{c,\min}(i) = \xi R_s(i)$ as the cutoff separation for each level. Here $R_s(i)$ is the radius of the sphere that encapsulates each box in level i. To define the near and far field pairs, one must start at the highest level. A group pair on level N_l is denoted as 'level-N_l far field' when the centres of their corresponding spheres are separated by more than $R_{c,\min}(N_l)$. Following that, level $N_l - 1$

group pairs are denoted as 'level $N_l - 1$ far field pairs' when their sphere centres are separated by more than $R_{c,\min}(N_l - 1)$, though they must not include interactions in any of the N_l-level far field pairs. This process is continued recursively. At level 1 the group centers that are separated by less than $R_{c,\min}(1)$ are labeled as near field pairs.

An example of the multilevel subdivision can be found in figure 3.7. The observer is again located in the box marked with an X. In level 1 with the smallest boxes, the signals are near field pairs. The somewhat larger boxes are then level 2, with the corresponding level 2 far field pairs. The largest boxes are level 3, in this case $N_l = 3$

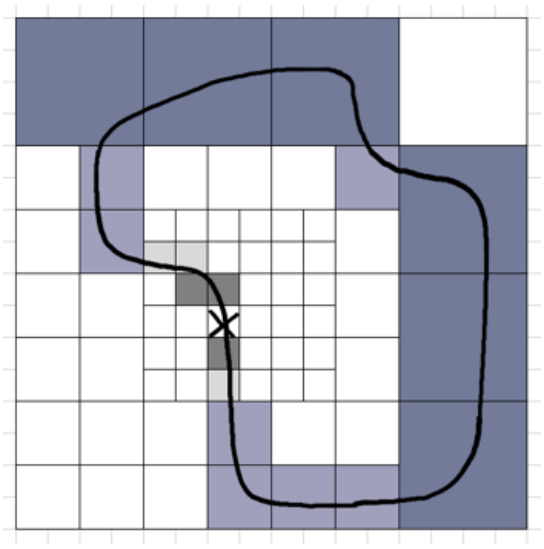


Figure 3.7: Example of multilevel subdivision

As before, the fundamental subsignal duration for level i can be set as

$$
\begin{aligned}
&T_s(i) = M_t(i)\Delta_t \\
&M_t(i) = \min_{\gamma,\gamma'} \{\lfloor (R_{c,\gamma\gamma'} - 2R_s(i))/(c\Delta_t) \rfloor\} \\
&T_{s,\gamma\gamma'}(i) = M_{t,\gamma\gamma'}\Delta_t \\
&M_{t,\gamma\gamma'}(i) = M_t(i)\lfloor (R_{c,\gamma\gamma'} - 2R_s(i))/(cT_s(i)) \rfloor
\end{aligned}
$$

3.5.2 Transforming

To make this algorithm more efficient, for a level i the information stored in level $i-1$ rays can be reused. This is done by implementing two operations, interpolation and splicing. Their two complementary operations at the observer side are resection and anterpolation.

Interpolation and anterpolation manipulate the outgoing respectively incoming rays. Interpolation rewrites the rays in terms of spherical harmonics, increases the sampling rate and zero-pads the outer spherical spectrum. Anterpolation truncates the spherical harmonics and lowers the sampling rate over the sphere.

An outgoing ray can be spliced into two rays and interpolated; resection is its complementary operation. Figures 18.10 and 18.11 of [3] illustrate these operations well. Figure

18.10 from [3] has been added in figure 3.8 to show how the process works visually, as this makes it easier to understand the process.

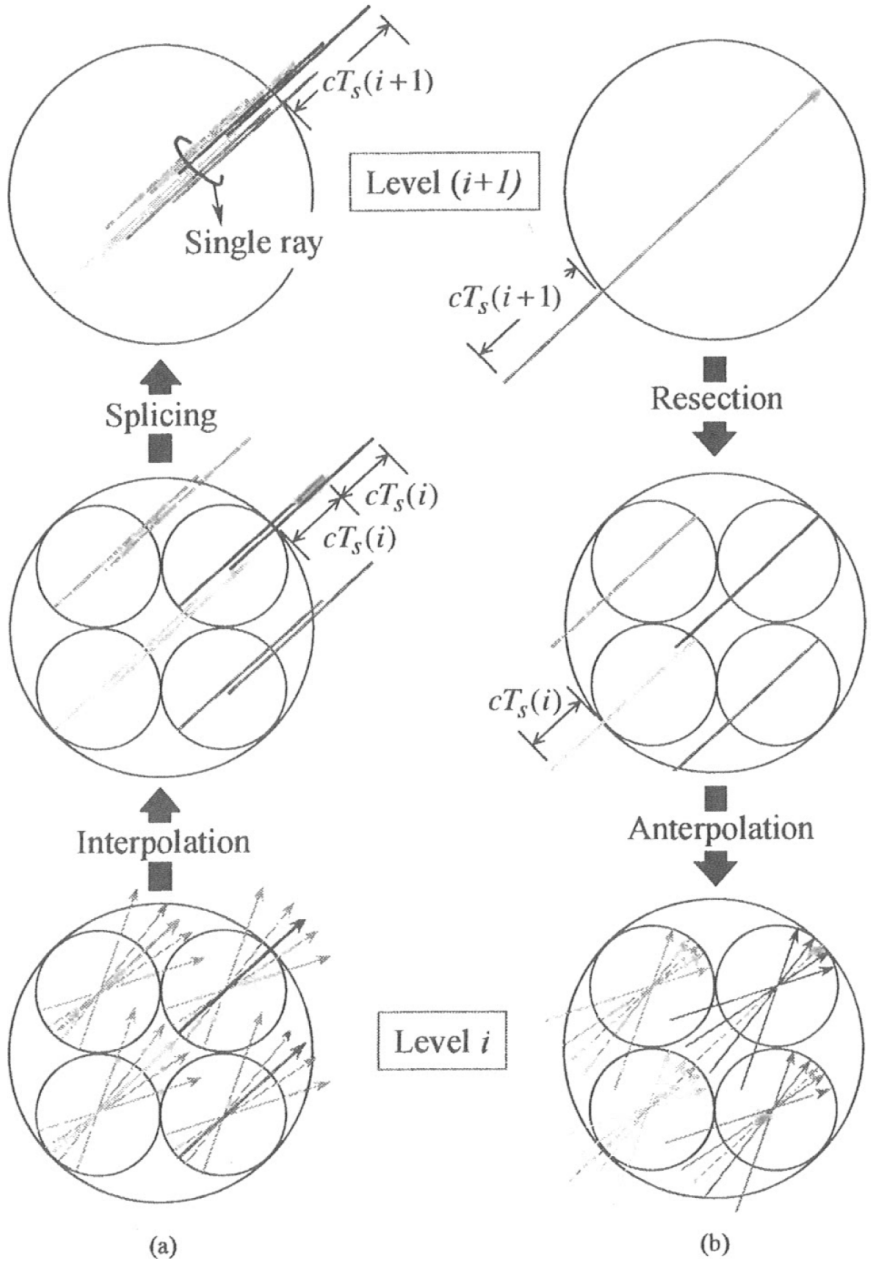


Figure 3.8: Figure 18.10 from [3] that visualises the operations mentioned in section 3.5.2.

3.5.3 The algorithm

Define the multilevel evaluation of 3.2 as follows:

1. *Evaluation of the near field contributions.*
   As in the two-level scheme, all the near-field interactions are dealt with via the MOT algorithm. They all reside on the first level.

2. *Evaluation of the far field contributions*

These contributions will be evaluated in a three-stage process, which is comparable to the two-level scheme. Since independently setting up all of the outgoing and incoming rays is too costly, apply interpolation, splicing, resecting and anterpolation for a more efficient construction.

(a) *Construction of outgoing rays*
At level 1, convolve the source signatures of $V_n^+(\hat{\mathbf{k}}_{pq}, t)$ from 3.43. Higher level rays are constructed from previous levels through interpolation and splicing. Levels must be crossed starting from level 1. So if the group pair resides in level i, the rays must cross levels $1, 2, \ldots, i-1, i$.

(b) *Translation*
As in the two-level scheme, the rays are translated between the far field pairs (γ, γ'). Since the number of harmonics $K(i)$ is level dependent, a new table for the translation function $\tilde{\mathcal{T}}(\theta', \Omega)$ has to be created for each level.

(c) *Projection of incoming rays onto observers.*
Starting at level $N_l - 1$, the incoming rays are resected and anterpolated at the higher level. By convolving incoming rays at level 1 via $\tilde{V}_n^-(\hat{\mathbf{k}}_{pq}, t)$ from 3.44, the fields at the observer sphere can be constructed.

3.5.4 Complexity

As for the two-level algorithm, the complexity of the multilevel algorithm is determined in [3]. Again minimizing it by setting $M_s \propto \sqrt{N_s}$, the complexity of step 1 is equal to $\mathcal{O}(N_t N_s)$ and the complexity of step 2 is equal to $\mathcal{O}(N_t N_s \log^2(N_s))$. Thus the total complexity is equal to $\mathcal{O}(N_t N_s \log^2(N_s))$.

3.6 Some notes

As mentioned before, the Fast Multipole Method and the Fast Fourier Transform are related per [8]. Both permit a sparse matrix for the matrix Z of 3.2. Since by 3.39 a Fourier transform is applied to find the translation function 3.41, the relation with FMM is understandable. This also peaks the interest in the Fourier transform and speeding it up: a large part of the algorithm is applying Fourier transforms to translation functions. Optimizing that part of the algorithm accelerates the algorithm itself considerably. The same also applies for the multilevel algorithm.

Chapter 4

Background Information

This chapter contains all the background information on Fourier transform and on programming on GPUs in, for instance, the programming language Julia. An important part of the PWTD algorithm is the use of Fourier transforms, which the entire first section is dedicated to. It discusses the discrete Fourier transform, various algorithms to determine them and a way how to parallelise Fourier transforms. The second section is devoted to the programming language Julia, while the third section is dedicated to GPU programming.

4.1 Fast Fourier Transform

As seen in the section on the Plane-Wave Time-Domain algorithm, the main part of equation 3.32 consists of multiple convolutions. Convolutions can be solved by using Fourier transforms, per the theory below. Furthermore, Fourier transforms are excellent candidates for parallel programming, therefore accelerating Fourier transforms by using a GPU can result in large performance gains.

4.1.1 Theory

The Fourier transform transforms a function of time into into a function of frequency, which is complex-valued; it lives in the frequency domain. The Fourier transform is represented as follows, for a function f in the time domain:

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(t) e^{-2\pi i t \xi} dt \tag{4.1}$$

$$f(t) = \int_{-\infty}^{\infty} \hat{f}(\xi) e^{2\pi i t \xi} d\xi \tag{4.2}$$

Discrete Fourier Transform

To evaluate the Fourier transform numerically, the discrete Fourier transform (DFT) is introduced. Assume there is a finite sequence of examples of a function. The DFT converts this sequence into a sequence of same length discrete-time Fourier transforms. If $\{\mathbf{x_n}\} := x_0, x_1, \ldots, x_{N-1}$ is the sequence to be converted and $\{\mathbf{X_k}\} := X_0, X_1, \ldots, X_{N-1}$ is the sequence the function is converted to, the DFT can be denoted as

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \tag{4.3}$$

$$= \sum_{n=0}^{N-1} x_n \left[\cos(2\pi kn/N) - i\sin(2\pi kn/N)\right] \tag{4.4}$$

The transformation is denoted by $\mathcal{F}$, such that $\mathbf{X} = \mathcal{F}(\mathbf{x})$. $w = e^{2\pi i/N}$ corresponds to the first complex N-th root of 1. Evaluating the DFT directly requires $\mathcal{O}(n^2)$ operations.

The DFT can be used to approximate the Fourier transform of functions on an interval. Assume the function $f(t)$ is to be evaluated on the interval $[t_0, t_1]$. The interval is split into N gridpoints, where the distance between two neighbouring gridpoints is denoted as Δt and the total distance of the interval is denoted as T. It follows then that $t_j = t_0 + j\Delta t$ such that $t_1 = t_0 + (N-1)\Delta t$. The interval can be approximated with a sum, which is denoted as:

$$\int_{t_0}^{t_1} f(t)e^{-it\omega}dt = \Delta t \sum_{j=0}^{N-1} f(t_j)e^{-it_j\omega}$$

$$= \Delta t\ e^{-it_0\omega} \sum_{j=0}^{N-1} f_j e^{-ij\Delta t\omega} = (*)$$

By setting $\omega_k = k\frac{2\pi}{N\Delta t} = k\frac{2\pi}{T}$, then

$$(*) = \Delta t\ e^{-it_0\omega_k} \sum_{j=0}^{N-1} f_j e^{-ijk\frac{2\pi}{N}}$$

It follows that

$$\sum_{j=0}^{N-1} f_j e^{-ijk\frac{2\pi}{N}} = \text{DFT}\left[f_j|_{i=0}^{N-1}\right]_k$$

Convolutions

As discussed at the beginning of the section, Fourier transforms can be used to determine the convolution of two functions. The Convolution Theorem is stated below.

Theorem 1 (Convolution Theorem)**.** *Let f and g be two functions with convolution $f * g$. Let $\mathcal{F}$ denote the Fourier transform, then $\mathcal{F}(f)$ and $\mathcal{F}(g)$ are the Fourier transforms of f respectively g. Denoting $\cdot$ as point-wise multiplication, we then have*

$$\mathcal{F}(f * g) = \mathcal{F}(f) \cdot \mathcal{F}(g) \tag{4.5}$$

$$\mathcal{F}(f \cdot g) = \mathcal{F}(f) * \mathcal{F}(g) \tag{4.6}$$

This also works for the Fourier inverse:

$$f * g = \mathcal{F}^{-1}\left(\mathcal{F}(f) \cdot \mathcal{F}(g)\right) \tag{4.7}$$

$$f \cdot g = \mathcal{F}^{-1}\left(\mathcal{F}(f) * \mathcal{F}(g)\right) \tag{4.8}$$

The proof can be found at [11]. A standard convolution algorithm has a complexity of $\mathcal{O}(n^2)$; the Fourier transform reduces this to $\mathcal{O}(n \log n)$. The Convolution Theorem is also applicable to the discrete Fourier transform.

4.1.2 Fast Fourier Transform

As discussed before, the computational cost of the DFT is quite high at $\mathcal{O}(n^2)$. To reduce this, various algorithms have been developed that lower the computational costs. These algorithms are called fast Fourier transforms and can rapidly compute DFTs by factorizing the DFT into a product of sparse matrices or factors. This can reduce the computational costs from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$. The most-used FFT algorithms are the Cooley-Tukey algorithm and Split-Radix algorithm.

Cooley-Tukey algorithm

The Cooley-Tukey algorithm [9] is one of the most well-known and widely-used FFT algorithms. It rewrites the DFT into terms of smaller DFTs recursively, and solves these. This reduces the computation time to $\mathcal{O}(N \log N)$. Furthermore, these smaller DFTs can be combined with other algorithms to solve them quickly.

In general, Cooley-Tukey algorithms follow the same re-expression of the DFT, which has a composite size $N = N_1 N_2$:

1. Perform N_1 DFTs of size N_2
2. Multiply by complex roots of unity
3. Perform N_2 DFTs of size N_1

In part 2, these complex roots are also called twiddle factors. These can be written as $\omega_N^n = \exp(-2\pi i k n/N)$. The twiddle factor is also known as the phase factor, and it has some symmetry and periodicity properties attributed to it:

$$\text{Symmetry property: } \omega_N^{k+\frac{N}{2}} = -\omega_N^k$$
$$\text{Periodicity property: } \omega_N^{k+N} = \omega_N^k$$

Normally, either N_1 or N_2 is a small factor called the radix, which is generally a prime number such as 2 or 3. If N_1 is the radix, the algorithm is called a 'decimation in time' (DIT) algorithm; if N_2 is the radix it is called a 'decimation in frequency' (DIF) algorithm. An example of a radix-2 DIT algorithm follows.

The DFT is defined as in 4.3. The radix-2 DIT first computes the even-indexed DFTs ($x_{2m} = x_0, x_2, \ldots, x_{N-2}$) and then the odd-indexed DFTs ($x_{2m+1} = x_1, x_3, \ldots, x_{N-1}$). These two results are combined to determine the DFT of the entire sequence. This can then be performed recursively. This is the same as writing

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N}(2m)k} + \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N}(2m+1)k} \tag{4.9}$$

The common multiplier $e^{-\frac{2\pi i}{N}k}$ is then factored out of the odd-sum. The two sums are the DFT of the even-indexed part x_{2m} (which is denoted as E_k) and the DFT of the odd-indexed part x_{2m+1} (which is denoted as O_k). It then follows that

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N/2}mk} + e^{-\frac{2\pi i}{N}k} \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N/2}mk} \tag{4.10}$$

$$= E_k + e^{-\frac{2\pi i}{N}k} O_k \tag{4.11}$$

Due to the periodicity of the complex exponential, $X_{k+\frac{N}{2}}$ is also obtained; its derivation can be found in [9]:

$$X_{k+\frac{N}{2}} = E_k - e^{-\frac{2\pi i}{N}k} O_k \tag{4.12}$$

The algorithm gains its speed by re-using the results of these intermediate computations.

Split-Radix Algorithm

The split-radix algorithm [12] is derived from the Cooley-Tukey algorithm that uses a combination of the radix-2 and radix-4 algorithms. It recursively re-expresses a DFT of length N into three DFTs, one of length $N/2$ and two of length $N/4$. The algorithm therefore only works if N is a multiple of 4. As it divides the DFT into smaller DFTs, it can be used in combination with other algorithms.

The DFT definition 4.3 is again assumed. The first part of the algorithm sums over the even indices x_{2n_2}, then it sums over the odd indices via two pieces: x_{4n_4+1} and x_{4n_4+3}, depending on the index being either 1 $\mod 4$ or 3 $\mod 4$; n_m denotes an index that runs over $0, \ldots, N/m - 1$. Using $\omega_N = \exp(-2\pi i/N)$, the summation then looks like

$$X_k = \sum_{n_2=0}^{N/2-1} x_{2n_2} \omega_{N/2}^{n_2 k} + \omega_N^k \sum_{n_4=0}^{N/4-1} x_{4n_4+1} \omega_{N/4}^{n_4 k} + \omega_N^{3k} \sum_{n_4=0}^{N/4-1} x_{4n_4+3} \omega_{N/4}^{n_4 k} \tag{4.13}$$

The smaller DFTs can then be performed recursively and combined. Using the twiddle factors and denoting U_k as the sum of size $N/2$ and Z_k and Z'_k as the first and second sums of size $N/4$, respectively, it can be written as

$$\begin{aligned}
X_k &= U_k + \left(\omega_N^k Z_k + \omega_N^{3k} Z'_k\right) \\
X_{k+N/2} &= U_k - \left(\omega_N^k Z_k + \omega_N^{3k} Z'_k\right) \\
X_{k+N/4} &= U_{k+N/4} - i\left(\omega_N^k Z_k - \omega_N^{3k} Z'_k\right) \\
X_{k+3N/4} &= U_{k+N/4} + i\left(\omega_N^k Z_k - \omega_N^{3k} Z'_k\right)
\end{aligned}$$

This gives all the outputs for X_k by letting k run over $0, \ldots, N/4 - 1$, thus requiring less computations.

4.1.3 Parallel FFT

From the algorithms described above, it becomes evident that a parallel implementation of FFTs is a valid extension due to the splitting of the DFT into smaller DFTs. There are two implementations to consider: the parallelisation of current algorithms and the development of new parallel algorithms.

The easiest way to use parallel programming for FFTs is to parallelise the currently used algorithms. A paper by Michael Balducci et al [4] describes parallel forms of the most popular FFT algorithms. As an illustration, the parallel form of the Radix-2 algorithm is discussed. It follows the development of this parallel form from an article [22] by Somasundaram Meiyappan.

The Decimation-In-Frequency form of the Radix-2 algorithm sets $N_2 = 2$, thus it performs $N/2$ DFTs of size 2. Figure 4.1, figure 3 from article [22], illustrates this implementation for a DFT of length 8.

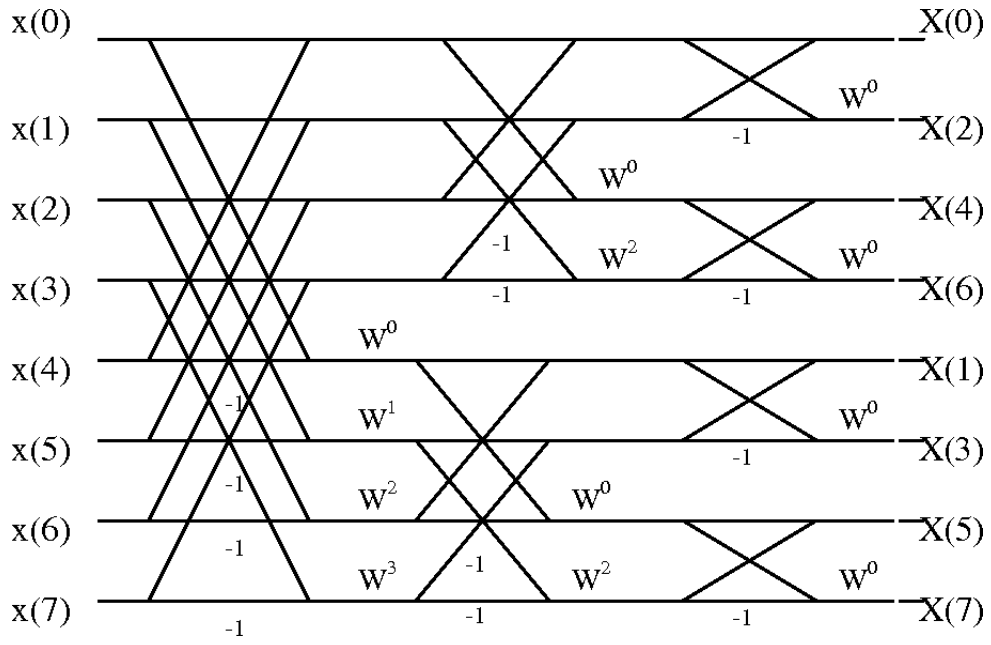


Figure 4.1: 8-point Radix-2 FFT DIF-implementation

The Radix-2 algorithm divides the computation of the DFT into $\log_2 N$ stages, in this case 3. In the first stage, a single 8-point DFT is performed, in the second stage two 4-point DFTs and in the third stage four 2-point DFTs. The two latter stages are well suited for parallel execution, as the two respectively four DFTs can be performed simultaneously.

The downside to using parallel programming here is the communication costs after each stage. The results from the performed DFTs have to be communicated to the other processors before the next stage can start. Figure 4.2 (figure 5 from [22]) shows how this works using four processors.

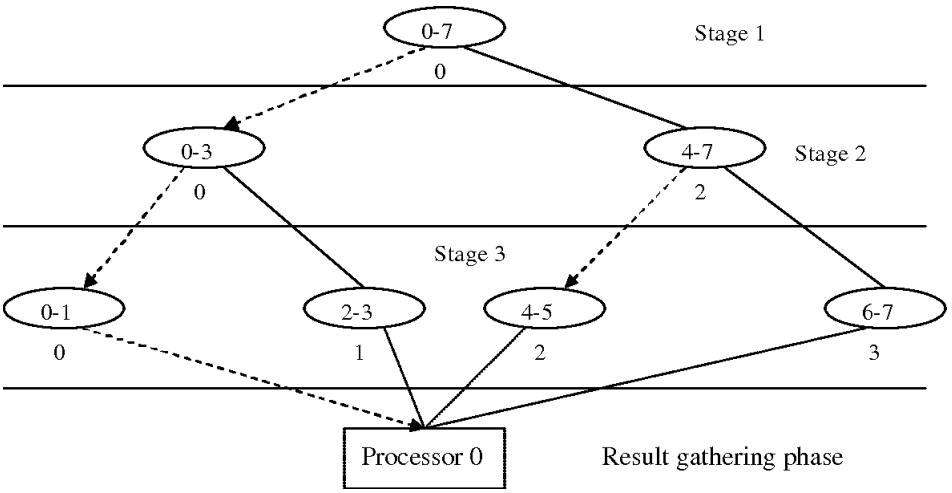


Figure 4.2: 8-point Radix-2 FFT on four cores

In the first stage again a single 8-point DFT is performed. The results are then communicated, where processor 2 is awaiting the results from processor 0. In stage 2 two 4-point DFTs are performed on processor cores 0 and 2, after which the results have to be communicated over all processors before stage 3 can be started. Depending on the device used, this constant communication can considerably slow down the parallel implementation.

Article [22] does offer a solution of sorts that can reduce the communication costs. Denoted as Communicate Twice, the same DFT is computed on multiple processors, after which the results can directly be used. Figure 4.3, which corresponds to figure 6 from [22], shows how this works.

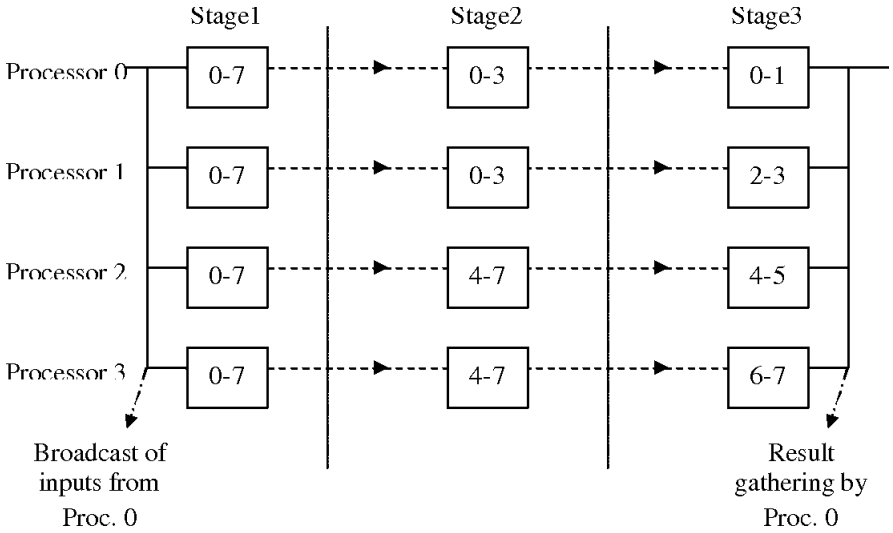


Figure 4.3: 8-point Radix-2 FFT using Communicate Twice-method

In stage 1, all the processors perform the same, 8-point DFT. It is assumed that all the results are approximately the same. Then processors 0 and 1 perform the first 4-point DFT whilst processors 2 and 3 perform the second 4-point DFT of stage 2. Finally, stage 3 is performed as usual, with all four processors computing a different 2-point DFT. At the end of stage 3, the results are added together.

The upside to this method is that it reduces communication time considerably. The

downside is that performing the same operation multiple times might give different results, which can distort the final value of the FFT.

Note also that the above situation is an ideal situation, where the right amount of processors are available so no DFTs have to be queued. Regular system are normally outfitted with four to eight processors (though the AMD Threadripper 3970X with 32 cores would surely come in handy) which means that larger DFTs will have to queue smaller DFTs on the processors, especially in the latter stages. This again brings more computation time with it. The usage of GPUs in this case brings a significant advantage: since they have hundreds of computing units available, larger DFTs can be computed efficiently in this way.

To counteract the scaling issue of this method, newer parallel FFT algorithms have been developed. An article [21] by Herbert Karner and Christoph Ueberhuber from the Technical University of Vienna discusses these newer algorithms.

4.1.4 Implementations

There are various implementations of the discrete Fourier transform. The most well-known are discussed below.

FFTW

FFTW, or the Fast Fourier Transform of the West, is a software library that can be used to compute DFTs efficiently. It was developed by Matteo Frigo and Steven G. Johnson at MIT [15]. It is a free library and is, in most instances, the fastest implementation of a fast Fourier transform.

FFTW's strength is that it accepts any input, independent of length, rank or it being real or imaginary. FFTW incorporates a planner that chooses the best algorithm for the input. The most-used algorithms are the Cooley-Tukey algorithm and the split-radix algorithm, as discussed before, as well as the prime-factor algorithm and Rader's algorithm. Besides these algorithms, the planner can also choose to use a combination of these algorithms or produce its own code for arbitrary array sizes, though the code generator can produce algorithms that the developers do not completely understand. More information on the planner can be found in the official FFTW documentation [16].

FFTW is free and reliably fast, which means that many programming languages have packages that use the library. For the programming language Julia, more information on the packages used for FFTW can be found at [29] and [35].

Intel MKL

Intel MKL, or Math Kernel Library, is a mathematics library that can be used by all devices that use Intel processors. Besides Fourier transforms, the library can accelerate various mathematical operations, including linear algebra, vector mathematics, statistics and partial differential equations. FFTs written for FFTW can easily be ported to Intel's library by using included interface wrappers.

Performance wise, Intel MKL and FFTW are more or less on a par, with each trying to trounce the other when it comes to speed-up whilst performing FFTs. Intel was at the top from 2011-2016, though FFTW has become the better alternative the last few

years. In his keynote during JuliaCon 2019 [40], professor Steven Johnson addressed the performance of FFTW, comparing it to various FFT implementations including Intel MLK. FFTW has the better performance, especially when the size is not a power of 2.

Nvidia cuFFT

Nvidia has developed its own FFT library that makes use of its CUDA platform for graphics cards and is called cuFFT [42]. It uses GPU-acceleration to achieve performance improvements up to ten times compared to CPU implementations. It also makes use of the Cooley-Tukey algorithm (as well as the Bluestein algorithm) and can work in conjunction with FFTW. There is also an OpenCL version called clFFT [31]. More information on the differences between CUDA and OpenCL can be found in chapter 5.

4.2 Julia

Julia is a relatively new programming language that was developed by Jeff Bezanson, Stefan Karpinski, Viral B. Shah and Alan Edelman [5]. The developers wanted to build a programming language that combined all the best parts of MatLab, Python, Ruby, Perl, R and C, whilst also being extremely fast. It was to be a programming language that was easy to use, but could still be extremely powerful when needed and could excel at everything that was thrown at it. The developers have more or less succeeded in this goal. Though still seen as up and coming, Julia is picking up more and more users, especially those performing numerical mathematical computations. It has currently been downloaded over 4 million times.

There are various reasons why Julia is so powerful, which will be discussed below in more detail. These include its unique syntax, its compiler, types, packages and ease of GPU usage.

4.2.1 History And Versions

Julia's first version was released on 14 February 2012, dubbed version 0.0. Exactly a year later version 0.1 was released to the general public. A new version was released every year, with 0.2 and 0.3 in 2014, 0.4 in 2015, 0.5 in 2016 and 0.6 in 2017. The first huge upgrade was released in August 2018 together with a version 0.7. This latter version was released to help test packages for the big 1.0 release and to guide users into any syntax and underlying code changes to the language.

Version 1.0 has been designated a long-term release, with support for at least a year. New .1 releases are released every few months with new features, with version 1.1 released in January 2019, 1.2 in August 2019, 1.3 in November 2019, 1.4 in April 2020 and 1.5 in August 2020. New updates for 1.0 are released on a regular basis to iron out any bugs.

4.2.2 Syntax

Julia has its own unique syntax, which has elements of MatLab and Python in it but also its own unique approach to mathematical programming. It follows Python in that it uses tabbed-indents for for-loops and such instead of curly brackets that open and close an operation. On the other hand, when using an index it will start at 1 like MatLab, instead

of 0 such as Python does. Many functions like `zeroes` that create an array of matrix filled with zeroes are equivalent to those found in MatLab.

It does add its own unique twist by adding options to include symbols such as $\in, \subset$ and Greek letters; these are defined in the same way as in LaTeX. These work in various ways:

- As Booleans; for instance, running `A = [1,2,3,4,5], 2∈A` will return `True`.
- As an index; one can run through a for-loop by using the set A above and setting `for i∈A, println(i), end` which will return 1, 2, 3, 4, 5.
- Setoperations; setting `A=Set([1,2,3])` and `B=Set([2,4,6])`, the operation `intersect(A,B)` returns `Set([2])`.

4.2.3 Types

One of the hallmarks of Julia is its approach to types. Their main attribute is that they are highly dynamic. This ranges from values being of any type when typesetting is omitted, to setting detailed types and type unions to values. Types can also be parametrised by other types. An extensive list of all the type-operations available can be found in the corresponding Julia documentation [38].

4.2.4 Packages

Julia has a built-in package manager that manages installed packages in its own unique way. It is designed around environments, which are sets of packages that can be used by an individual project or can be used globally on a system. Environments can be updated independently from each other, which can prevent updates from breaking other packages.

Denoted by `Pkg` and called by pressing the]-key in the Julia REPL (which stands for Read-Eval-Print Loop), it can install packages from the general Julia repository or from specific Github repositories by using the `add 'Package'`-command. It can also handle updates of packages for the user; the command `status` returns a list of all installed packages and their versions, the command `update` updates the packages.

There are a few ways to create a package.

- Create a GitHub (or GitLab) repository and install it as a package. Adding files can then be done via Git.
- Use the package PkgTemplates [39] to create a package.
- Use the command `pkg> generate` to create a new package.

For more information on packages, see the Julia documentation [37].

Benchmarking is also done by using different packages. As benchmarking will encompass a large part of the thesis, this will be discussed in more depth in the following chapter on Methodology.

4.2.5 Compiler

Julia makes use of a so-called just-in-time, or JIT, compiler to run code. What this means is that the code is compiled whilst a program is executed rather than it being compiled before execution. The executing code is continuously analyzed to optimise speed-up. This can increase performance and speed up the code. However, it has its downsides as well. Initial execution can have somewhat of a delay due to the original code being compiled into bytecode, before it is dynamically compiled into machine code. This raises some problems with running benchmarks, as the source code has to be compiled into bytecode before it is compiled as machine code; this adds time to the overall execution time of the application. Running the application again on the pre-compiled bytecode improves performance and gains the desired speed-up of using a JIT-compiler [1].

4.3 GPU Programming

4.3.1 Introduction

The GPU, or Graphics Processing Unit, is an integral part of any personal computer, be it a laptop, phone or gameconsole. At its core it is a specialised unit for creating and storing images and outputting them to a display. The GPU relies on parallel programming and processing, which makes GPUs excellent candidates for algorithms that require parallel processing of large amounts of data.

Though traditionally used for video games, in recent years GPUs have been used for various other processes. These include video decoding and deep learning, where especially the latter has gained popularity. AI and machine learning development have thrived due to the general availability and affordability of powerful graphics cards.

Scientific Computing has also gained much from the implementation of graphics cards. The power of GPUs alone can speed up large-scale problems; parallel programming makes this even more powerful. This section takes a look at why GPUs are more powerful for certain tasks, and takes a look at the architecture of GPUs versus CPUs. A quick look at the programming languages for GPUs is also taken; the more popular one, CUDA, will be discussed in more depth in the Methodology chapter, as it is used for this thesis.

4.3.2 The Workings Of A GPU

A GPU differs from a CPU in various different ways. The most obvious difference is in the amount of cores each processing unit has. A CPU mostly has four to eight, even up to 24 cores. A GPU, on the other hand, can have thousands. The downside of the GPU is that, when compared to a CPU, it can only perform a fraction of the operations a CPU can perform. However, it can do them very, very fast by utilizing all those cores in parallel. Also of some importance is that CPUs run at higher clock speeds (the number of clock cycles per second) and are able to manage input and output of the system, something the GPU cannot do.

Most operating systems are also dependent on CPUs to run and GPU usage is only effective for larger-scale problems. Therefore most programs and algorithms start with CPU operations; parts of the code are then programmed directly onto the GPU before the results are sent back to the program. A systematic approach can be found in figure 4.4.

The image shows that via the IO (Input/Output) data is sent to the CPU, from where data is sent to the GPU, where parallel programming is then applied. The data is then sent back to the central computing system, which can in turn also send data between the two computing units. Finally, data is sent back to the IO before the program is stopped.

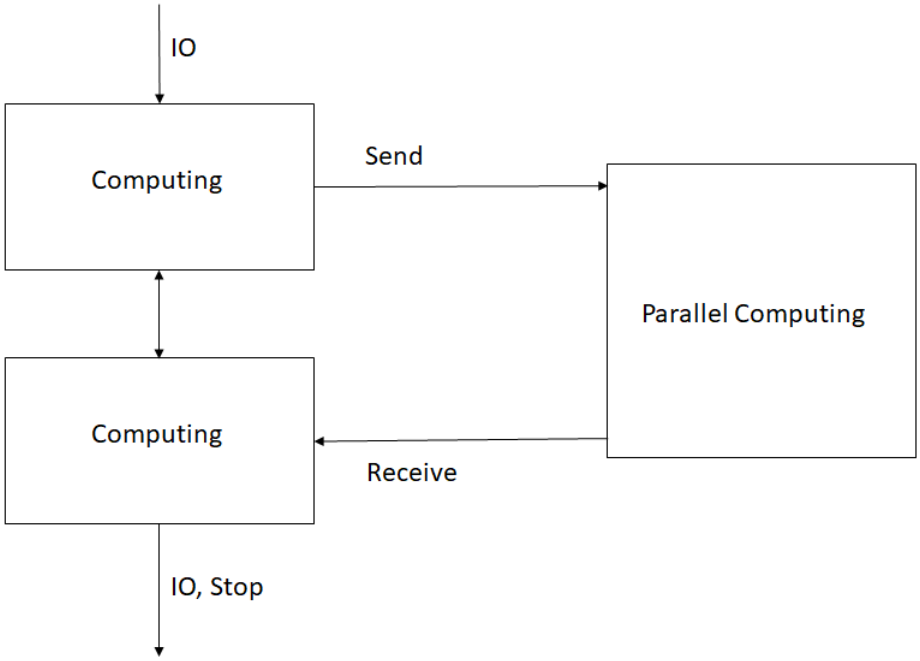


Figure 4.4: Operations in a regular CPU-GPU system

This data transfer between the CPU and GPU is the entire crux of GPU programming: though programming simple operations in batches on a GPU might reduce computational costs significantly, data transfer from the CPU to GPU and back is extremely slow. All the data has to be copied from the CPU RAM to the GPU VRAM before the operations can start, then afterwards the new data has to be copied back before it can be used by the CPU.

4.3.3 GPU Architecture

As discussed before, a GPU has a multitude of processors that all execute the same set of instructions in parallel, without any dependence between the cores. The GPU has a fixed number of multiprocessors, each of which contains a further eight scalar processors. The following image from [26] shows how this works in practice.

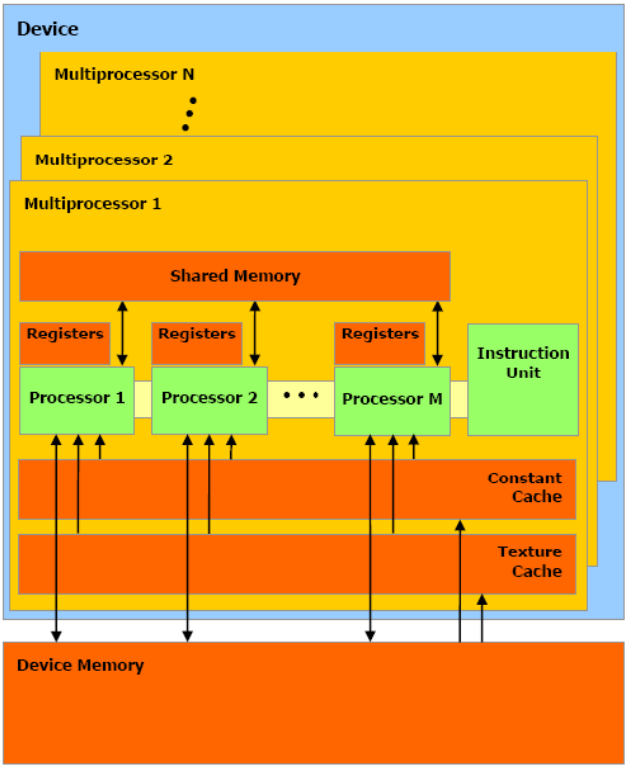


Figure 4.5: Architecture of a GPU, per [26]

As a side-note, the different types of memory in 4.5 are discussed below.

- **Global Memory** Memory of the device itself. Largest in size of all four memories, though also has the largest access time, about 200 times larger.
- **Register Memory** Each processing unit has its own register memory. Each thread that is run can only use one register.
- **Shared Memory** Memory that can be accessed by all threads in a multiprocessor block.
- **Texture Memory** Read-only memory on a multiprocessor.

Efficient utilization of the register- and shared memory should also cut down on computational and latency costs.

[26] chapter 3 goes into greater detail in how to use the architecture of the processors and memory effectively. It includes various methods and techniques that have been optimised for GPU computations. The rest of the reader is also insightful. For instance, chapter 2 describes various ways to use parallel programming with certain methods such as matrix-vector products and LU decomposition.

4.3.4 Single vs Double Precision

Another attribute to consider when using GPUs for computing is the precision of the values used. Most GPUs only support single precision operations, whilst CPUs support

double precision operations. To understand why, one must first look at the definitions of single and double precision.

Per the IEEE standard for Floating-Point Arithmetic [18], single and double precision are defined as follows:
A single precision floating point uses 32 bits to convey a value. It uses 1 bit to determine if the value is positive or negative, uses 8 bits to describe the exponent and 23 bits to represent the digits of the number.
A double precision floating point uses 64 bits to describe a value. As in single precision, it uses 1 bit to determine if the value is positive or negative. Furthermore, it uses 11 bits to describe the exponent and the remaining 52 bits to represent the digits of the number.

What this means is that double precision floating points are more accurate than single precision floating points. However, there is a trade-off to using them, as double precision values take up more memory and they need more computational power when in use. Because of this, GPUs mostly have more cores that can compute single precision floating points than cores that compute double precision floating points. Chapter 3 of article [14] discusses the maximum theoretical performance of a few CPUs and GPUs. As an example, an Nvidia GTX 1080ti has a maximum theoretical computing performance of 354 GFLOPS for double precision floating points, and 11340 GFLOPS for single precision floating points, due to the many more single precision cores available. GFLOPS stands for Giga FLOPS, or Floating Point Operations. There are videocards available (such as the Tesla GPUs used in the server) that have more double precision cores available. However, these cards are significantly more expensive than regular consumer cards.

4.3.5 GPU Programming Languages

There are two main competitors for GPU programming, which are CUDA and OpenCL. CUDA is developed by Nvidia and only works with its own cards; OpenCL is developed and maintained by the non-profit Khronos Group. Both are discussed shortly below; as mentioned before, a more in-depth look will be taken into CUDA in the chapter on Methodology.

CUDA

CUDA stands for Compute Unified Device Architecture and is a parallel computing platform that is developed by Nvidia for computing on GPUs. If developers have a GPU with CUDA cores, the CUDA platform acts as a software layer that gives direct access to the GPU's instruction set and parallel processing cores. It is designed to work with C, C++ and Fortran, eliminating the need for extensive GPU programming skills.

CUDA has a few advantages over regular GPU programming:

- Unified and shared memory; this encapsulates a fast shared memory region that can be shared among threads.
- Faster downloads and readings to and from a GPU
- Robust documentation
- Regular updates

The last two are especially important. Nvidia regularly updates CUDA with new features, the current version 11.0.221 is from August 2020. Because of these regular updates and support from Nvidia, developers are more likely to support and use CUDA. The platform has become synonymous with development in machine learning and AI development because of this. The fact that even Nvidia's mainstream graphics cards [33] support and use CUDA cores means that the bar is low for developers to start using CUDA in their programs.

OpenCL

OpenCL stands for Open Computing Language, and it is a framework for writing programs that can run on any platform consisting of CPUs, GPUs and other types of processors and hardware accelerators, as OpenCL views the system as a set of computing devices. It is an open standard that is maintained by the non-profit Khronos Group. [43]

OpenCL has its own C-like programming language called OpenCL C, which can make it cumbersome to program for. Third-party APIs do exist for most programming languages. OpenCL C includes ways to implement parallelism with vectors, operations and synchronization.

One of the advantages of OpenCL is that it is vendor-independent, meaning that it runs on almost every GPU, including for instance built-in Intel GPUs. This counteracts vendor lock-in. Furthermore, since it is maintained by a non-profit organization, the premise of the platform is to empower developers [41] instead of selling as much hardware as possible. The software is royalty-free and an open standard, meaning that anyone can use the platform for free. This does mean that there is less documentation available in comparison to CUDA, and due to the difficulty of implementing OpenCL as a result of its proprietary programming language, it is a lot less popular than CUDA.

Chapter 5

Methodology

Chapter 5 will focus on the methodology that was used for this thesis report. The first section focuses on the programming on GPUs, which is done via CUDA. Some examples illustrate how Julia makes it relatively easy to program on GPUs. The second section will focus on benchmarking, which is essential for determining when a certain process is faster on either the CPU or GPU. The final section focuses on the Plane-Wave Time-Domain algorithm and the package that thesis advisor Dr. Kristof Cools wrote for it.

5.1 CUDA Programming

The platform CUDA is essential for GPU programming for Nvidia graphics cards, as discussed in the previous Background chapter. Using CUDA in Julia requires the usage of various packages and the writing of so-called kernels. These are discussed below. Operations are also discussed; though they essentially work the same as operations for CPUs, some context is necessary for using operations on GPU arrays.

5.1.1 Kernels

Running functions on the GPU works via kernels. A function or operation is written, after which arrays are defined on the GPU and the operation or function is run on these arrays. The CUDA kernel is started by using the @cuda macro. An example below illustrates an easy kernel for adding two arrays together. N is taken as a power of 2. Note that these kernels are written in Julia and differ from regular CUDA kernels. Julia translates the kernel into a CUDA kernel with the proper configuration.

```
using CUDAnative, CuArrays

function gpu_add1!(y, x)
    for i in 1:length(y)
        @inbounds y[i] += x[i]
    end
    return nothing
end

x_d = fill(CuArray, 1.0f0, (N,))
y_d = fill(CuArray, 2.0f0, (N,))
```

```
@cuda gpu_add1(y_d, x_d)
```

These kernels allow code to speak directly to the GPU, and further variables can be specified. Note also that arrays defined on the GPU are denoted with an underscore `d`, where the d stands for device. This is to distinguish them from regular CPU arrays.

In the example below, the first example has been expanded upon to include the amount of threads of the GPU that should be used, and it uses the thread index number and dimension of each block to perform parts of the summation in parallel. Note that both examples are from the CuArrays documentation [32].

```
using CUDAnative, CuArrays

function gpu_add2!(y, x)
    index = threadIdx().x
    stride = blockDim().x
    for i = index:stride:length(y)
        @inbounds y[i] += x[i]
    end
    return nothing
end

x_d = fill(CuArray, 1.0f0, (N,))
y_d = fill(CuArray, 2.0f0, (N,))

@cuda threads=256 gpu_add2!(y_d, x_d)
```

Most standard operations on GPU arrays will try to optimise the usage of threads and blocks for the user, though the specification in kernels can be useful for certain applications.

5.1.2 CUDA in Julia

Julia is a high-level scripting language, which brings many advantages with it. This includes the ability to write kernels for the GPU natively for a program as well as all the code surrounding it. CUDA programming is done via various packages, of which the three most important ones are discussed below.

- **CUDAnative.jl** The main package used for running and compiling Julia kernels on CUDA hardware. This package makes sure the Julia GPU kernel is translated correctly to a CUDA kernel in C.
- **CuArrays.jl** A package that has various operations to create arrays on the GPU. This includes copying existing arrays and creating entirely new arrays on the GPU with for example random numbers or zeros.
- **GPUArrays.jl** A package that can correctly implement operations on CuArrays (or CLArrays) and always works in conjunction with one of the two packages.

5.1.3 Operations

Because of these packages, it is possible to perform basic operations on GPU arrays. As an example, one can create a CUDA array on the GPU and then perform an FFT on it, without requiring extra packages or difficult kernels. An example of this can be found below. A randomised matrix of size 512×512 is created as well as a same-sized matrix with zeros, where the FFT is written to. The GPU operation in this case is started by the `@sync` macro. The `@sync` macro waits for all prior operations to complete before it starts the new operation, to make sure that the GPU operation starts correctly. Starting a GPU kernel requires the `@cuda` macro as seen in the examples of the previous section.

```
x_d = CuArrays.randn(512,512)
y_d = CuArrays.fill(0.0f0, (512,512))
y_d = CuArrays.@sync(fft(x_d, 1))
```

Note that operations that incorporate a GPU array and a CPU array will not compute; furthermore, functions that are written for CPU variables and arrays will run slowly on GPU arrays (or not at all) as the variables have to be written back to the CPU before the function can commence.

Also of note is the usage of operations in kernels themselves. Most basic operations, such as addition and multiplication that are defined in the Base code of Julia, will work in a kernel. Regular functions such as the Fourier transform using the function `fft()` will not work in a kernel. GPU functions outside of basic operations have to be defined in the original package.

5.1.4 Fourier Transforms

The implementation of Fourier transforms in the FFTW package are a great example of defining a function for usage with GPU arrays. Due to Julia's typing implementation, functions can be defined for different types of input, in this case for both a GPU and CPU array input. This works as follows:

```
julia> using CUDAnative, CuArrays, FFTW, AbstractFFTs

julia> x = randn(ComplexF32, (512,)); y = similar(x);

julia> x_d = CuArray(x); y_d = CuArray(y);

julia> y = fft(x);

julia> y_d = fft(x_d);

julia> typeof(y)
    Array{Complex{Float32},1}

julia> typeof(y_d)
    CuArray{Complex{Float32},1}

julia> y = fft(x_d)
    ERROR: CUFFTError
```

Note that the added `julia>` implies that the code has been run in the Julia REPL, with the corresponding results printed underneath each line. This particular example also shows how a CPU array can be copied to the GPU by using the `CuArray()`-function. By using the `typeof()` command the typing of a variable or array is returned. It shows that the Fourier transform works irregardless of typing. The last command shows that a CPU array cannot receive the contents of a function run on a GPU array; the system throws an error.

5.2 Benchmarking

To determine if using GPUs for certain operations is faster than using CPUs, these operations need to be timed, or benchmarked, to determine which method is faster. Benchmarking in Julia can be done by calling packages into the code, and specifying which part of the code has to be benchmarked. Julia also provides built-in tools for benchmarking. Both options are used in the experiments as described in the following chapter, and are discussed in more detail below.

5.2.1 Built-in Tools

Julia provides various macros that can be used for benchmarking purposes. The most-used macros are as follows:

The `@time`-macro is one of the most widely-used benchmarking tools Julia provides. It is used together with the expression that the user wishes to benchmark. It prints the time it takes to execute the expression, the number of allocations to the memory for the expression to execute as well as the amount of memory used, and the total number of bytes allocated for the execution. After that the value of the expression is returned.

An example:

```
julia> @time exp(265)
    0.043850 seconds (85,71 k allocations: 4.788 MiB)
1.2247225219887542e115
```

The `@timed`-macro expands upon the `@time`-macro by providing the same results as well as the garbage collection time and various memory allocation counters. In this case, these values are returned as an array:

```
julia> @timed exp(265)
(1.2247225219987542e115, 2.001e-6, 176, 0.0,
    Base.GC_Diff(176, 0, 0, 5, 0, 0, 0, 0, 0))
```

Returning these values as an array has its benefits: values can easily be assigned to variables and can be reused in other arrays to, for instance, portray the increase in computation time over the increase of size for the original problem.

The `@elapsed`-macro is the simplest of the three. It only returns the computation time of the expression, without returning the result or any other values.

```
julia> @elapsed exp(265)
3.3999e-5
```

5.2.2 BenchmarkTools

BenchmarkTools.jl [24] is a package with the sole purpose of benchmarking code in Julia, and was primarily developed by Jarrett Revels. Where the built-in tools provide rudimentary benchmarking options, the BenchmarkTools.jl package provides a diverse benchmarking toolset that covers everything from basic benchmarking to running collections of benchmarks, saving and reloading benchmarking parameters and the easy comparison of benchmarks. Most of this is covered in the BenchmarkTools.jl-documentation; below some of the highlights and tools that are used for this report are discussed. As is the case with the built-in Julia tools, most of these tools are used via macros.

The standard benchmarking macro from the BenchmarkTools.jl package is the `@benchmark`-macro. As before, using it is pretty straightforward:

```
julia> @benchmark exp(265)
BenchmarkTools.Trial:
    memory estimate:  0 bytes
    allocs estimate:  0
    --------------
    minimum time:     0.001 ns (0.00% GC)
    median time:      0.001 ns (0.00% GC)
    mean time:        0.023 ns (0.00% GC)
    maximum time:     0.101 ns (0.00% GC)
    --------------
    samples:          10000
    evals/sample:     1000
```

The `@benchmark`-macro provides a lot more data that the built-in tools. It also benchmarks differently. It performs the benchmark multiple times (in this case 10000 times) and determines the minimum, maximum, median and mean time for the expression. Furthermore, these values can be assigned to variables:

```
julia> @BenchmarkTools.mean(@benchmark exp(265))
BenchmarkTools.TrialEstimate:
    time:             0.026 ns
    gctime:           0.000 ns (0.00%)
    memory:           0 bytes
    allocs:           0

julia> x = @BenchmarkTools.mean(@benchmark exp(265)).time

julia> x
0.022997000000000014
```

Another way that BenchmarkTools.jl differs from the Julia tools is in the use of explicitly interpolating external variables of the expression. The reason why this is useful is that the evaluation of a variable also takes time. Explicitly interpolating it leaves out the computation time of the variable, and only focuses on the benchmarking of the expression itself. Below an example from the first experiment of this report that showcases this:

```
x_d = curand(N,)
y_d = curand(N,)
z_d = cufill(0.0f0, (N,))

times_gpu_vec[i,2] = BenchmarkTools.mean(@benchmark
    gpu_multi_vecvec!($x_d, $y_d, $z_d)).time
```

In this example, the function `gpu_multi_vecvec!()` is benchmarked for various GPU arrays of size 2^i. This function multiplies two vectors and stores the result in the third vector. The GPU arrays x_d, y_d and z_d are determined beforehand in the code; during the benchmark they are explicitly interpolated. In this way, the computation time of the GPU function can be compared to the corresponding CPU function without including the computation time of the necessary vectors. This example also shows the saving of the mean time as a variable in an array.

Besides the `@benchmark`-macro, the BenchmarkTools.jl package also provides comparable macros to the built-in Julia tools. These are `@btime`, `@btimed` and `@belapsed`. The results returned are in the same form as before, the main difference is that the benchmarking is done based on the BenchmarkTools.jl package instead of the built-in tools.

BenchmarkTools.jl includes many more tools for benchmarking, though they are outside the scope of this report.

5.3 PWTD Package

Thesis advisor Dr. Kristof Cools has, over the last few years, worked on a package [10] in Julia that can perform the Plane-Wave Time-Domain algorithm. Instead of writing an entire package for the GPU, it was decided that this package would be expanded with GPU enhancements for parts of the code. Since the PWTD.jl package makes use of the SampleArrays.jl package, also by Dr. Cools, that package was also reworked.

To rework the packages, they were forked in GitHub so the changes would not interfere with the original code. This was done in the IDE (Integrated Development Environment) Atom, which includes this functionality as well as other functionality of the Git language. Furthermore, Julia provides tools for developing and testing packages. By using the command `pkg> dev #package` in the package manager, a special development version of the package is created that can be changed without repercussions for the original package. This is done for both the PWTD.jl and SampleArrays.jl packages, and Julia is smart enough to use the dev-version of SampleArrays.jl in the dev-version of PWTD.jl.

Chapter 6

Numerical Experiments Setup

This chapter discusses the various experiments, or simulations, that were developed to determine the computational gains that GPU acceleration might bring to mathematical operations in general as well as the Plane-Wave Time-Domain algorithm. The chapter starts with a short introduction to discuss the machines that were used for the experiments. The first experiment discussed concerns itself with the testing of various basic operations on both the CPU and the GPU, to determine when and if it better to use a GPU for these operations. The second experiment expands upon this by determining the same answers, but then for Fourier transforms, with a smaller experiment that determines if using the planner functionality of the FFTW package is a reasonable alternative to the regular Fourier transform function. The third and final experiment concerns itself with the implementation of GPU-accelerated FFTs in the Plane-Wave Time-Domain algorithm.

6.1 Initial Setup

All experiments for this thesis were done in the programming language Julia on three different systems. Though the initial idea was to only use one system for testing (a server at Capgemini), due to scheduling issues with the server the experiments were also run on a laptop borrowed from Delft University of Technology as well as on a desktop built by the author of this report. The hardware and software specifications for each device are discussed below, as well as their general setup.

6.1.1 Capgemini Server

The server provided by Capgemini is made by HP and has top of the line specifications. These are:

- Two Intel Xeon Gold 12-core processors
- 192 GB RAM
- 4TB SSD
- Two Nvidia Tesla V100 GPUs with 16 GB HBM2 Memory each, and 5120 cores each

The two Intel processors also include hyperthreading. Hyperthreading is a system where each core of the CPU can manage two threads simultaneously; this results in a total of 48 threads available for the system. Furthermore, the graphics cards have 16 GB HBM2 memory available each and are connected via an NVLink bridge.

The system is running Ubuntu 16.04, CUDA 9.0 and Julia 1.0. However, due to certain circumstances, the software cannot be updated correctly or at all. This is because the server is also used for other applications, including an AI demo that requires the server to be connected to a router, which is then again connected to a different router, which is then connected to the internet. Somewhere along the line the connection to the internet is lost, hence the installation of updates and new packages is currently not possible. Furthermore, the demo requires certain software to be kept as-is, so for instance CUDA cannot be updated to version 11. (Running one of the experiments on the same laptop with both CUDA 9 and 10.1 results in the same output. However, support for the Tesla GPUs was added in CUDA 9, so it is feasible that improvements have been made in CUDA versions 10 and 11. Therefore it is difficult to say what kind of an impact this could have had on performance.) This does mean that drivers and packages haven't been updated in quite a while, which can also hamper performance.

Because the server was also used for other research and demonstrations, access to it was sometimes limited. As this led to some delays in testing, a laptop from the University was borrowed so work could be done when the server was in use.

6.1.2 University Laptop

The laptop that was used is the high-end model of the Delft University of Technology Laptopproject 2017-2018, which is an HP Zbook Studio G4. This project aims to provide powerful laptops at an affordable price to students for their studies. The laptop has the following hardware specifications:

- Intel Core i7-7700HQ
- 8 GB DDR4 RAM
- Nvidia Quadro M1200 with 1 GB VRAM and 640 CUDA cores

Software wise, it is running Windows 10 with the 1909 update, CUDA 11 and Julia 1.3. Drivers and packages are all up to date. The reason an older version of Julia was used is because the update to a newer version failed during installation.

6.1.3 Personal Desktop

The desktop was built by the author and has the following hardware specifications:

- AMD Ryzen 5 3600X with six cores
- 32 GB DDR4 RAM
- Nvidia RTX 2080 Super with 8 GB VRAM and 3072 CUDA cores

The AMD processor also has hyperthreading, hence it can be seen as a 12-core processor. Software wise, it is running Windows 10 with the 2004 update, CUDA 11 and Julia 1.4. Drivers and packages are all up to date.

6.1.4 Theoretical Computing Performance

The theoretical maximum performance of the three GPUs [44], [45], [46] for both single and double precision can be found in table 6.1.

GPU Name	Single Precision	Double Precision
Nvidia Quadro M1200	1399 GFLOPS	48 GFLOPS
Nvidia Geforce RTX 2080 Super	11150 GFLOPS	349 GFLOPS
Nvidia Tesla V100	141320 GFLOPS	70660 GFLOPS

Table 6.1: Theoretical performance of the used GPUs

The Quadro and Geforce card both show a massive performance difference between single and double precision, by a factor of 32. For the Tesla, this factor is only 2 as Nvidia has added more double precision cores to the card. This does mean that, for all three GPUs, it is better to use single precision numbers, as the performance is then greater. This does impact the accuracy of the experiments. Jowever, the main research question deals with optimizing performance, and since Δt is chosen small enough, using single precision will not hamper the outcome of the PWTD algorithm.

6.1.5 Development

The development of the experiments was mainly done on the University laptop, after which the code was ported to the server to see if results were the same or improved, due to the vastly more powerful hardware. Testing on the desktop was done at a later stage, as it was built after the experiments had originally been set up.

6.2 Experiment 1: General GPU Acceleration

This first experiment was developed to determine the performance increase a GPU might provide for various operations, and to see for which problem-sizes it is more advantageous to use the GPU. For each operation, two or three random vectors or matrices are created of size 2^N, with N varying from 1 to a predetermined maximum value dependent on the device; as the server has more memory available, it can handle larger sizes.
The tested operations are as follows:

- Vector-vector addition
- Vector-vector multiplication
- Vector-matrix pointwise multiplication
- Vector-matrix regular multiplication
- Matrix-matrix addition
- Matrix-matrix pointwise multiplication
- Matrix-matrix regular multiplication

The addition operation speaks for itself. To illustrate the regular and pointwise multiplication operations consider

$$A = \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix}; \qquad B = \begin{bmatrix} b_1 & b_2 \\ b_3 & b_4 \end{bmatrix}$$

Pointwise multiplication $A \cdot B$ results in

$$\begin{bmatrix} a_1 b_1 & a_2 b_2 \\ a_3 b_3 & a_4 b_4 \end{bmatrix}$$

Regular multiplication $A \times B$ results in

$$\begin{bmatrix} a_1 b_1 + a_2 b_3 & a_1 b_2 + a_3 b_4 \\ a_3 b_1 + a_4 b_3 & a_3 b_2 + a_4 b_4 \end{bmatrix}$$

The reason these operations were chosen is because they form the basis for most other operations and functions.

The time it takes for the CPU and GPU to perform each operation is benchmarked as discussed in chapter 4 using `BenchmarkTools.jl`. For each operation, the N for which the GPU outperforms the CPU is then determined.

The programming language used for this experiment is Julia. To set the experiment up, certain packages have to be added so the GPU can be called and used for operations. The most important ones for this experiment are the packages mentioned in Chapter 4, as well as the package `CuArrays.CURAND`. This package can create arrays with random numbers.

For each operation and size N, three randomised vectors or matrices are created for both the CPU and the GPU. That means that each matrix and vector consists of values between 0 and 1, randomly chosen; hence they are not sparse. This is valid for all matrices and vectors in experiments 1 and 2. Next, the operation is performed on the corresponding vectors or matrices. Using BenchmarkTools, the operation is performed 1000 times and the average time is determined for the operation to be performed. The times are then plotted against the size of the vector or matrix for both the CPU and the GPU.

6.3 Experiment: Planner Test

Experiment 2 focuses on the acceleration of the fast Fourier transform. As described in chapter 3.3 on the fast Fourier transform, the FFTW package includes a planner functionality that can determine the optimal algorithm for a certain size and type of array, which can then be reused for the same type of array, thus theoretically lowering the computation time for multiple FFTs of the same size.

To show that using the planner functionality does actually lower the computation time, an extra side-experiment was set up that benchmarks the computation times for a certain number of Fourier transforms for both the planner and for the regular `fft()`-function.

This simulation is done for differing sizes of the matrix that is to be transformed. Three different types of matrices are used, since powers of 2 and square matrices are easier to transform due to established algorithms. The first type is of size $N \times N$, the second type is of size $N \times \lceil N/3 \rceil$ to test non-square matrices, and the third type is of size $N - 5 \times \lceil N/3 \rceil$

to test non-square matrices of an uncommon size. The values of N are taken from the array $[16, 64, 256, 431, 512, 1024]$.

The benchmarking of the experiment includes the communication time to and from the GPU. Furthermore, the planner is compared to the regular `fft`-function on both the CPU and the GPU, but separately from one another. The CPU and GPU are compared in the main part of experiment 2, as this experiment was created purely to show the effects of using the planner.

Using the planner is a straightforward process. Arrays are defined as usual, after which the function `plan_fft` is used on the array that is to be transformed. This is shown in the code below:

```
x_d = CuArrays.randn(512,512)
y_d = CuArrays.fill(0.0f0, (512,512))
p_d = plan_fft(x_d)

y_d = p_d * x_d
```

The planner is multiplied with the array, resulting in the desired Fourier transformation.

6.4 Experiment 2: FFT GPU Acceleration

The second experiment has more or less the same premise as the first experiment, only it is more directly correlated to the third main experiment. In this experiment, the Fast Fourier Transform is performed on both the CPU and the GPU for both vectors and matrices of size 2^N by 1 respectively 2^N by 2^N; again N varies.

To do so, the FFTW package, as discussed in chapter 3.3.3, is used. As discussed before, the built-in functions of FFTW support both CPU and GPU arrays, so this doesn't result in any discrepancies when comparing the computation times for both types of arrays. The usage of different packages might have distorted the results.

The planner functionality is incorporated throughout the experiment, per the results of the planner experiment. The rest of the experiment concerns itself with the benchmarking of the Fourier transform on both the CPU and the GPU. Both benchmarking tools as described in Chapter 4 are used. However, during the development of the BenchmarkTools part, initially the size of the matrices was determined using a for-loop. The results were unexpected and not concise with the results of experiment 1, and the size of the matrices maxed out far earlier than the memory would normally allow. It seemed as if using a for-loop completely filled the memory.

To combat that, the sizing and benchmarking of the Fourier transforms was taken out of the for-loop. Furthermore, after the matrices of power 2^8 had been determined and used, the Julia garbage collection command was used to clear out the memory before performing the next computation. (This was the original bottleneck size.) In this case, the experiment maxed out at matrix-size 2^{12} on the laptop. On the desktop this was 2^{14} and for the server 2^{16}.

Because of this, the experiment can be divided into three parts:

1. BenchmarkTools in a for-loop

2. BenchmarkTools rewritten

3. Built-in Julia benchmarking

Before the details of each part are discussed, some general notes about the experiment:

- The planner function for the CPU computation of the Fourier transform includes some rudimentary parallel programming. This can be used by setting the `FFTW.set_num_threads()` variable with an integer value different from 1, but not higher than the number of threads available from the CPU. The planner will then use that amount of threads for its computation. The GPU planner doesn't use this functionality, as it will already use the (far larger) number of threads provided by the GPU.
- The plan is defined as a matrix either on the CPU or the GPU, it cannot be used on both.
- A plan can also be created for the inverse Fourier transform, and can be used in the same way.
- The CPU computation uses double precision values (ComplexF64), the GPU computation uses single precision values (ComplexF32, also denoted by the added `f0` to 0.0). This ensures that each computation plays to its own strengths.

6.4.1 Experiment Parts

BenchmarkTools In A For-loop

This is the original development of experiment 2. The idea was to first determine the CPU computation times, but for a different number of CPU cores. As mentioned before, the `fft`-package also includes parallel Fourier transforms for the CPU. Since the laptop used has a CPU with four cores, the experiment was run for 1, 2 and 4 cores on all devices. The results can be found in the next chapter.

After that, the minimal value of the three CPU computations is taken and compared to the GPU computation.

BenchmarkTools Rewritten

As discussed in the prior section, the experiment seemed to not be fully optimised due to the use of a for-loop. It was decided to rewrite the code so a for-loop was no longer used and instead all the code was written out and it included garbage collection so memory was freed up.

Built-In Julia Benchmarking

Since BenchmarkTools seemed to be flooding all the memory whilst performing computations, the built-in Julia benchmarking was also applied to the experiment to see if this resulted in any differences.

Benchmarking in general as well as the differences in benchmarking between BenchmarkTools and the built-in tools are discussed below.

6.4.2 Benchmarking

Below some more details about the benchmarking used in this experiment.

BenchmarkTools Benchmarking

The general build-up of the experiment is as follows, where S is the size of the matrix as a power of 2:

```
#CPU computation
x = randn(ComplexF64, (S,S))
y = similar(x)
p = plan_fft(x)
comp_times_cpu[i] = BenchmarkTools.mean(@benchmark y = p * x).time

#GPU computation
x_d = CuArrays.randn(S,S)
y_d = CuArrays.fill(0.0f0, (S,S))
p_d = plan_fft(x_d)
comp_times_gpu[i] = BenchmarkTools.mean(
    @benchmark y_d = p_d * x_d).time
```

Built-In Benchmarking

As a counter to the BenchmarkTools package, a separate version of the experiment using the built-in Julia benchmarking tools was also built. It replaces the BenchmarkTools macros with Julia macros. Since there is no comparable macro that runs the expression multiple times and takes the average, each expression is run 100 times in a for-loop and the mean is then taken. This is to ensure that the results of both benchmarking tools are comparable.

6.5 Experiment 3: PWTD GPU Acceleration

Experiment 1 and experiment 2 are more or less precursors for the final experiment, which is the GPU implementation of the Plane-Wave Time-Domain algorithm. They showcase why (partly) applying GPU implementations for the algorithm can speed up its eventual computation.

Experiment 3 consists of a few steps that were followed to develop the GPU implementation. The steps are as follows; each step is discussed in more detail below. In the next chapter, the results of the full implementation are discussed.

Step 1: GPU Implementation Of Fourier Transforms

Step 2: Implementation Of Planner Functionality

Step 3: Rewriting The Convolve Function

Step 4: Eliminating All Unnecessary CPU-GPU Copies

6.5.1 GPU Implementation Of Fourier Transforms

The first part of the GPU implementation is to determine which arrays are important for the FFTs, and if and how they can be transferred to the GPU. The Fourier functions for the Plane-Wave Time-Domain algorithm are actually housed in the SampleArrays.jl package, in a file aptly titled `fourier.jl`. SampleArrays.jl also defines the signals required for the PWTD algorithm to function; the Fourier and inverse Fourier functions in the `fourier.jl` file are specifically written for these signals.

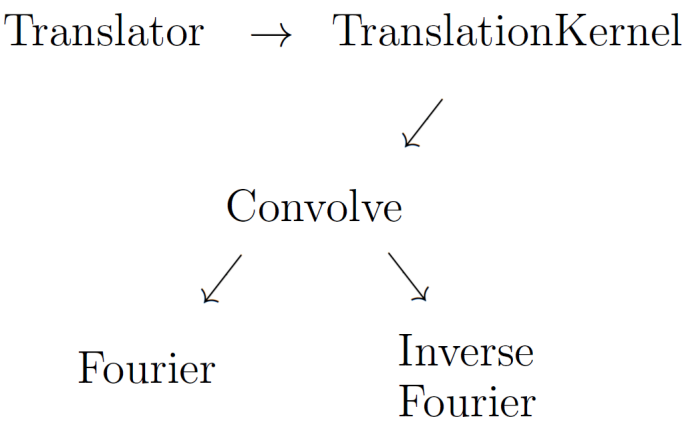
The first step is to add the correct additional packages to the PWTD.jl and SampleArrays.jl packages. These are CuArrays, CUDAnative, CuArrays.CURAND, CuArrays.CUFFT and GPUArrays.

The second step is to determine which arrays have to be copied to the GPU for the FFT to also perform on the GPU. Thus it is important that the correct part of the signal is copied to the GPU and the Fourier function is rewritten in the correct way. Furthermore, any multiplication with a vector or scalar has to be rewritten such that the vector and scalar are also copied to the GPU before multiplication occurs.

The final step is to run a test to see if the new implementation works correctly. In their original forms, the PWTD.jl and SampleArrays.jl packages make use of the regular `fft`-function from the FFTW package. This function can also be used on GPU arrays, making it relatively easy to test the package in its current state.

6.5.2 Implementation Of Planner Functionality

After determining which arrays have to be copied to the GPU, it is time to start implementing the planner functionality into the package. The aim is to use the same planner as much as possible, thus to define it as early as possible in the hierarchy of the package. It is therefore necessary to look at the current hierarchy of functions in the package:



The translator is combined with a bundle of signals that needs to be transformed. Then, for each signal in the bundle a TranslationKernel is determined which is then passed on, together with the signal, into the Convolve function. Here a Fourier transform is performed on the Translation kernel and on the bundle, which are then multiplied. The inverse Fourier transform is then performed on the result. This corresponds to part 2*b* of the Multilevel PWTD algorithm.

The idea is then to define the planner in the Translator, for both the forward and the backward Fourier transform. The planner is then added to each connected function as an extra attribute, until it is used in the Fourier functions. In this way, the planner does not have to be determined for every signal of the signal bundle separately.

Finally, the `fft`-functions in the forward and backward Fourier transforms are replaced with a multiplication of the bundle and the planner, as shown in chapter 5.4.1.

6.5.3 Rewriting The Convolve Function

Sadly only replacing the `fft`-functions by the GPU planner does not work, as the `convolve2`-function in the convolve.jl file of PWTD.jl actually has a differently-defined Fourier transform function for the TranslationKernel. Rewriting this function for the GPU has proven to be quite difficult due to certain functions being CPU-functions only. Therefore, the regular FFT is performed, after which it is copied to the GPU. This is then multiplied with the Fourier transform of the signal, per Theorem 1.

6.5.4 Eliminating All Unnecessary CPU-GPU Copies

To make sure that the GPU-accelerated implementation of the PWTD is as optimal as possible, it is imperative that all CPU-GPU copies that have nothing to do with the accelerated Fourier transform are either circumvented or outright eliminated. Unnecessary CPU-GPU copies are, for example, getting the value of a certain index of a GPU array outside of a CUDA kernel, or performing CPU-only functions on a GPU. Julia will either throw up an error or copy the array to the CPU, after which the function is performed and the data is copied back to the GPU.

When such an unnecessary copy does occur, CUDAnative.jl will throw up the following error:

```
Warning: Performing scalar operations on GPU arrays: This is very
slow, consider disallowing these operations with 'allowscalar(false)'
```

It also shows in which file and line the scalar operation has been performed. Using `allowscalar(false)`, these scalar operations are disallowed and will throw up an error with again the corresponding file and line. In this way, all needless CPU-GPU copies can be eliminated for a more optimal package.

After all these steps have occured, one of the included testing files is timed against the CPU version and benchmarked. The averaged times are then compared.

Chapter 7

Numerical Results

This chapter includes all the results to the numerical experiments as described in Chapter 6. Each experiment has its own section dedicated to it, with subsections included for different versions of each experiment and where otherwise necessary. In each part the experiment is repeated shortly with some added context, after which the results are presented and then discussed. Conclusions are drawn in the following chapter.

Some notes before the results are presented:

- Concerning benchmarking: both the built-in Julia tools as well as the BenchmarkTools.jl-package have been used for benchmarking. Both have their strengths and weaknesses, however, when using the @benchmark macro in a for-loop it gives a CUFFT memory error, due to not enough memory being available. This happens for matrices that take up about 64MB in size, which is small for a graphics card with at least 1 GB of video RAM. It seems as though BenchmarkTools fills the entire available memory with all the matrices it needs for its benchmark before actually performing the benchmark. This has caused some issues with primarily efficient coding but also not being able to scale the experiments to an expected size. Where applicable, it will be discussed again as well as ways to work around the constraint.
- The planner experiment has an entire section dedicated to it. During the development of the experiment the results gave rise to an extra experiment. These have been discussed in more detail in the previous chapter.

7.1 Exp. 1: General GPU Acceleration

In this section the results are presented for the first experiment as described in the previous chapter. During this experiment, various operations were performed on both the CPU and the GPU to show if and when the GPU becomes a more viable option. This experiment was run on the personal desktop, University laptop and the server at Capgemini. The corresponding plots can be found in the tables 7.1 for the desktop, 7.2 for the laptop and 7.3 for the server.

It can be observed that, for all three devices, at a certain point the GPU will be more efficient than the CPU for all seven operations. Furthermore, since the computation time is set at a $\log_{10}$ scale, it can be noted that at certain points the GPU is more than 10 times faster than the CPU. An example of this is the last image in table 7.1. For matrices of size 2^{10} the GPU is $10^{1.7} \approx 50$ times faster!

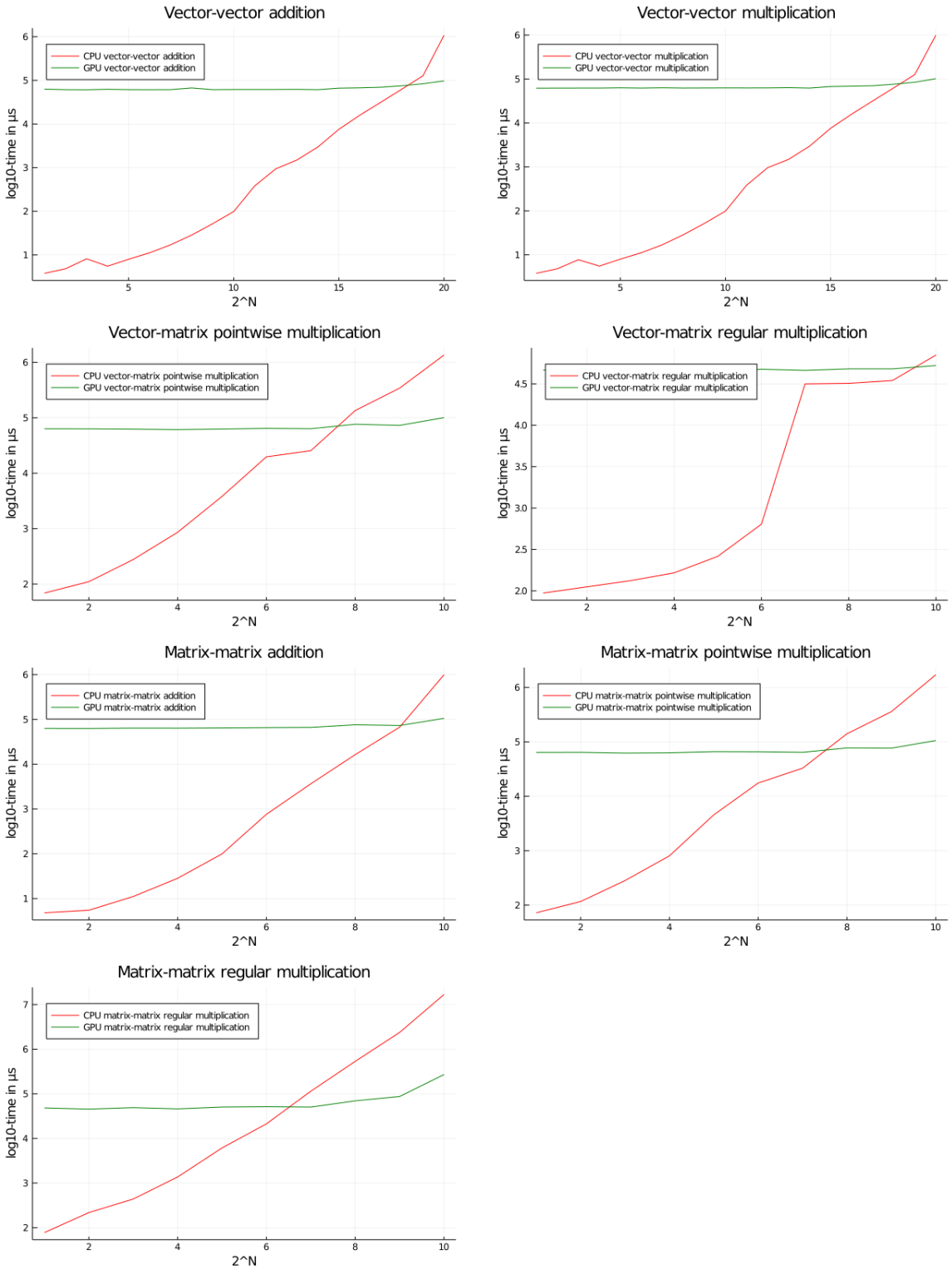


Table 7.1: Results from experiment 1 on the desktop, showing that from a certain matrix- and vector size it is more efficient to use a GPU.

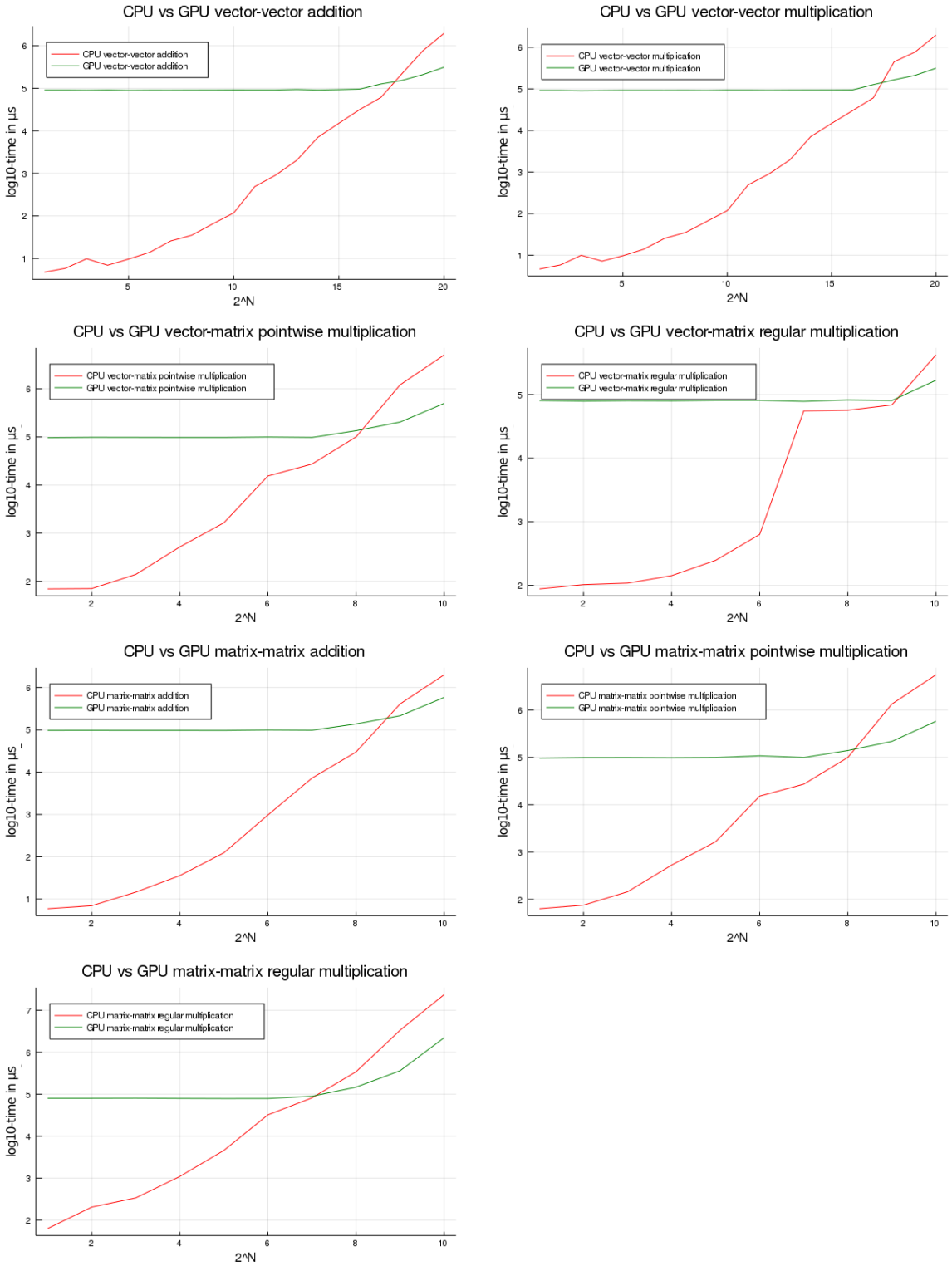


Table 7.2: Results from experiment 1 on the laptop, showing that from a certain matrix- and vector size it is more efficient to use a GPU.

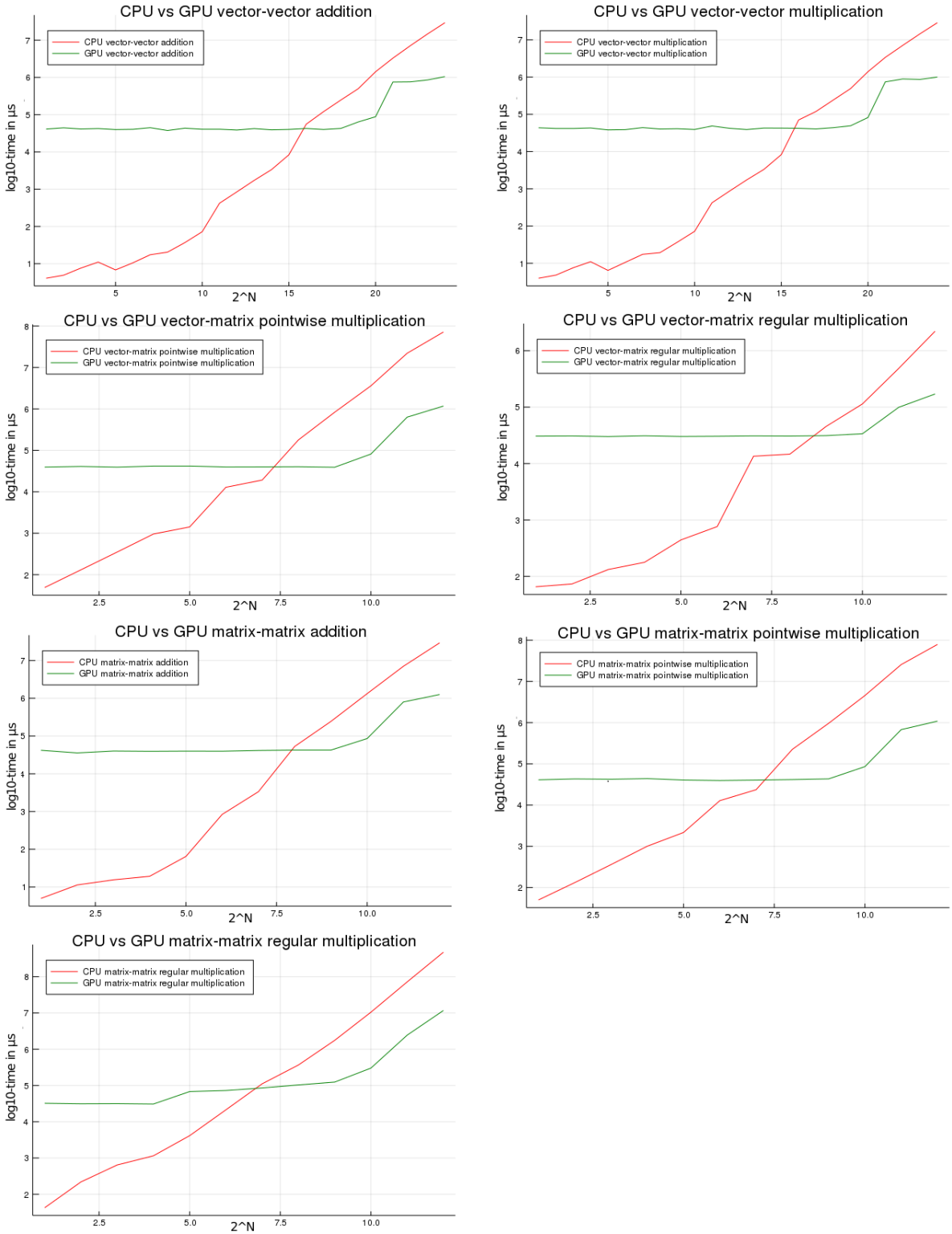


Table 7.3: Results from experiment 1 on the server, showing that from a certain matrix- and vector size it is more efficient to use a GPU.

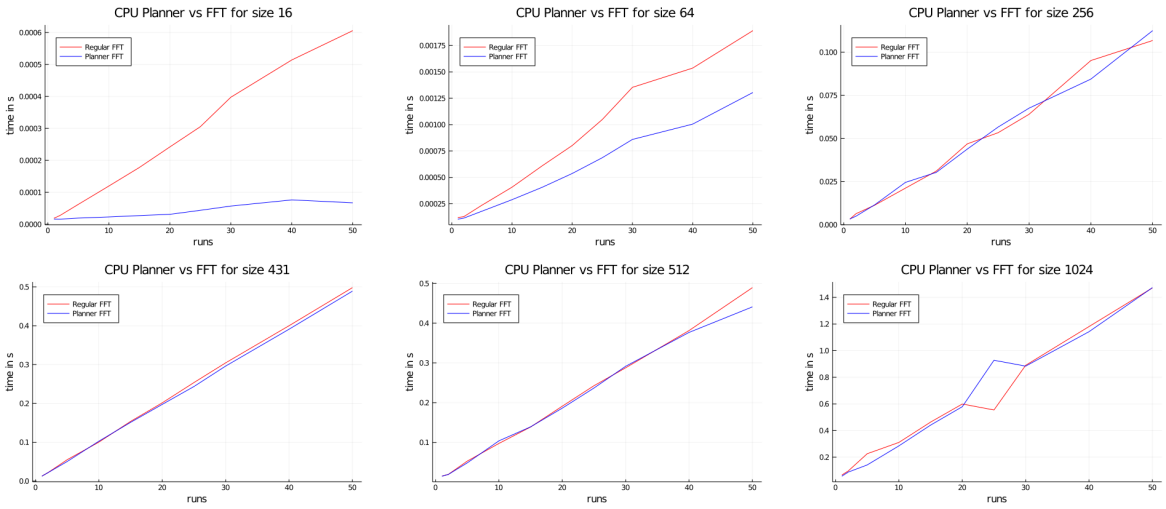


Table 7.4: Planner results for CPU on desktop for matrix type 1. For large matrix sizes, the two FFT methods are tied when it comes to computation time.

7.2 Planner Results

This section shows and discusses the results from the extra experiment that was created to showcase the speed increase that the planner-functionality should provide. The regular `fft()`-function and the planner function were applied to matrices of increasing size, namely of size 16, 64, 256, 431, 512 and 1024, on both the CPU and the GPU. The FFT or planner-matrix was then used to transform this matrix an r number of times, where r stands for the number of runs. For the CPU r runs from 1 to 50, for the GPU from 1 to 10 on the laptop and from 1 to 20 on the server and desktop.

As mentioned in the previous chapter, the experiment was run on three differently sized types of matrices, namely of size $N \times N$, of size $N \times \lceil N/3 \rceil$ and of size $N - 5 \times \lceil N/3 \rceil$, where N is one of the above sizes. The experiment was run on all three devices.

7.2.1 Square Matrices

The results for the first type of matrix on the desktop can be found in the tables 7.4 and 7.5, the results of the laptop can be found in tables 7.6 and 7.7, the results for the server in tables 7.8 and 7.9.
The CPU comparison differs quite a lot from the GPU computation on the laptop. For the matrices of size 16 and 64, the planner is the clear winner. But after that, both the planner and the regular FFT perform almost the same. It is not clear why this seems to be the case; this could be due to the fact that algorithms for powers of 2 are very efficient and straightforward, hence the algorithm determination of the `fft`-function is presumably not computation-heavy, though the same experiment on a matrix of size 431 shows the same result. It does follow that, for the CPU, there is no real preference for the planner or the regular FFT after a certain size, at least for square matrices. This is the case for all three devices.

The laptop GPU is an entirely different case. For all sizes after a certain (low) number of runs, the planner is more efficient than the regular FFT. Do note that each set of runs

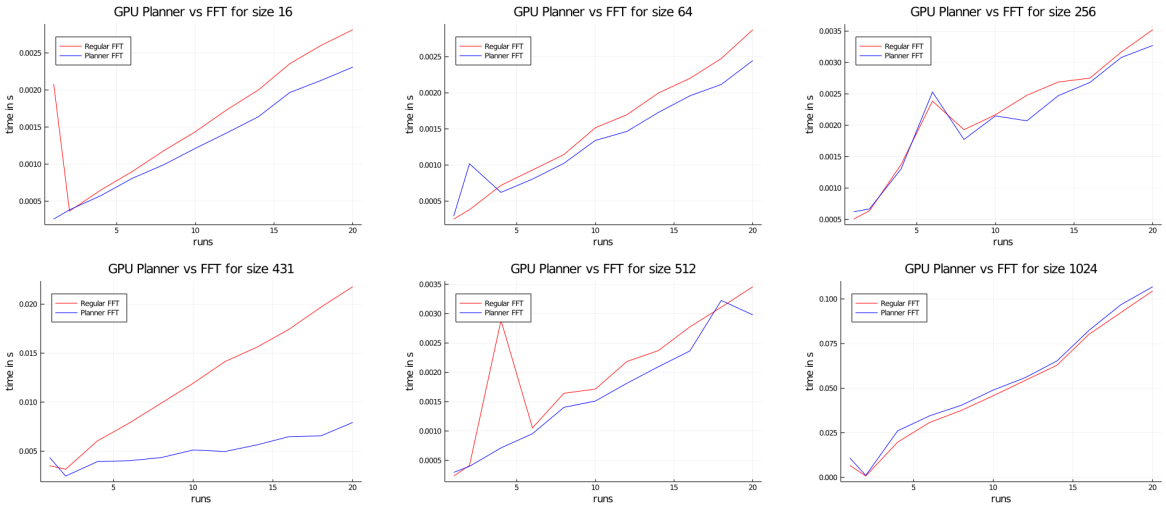


Table 7.5: Planner results for GPU on desktop for matrix type 1. The planner functionality is, in nearly all cases, the better choice.

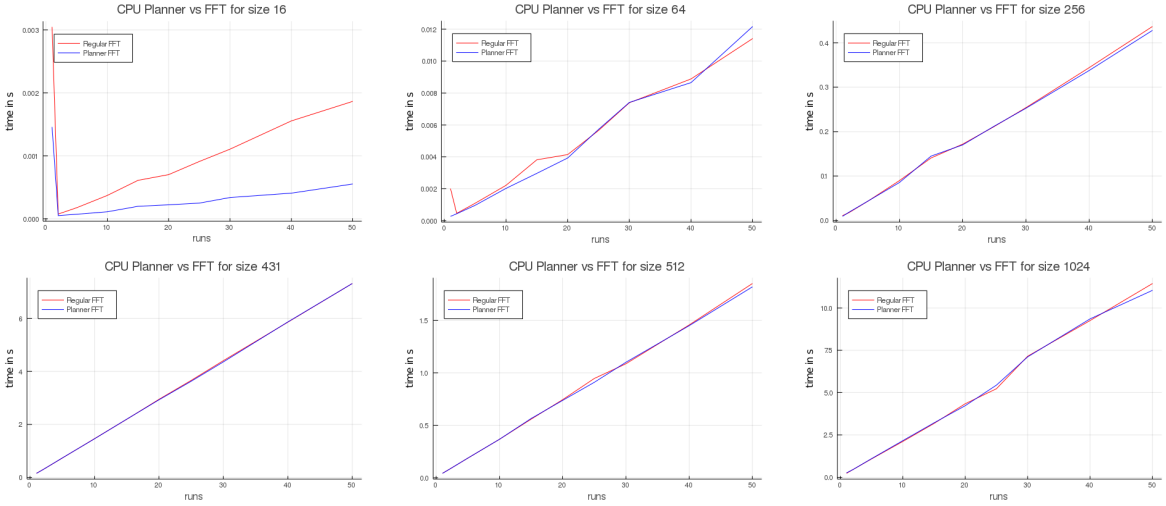


Table 7.6: Planner results for CPU on laptop for matrix type 1. For large matrix sizes, the two FFT methods are tied.

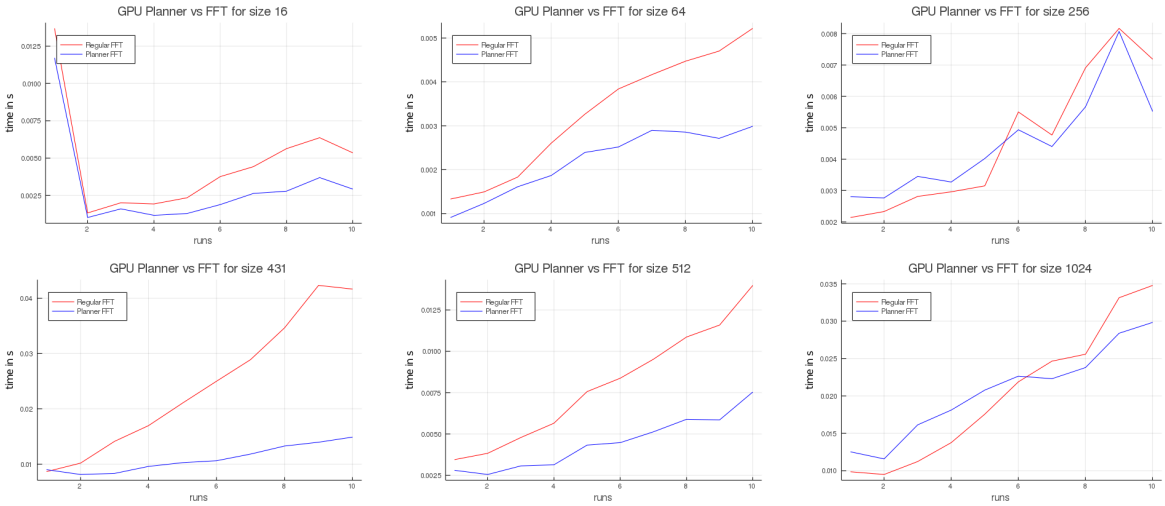


Table 7.7: Planner results for GPU on laptop for matrix type 1. The planner functionality is, in nearly all cases, the better choice.

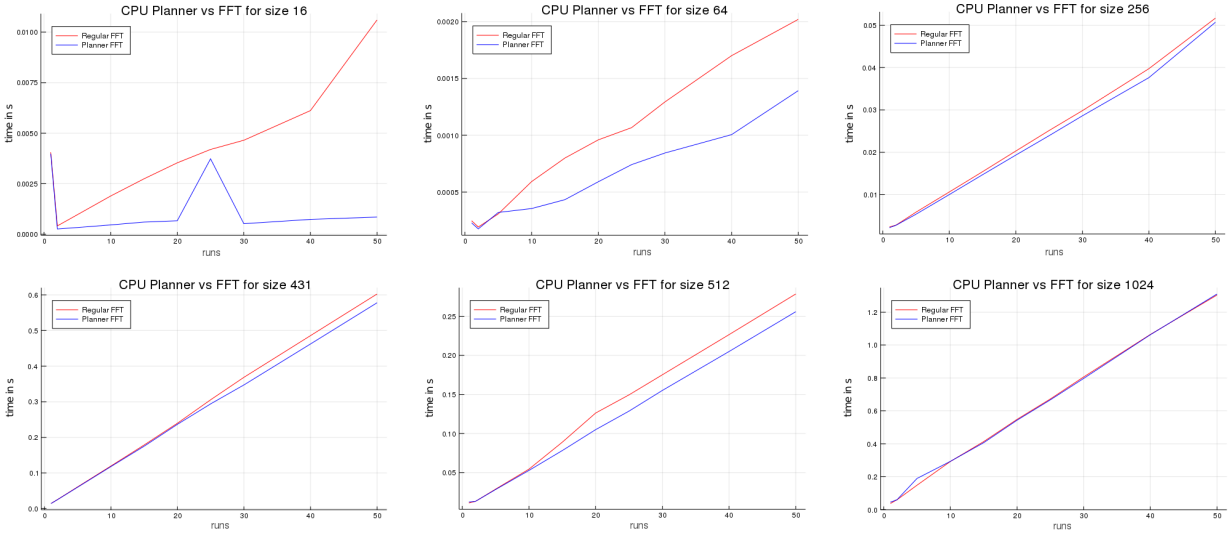


Table 7.8: Planner results for CPU on server for matrix type 1. The planner functionality is, in nearly all cases, the better choice.

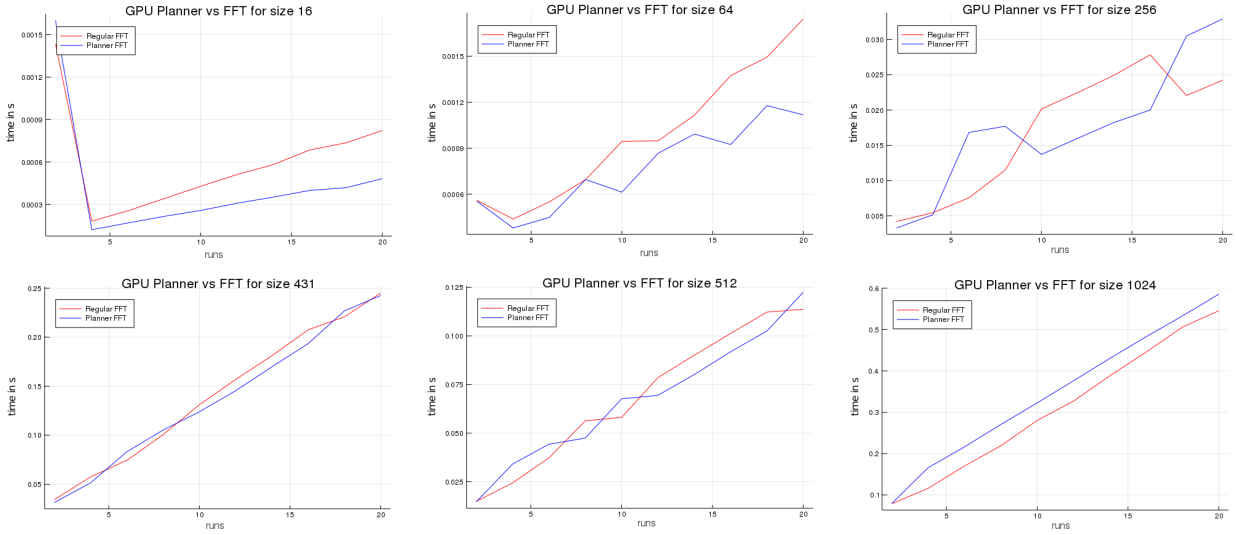


Table 7.9: Planner results for GPU on server for matrix type 1. The planner method seems the better choice in most cases, though the regular FFT method works better with larger sizes.

per size was only performed twice due to memory constraints. It does show that using the planner functionality for the GPU is recommended.

In the first two images of the CPU desktop computation, the planner seems to be the clear winner. However, after the size of the matrix increases, the two methods seem to be tied. Again this might be because FFT algorithms for powers of 2 are relatively easy to determine. On the GPU, the planner seems to be the clear winner in nearly all cases except for the last image, though there the computation times of the two methods are fairly close.

The CPU computations on the server show more or less the same as those on the laptop, though the planner does beat out the regular FFT function more often than not. The GPU on the server is not the clear winner that is expected after the laptop and desktop results. Though the GPU does win from the CPU in about half the cases, in some cases it does not, especially at larger sizes. It is not clear why this is. The experiment has been run on both one and two GPUs to see if this might explain why the GPU isn't always faster, but the results were relatively the same. Other reasons for the discrepancy might be the relative ease with which the regular FFT function can determine the correct algorithm to use for the FFT for square matrices of power 2, or the far speedier CPU that the CPU inhabits compared to the laptop's CPU. Because of this, it was decided to run the experiment on differently sized matrices to see how those results would compare.

7.2.2 Same Size, But Rectangular

In this simulation, the size of the matrix was changed to $N \times \lceil N/3 \rceil$ to keep the power of 2 on one side, but to change the other dimension to something more random. Again the experiment was run on both the CPU and the GPU. Results can be found in the tables 7.12, 7.13, 7.14 and 7.15.

In this situation it can be seen that on all three devices, the planner function is faster or as fast as the regular FFT function on the CPU as the size of the matrix increases. This is

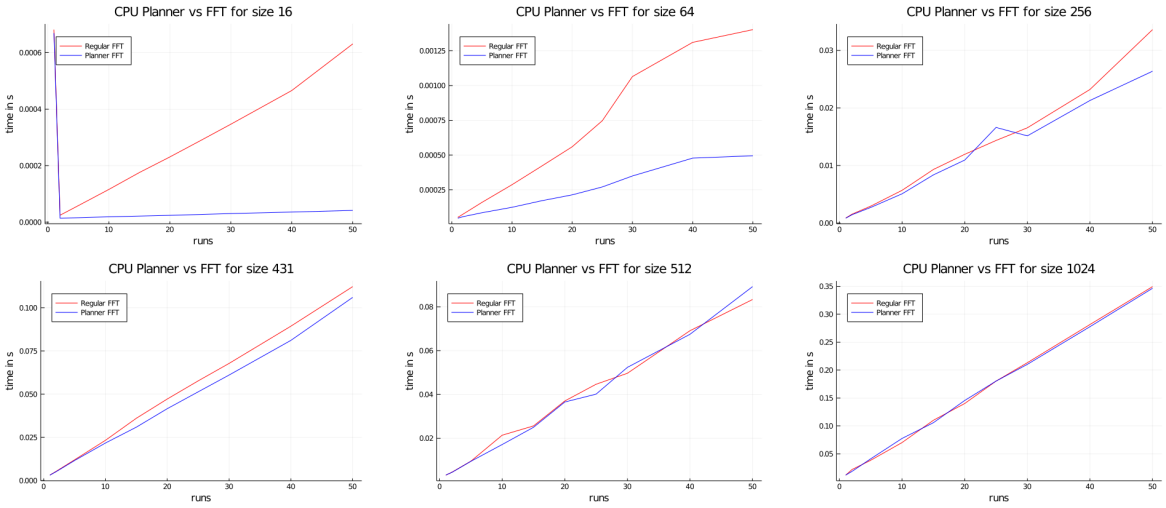


Table 7.10: Planner results for CPU on desktop for matrix type 2. The planner method is the better method in most cases, though for larger sizes the methods are more evenly matched.

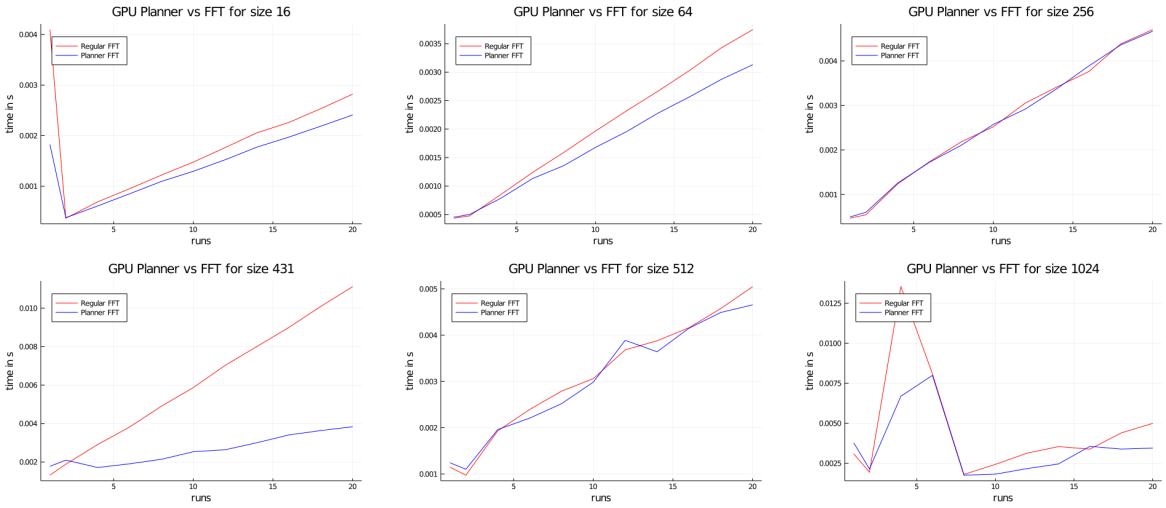


Table 7.11: Planner results for GPU on desktop for matrix type 2. In most cases the planner functionality is the better choice.

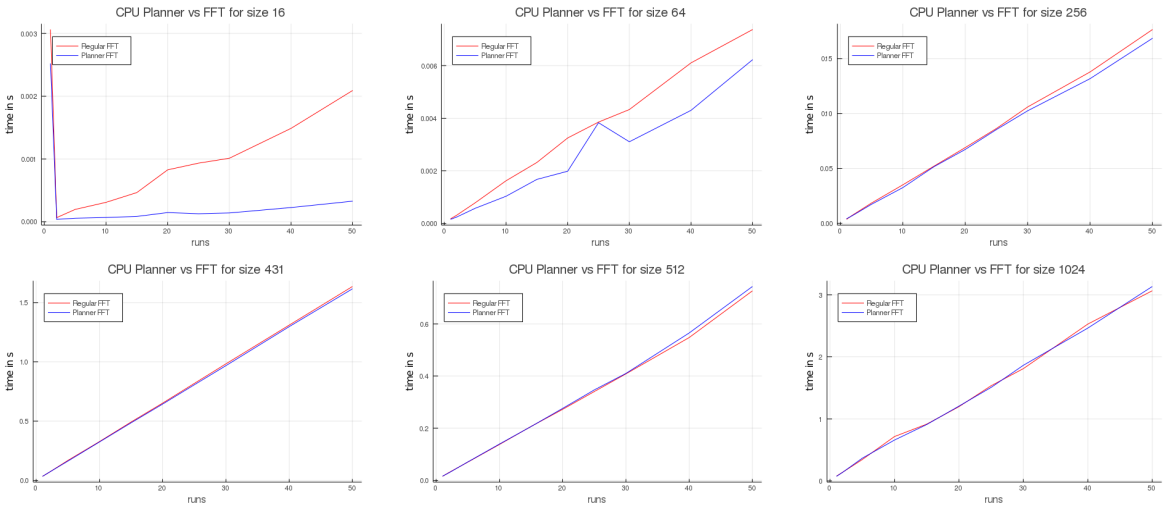


Table 7.12: Planner results for CPU on laptop for matrix type 2. For larger sizes, both methods are tied.

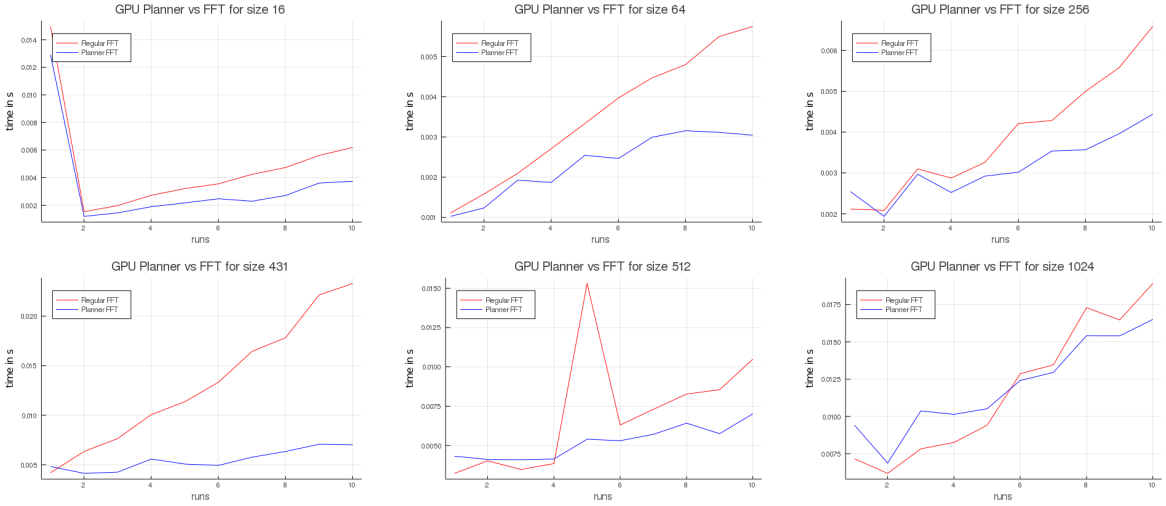


Table 7.13: Planner results for GPU on laptop for matrix type 2. The planner functionality is clearly the better choice.

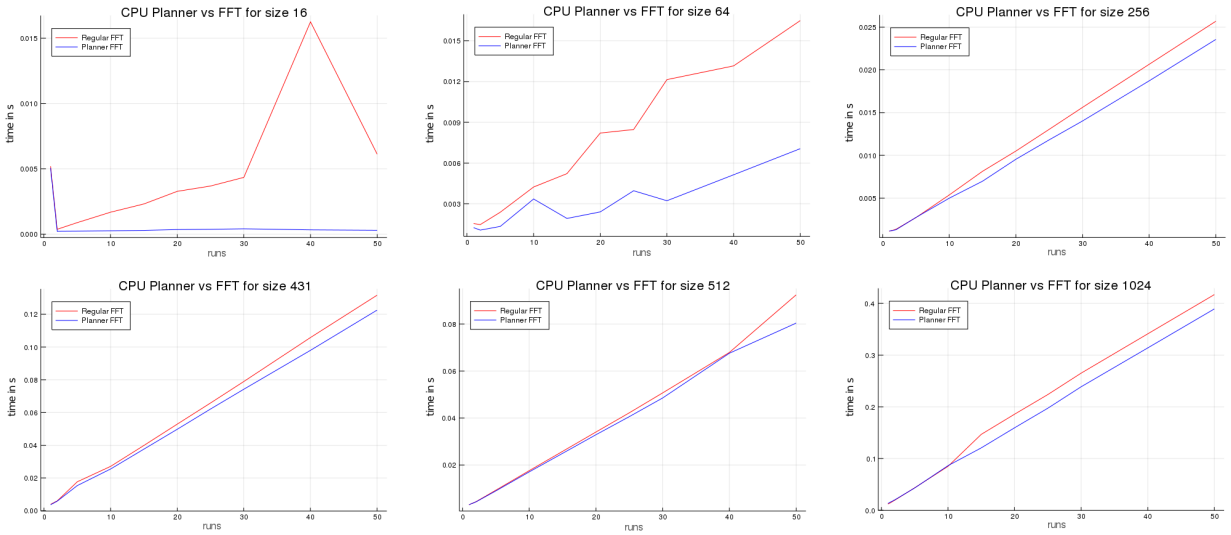


Table 7.14: Planner results for CPU on server for matrix type 2. In all cases the planner is clearly the better choice.

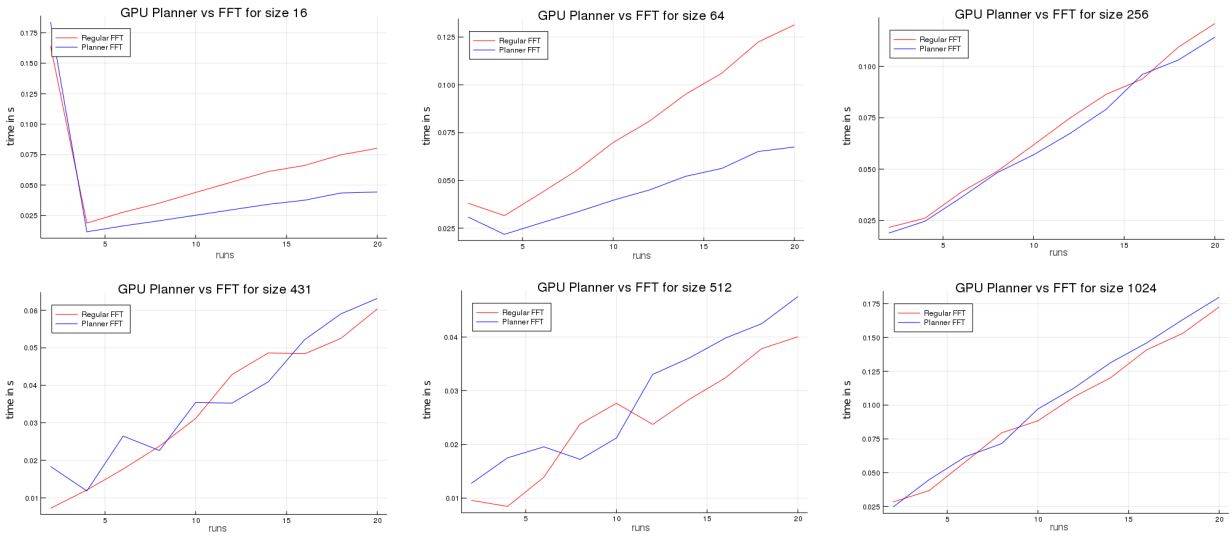


Table 7.15: Planner results for GPU on server for matrix type 2. As before, for smaller sizes the planner method is better, but for larger sizes both methods are somewhat tied.

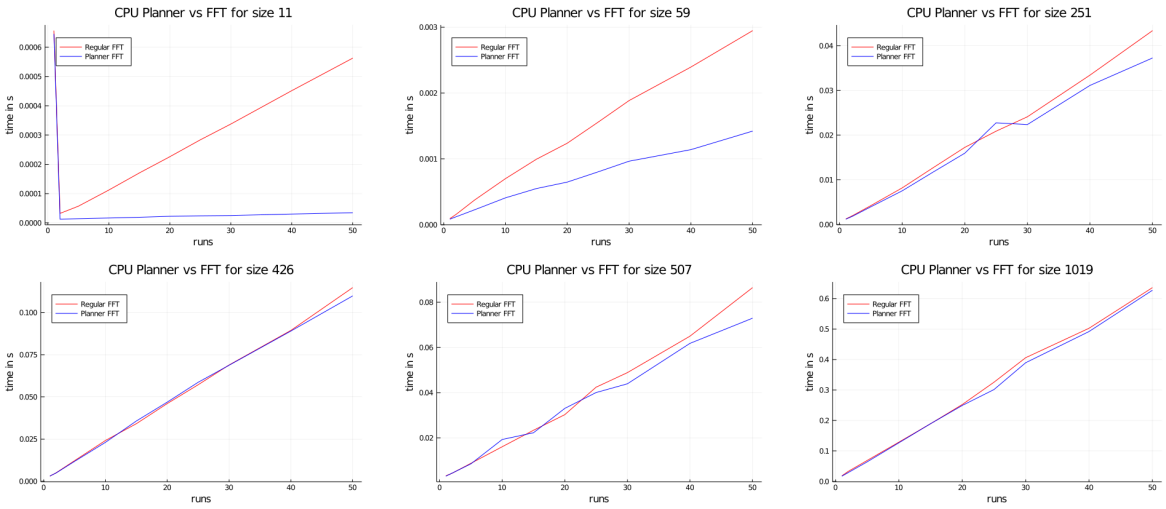


Table 7.16: Planner results for CPU on desktop for matrix type 3. In nearly all cases, the planner functionality is the clear winner.

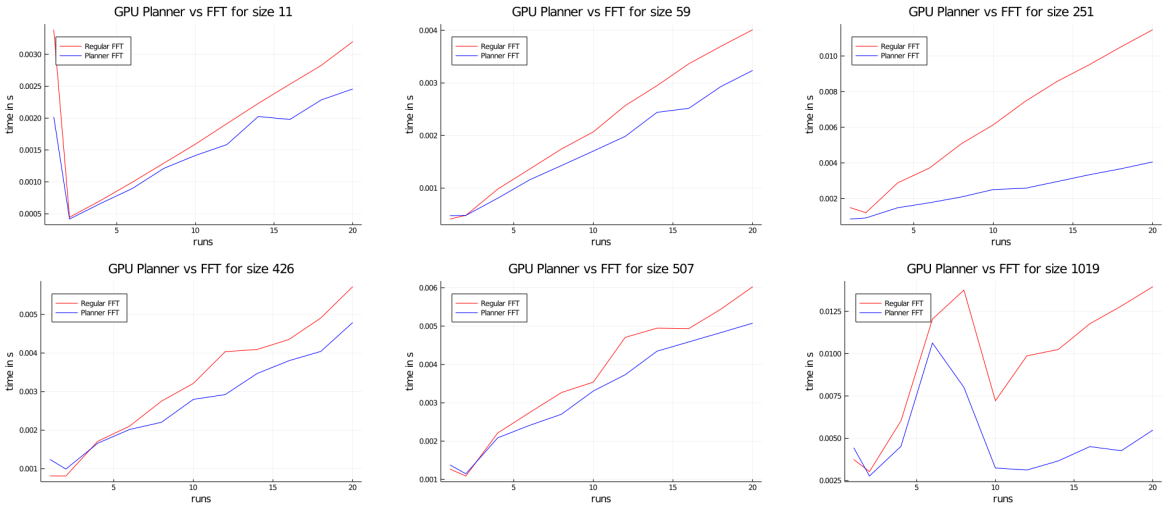


Table 7.17: Planner results for GPU on desktop for matrix type 3. In all cases, the planner method is the better choice.

the same on the GPU for the laptop and desktop, but again the GPU on the server shows that the planner isn't faster in all cases. The results have somewhat improved for the planner, however, as the timings are closer to each other than in the previous situation.

7.2.3 The Third Matrix Type

In this case the size of the matrix was chosen as $N - 5 \times \lceil N/3 \rceil$ so one size would no longer be a power of 2. The results can be found in the tables 7.16, 7.17,7.18, 7.19, 7.20 and 7.21.
In this case, the planner is more or less on par with the regular FFT function on the CPU on the laptop, and considerably faster on the GPU. On the server the planner and regular function are again more or less on par on the CPU, and in this case the GPU usage results in a faster computation of the Fourier transform. For the desktop, the planner functionality is always faster, for both the CPU and the GPU.

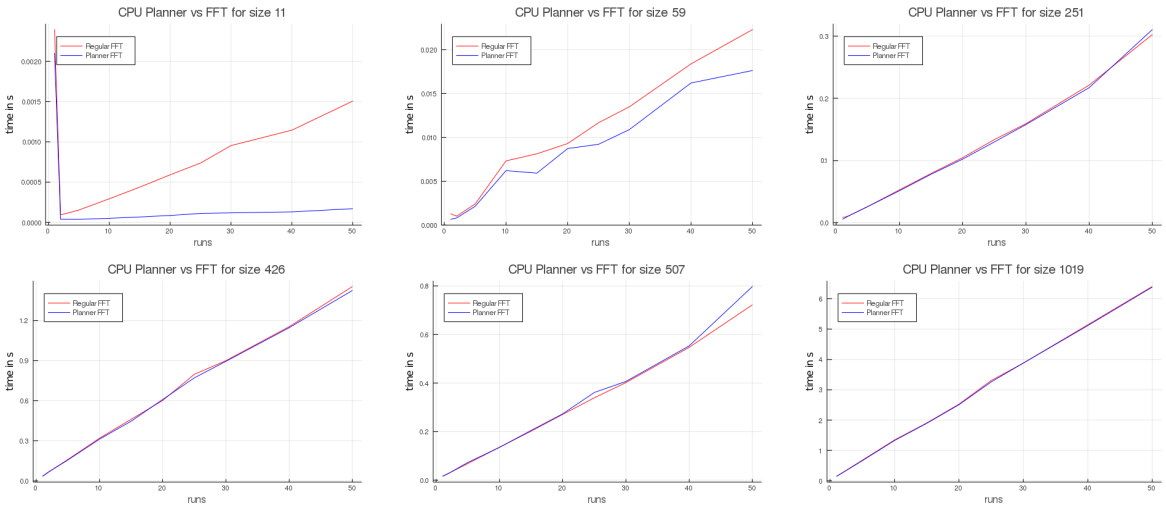


Table 7.18: Planner results for CPU on laptop for matrix type 3. For larger sizes, the two methods are tied.

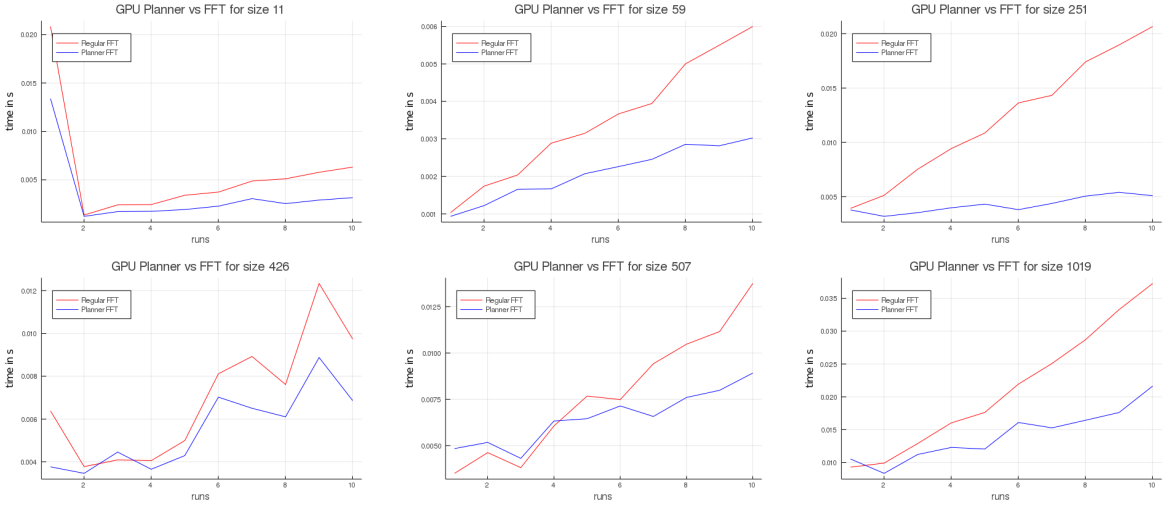


Table 7.19: Planner results for GPU on laptop for matrix type 3. In all cases, the planner method is the better choice.

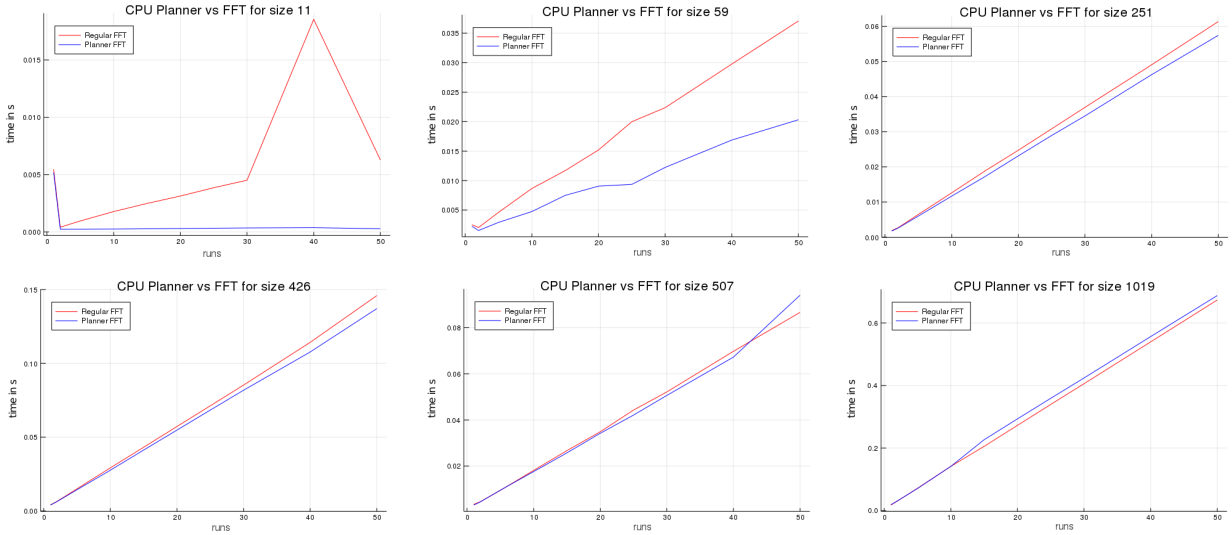


Table 7.20: Planner results for CPU on server for matrix type 3. In nearly all cases the planner method is the better choice, except for the larger sizes.

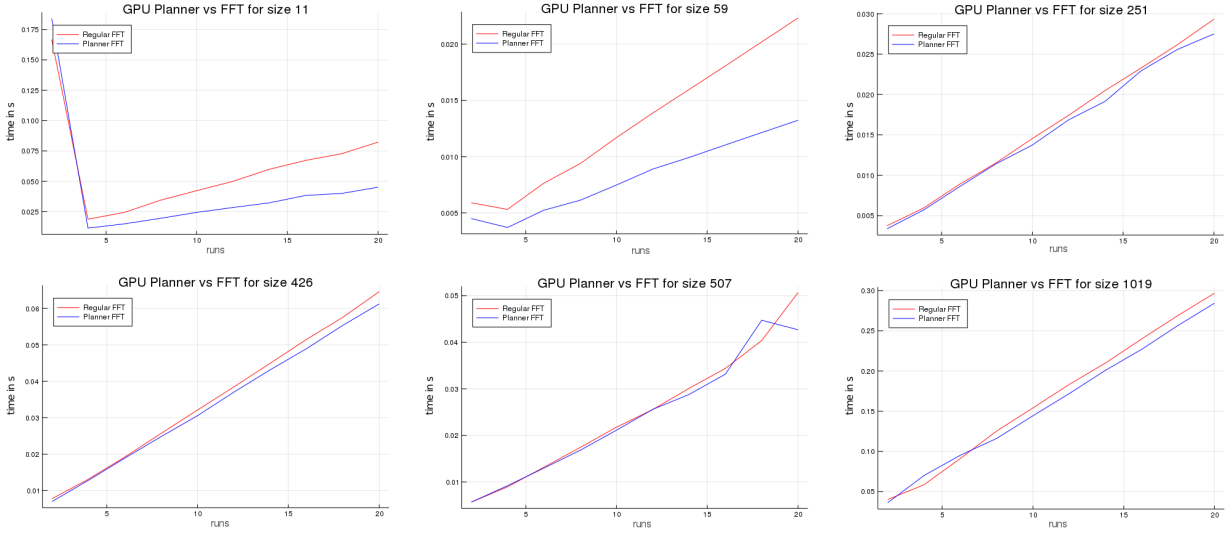


Table 7.21: Planner results for GPU on server for matrix type 3. In all cases, the planner method is the better choice.

It follows that, on the laptop and desktop at least, using the planner on the GPU is always the better option, were as for the CPU there is not always an immediate speed gain using the planner. On the server, the CPU planner and FFT function have relatively the same speed; on the GPU the planner is only noticably faster for irregularly shaped matrices, for square matrices or matrices with one side size power of 2, the planner is not always the faster option.

It is not clear why the latter is the case. This might be attributed to the long start-up time necessary to get operations going on the GPU, or software not being up to date. It can be concluded that, on the laptop and the desktop, using the planner functionality will result in faster computations. For the server there is no definite conclusion.

7.3 Exp. 2: FFT GPU Acceleration

This section functions as a continuation of the previous section, which together cover all the results of the second experiment as described in Chapter 6. The results of the experiments comparing the computation times of the Fourier transform on the CPU and the GPU will be presented and discussed.

As in the prior experiments, the images are in a log-format. The horizontal axis denotes the size of the matrix as powers of 2; thus at point 3, the actual size of the matrix is $2^3 = 8$ and at point 10 the actual size is $2^{10} = 1024$. The vertical axis donates the time as a power of 10, so at point 3 the actual computation time was $10^3 = 1000\mu s$.

7.3.1 Part 1: BenchmarkTools In A For-loop

For each device, the first two images compare the computation time of Fourier transforms on the CPU with 1 core, 2 cores or 4 cores. The remaining two images are plots for the Fourier transform on both a CPU and a GPU. The plots can be found in tables 7.22, 7.23 and 7.24.

On all three devices, it is immediately noticeable that using more cores on the CPU results in a lower computation time after a certain size is reached, though it differs per device when that size is reached.

Regarding the CPU vs GPU computation times, the CPU seems to be the better choice in most cases. This is unexpected, as experiment 1 showed that for basic operations it is almost always better to use a GPU after a certain size. It seems that using BenchmarkTools.jl in a for-loop has resulted in it flooding the memory, hence why the GPU computation times are much larger than expected. Because of this, it was decided to run the experiment again without the for-loop. This is discussed in part 2.

7.3.2 Part 2: BenchmarkTools Rewritten

For experiment 2 part 2, only the FFT on matrices was determined and the number of CPU cores was set to 4. The computation time plots can be found in 7.25 for all three devices. The first plot is from the desktop, the second from the laptop and the third from the server.

From the table it can be noted that for the desktop and laptop, the GPU performs significantly better. The laptop plot does show the GPU computation time climbing at

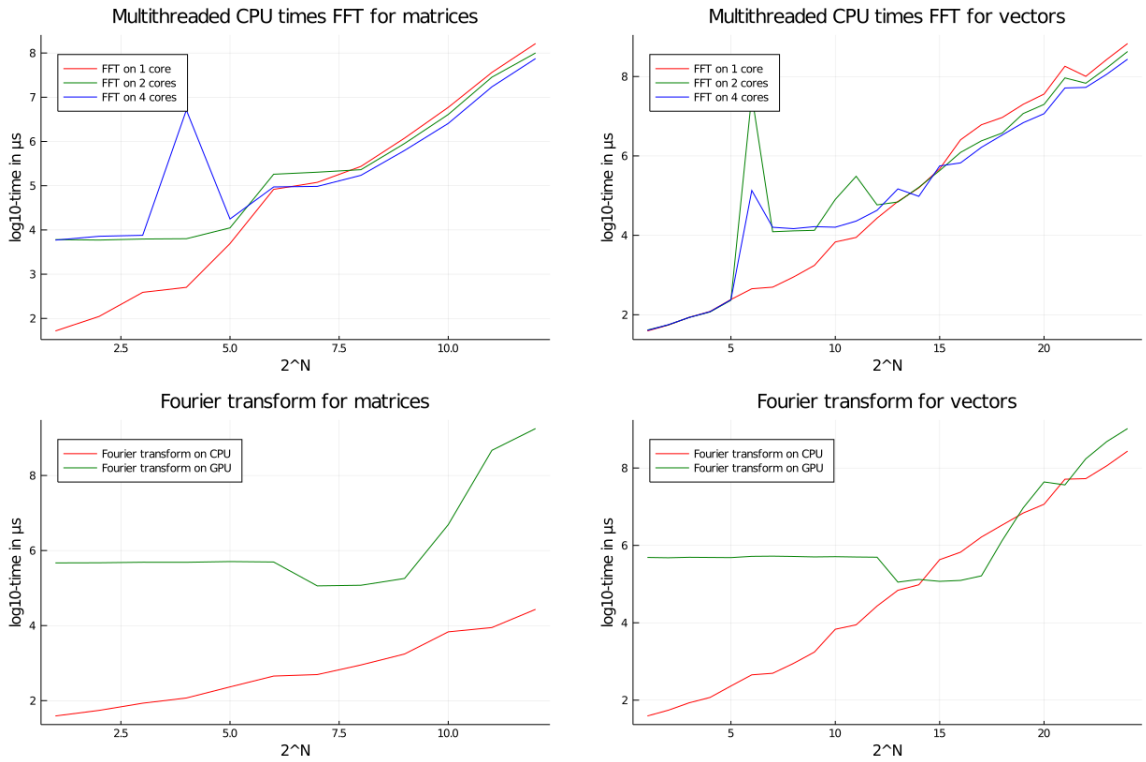


Table 7.22: Results for experiment 2 part 1 on desktop. The top half shows that using more core results in less computation time after $N = 8$ for matrices and $N = 15$ for vectors. The bottom half shows that the GPU is not a better choice for FFTs in this experiment.

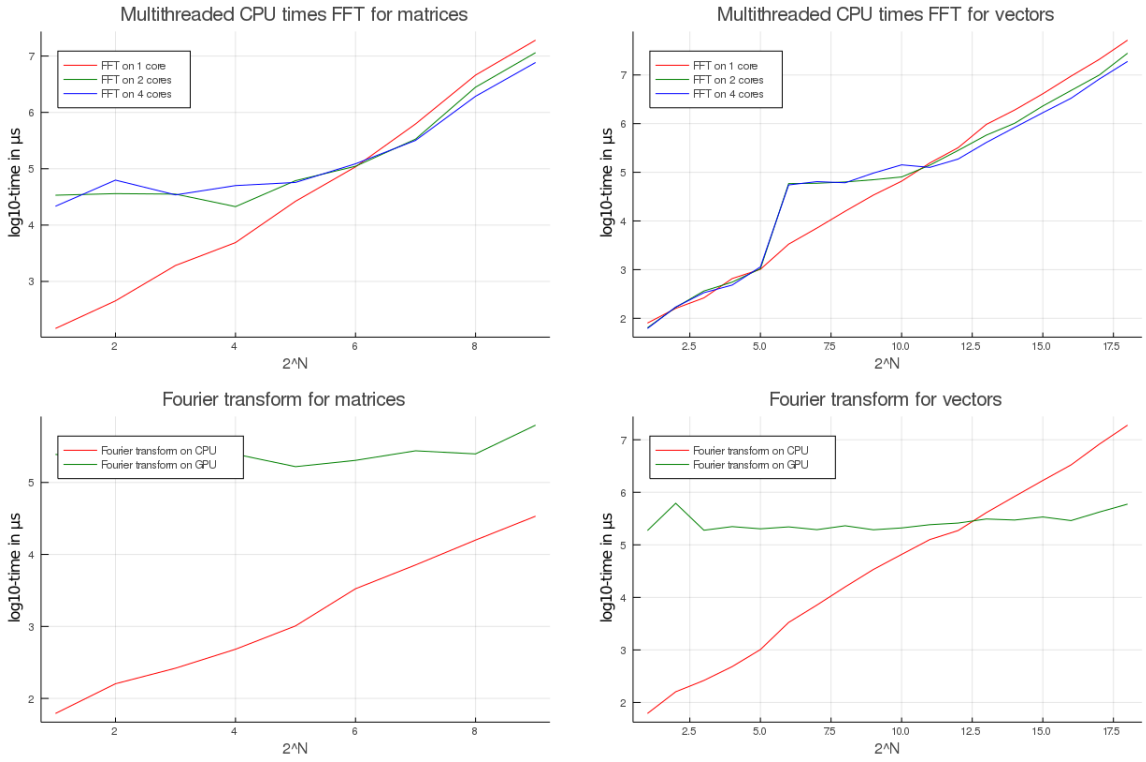


Table 7.23: Results for experiment 2 part 1 on laptop. The top half shows that using more core results in less computation time after $N \approx 6.5$ for matrices and $N = 11$ for vectors. The bottom half shows that the GPU is not a better choice for FFTs on matrices, though for vectors it is the better choice.

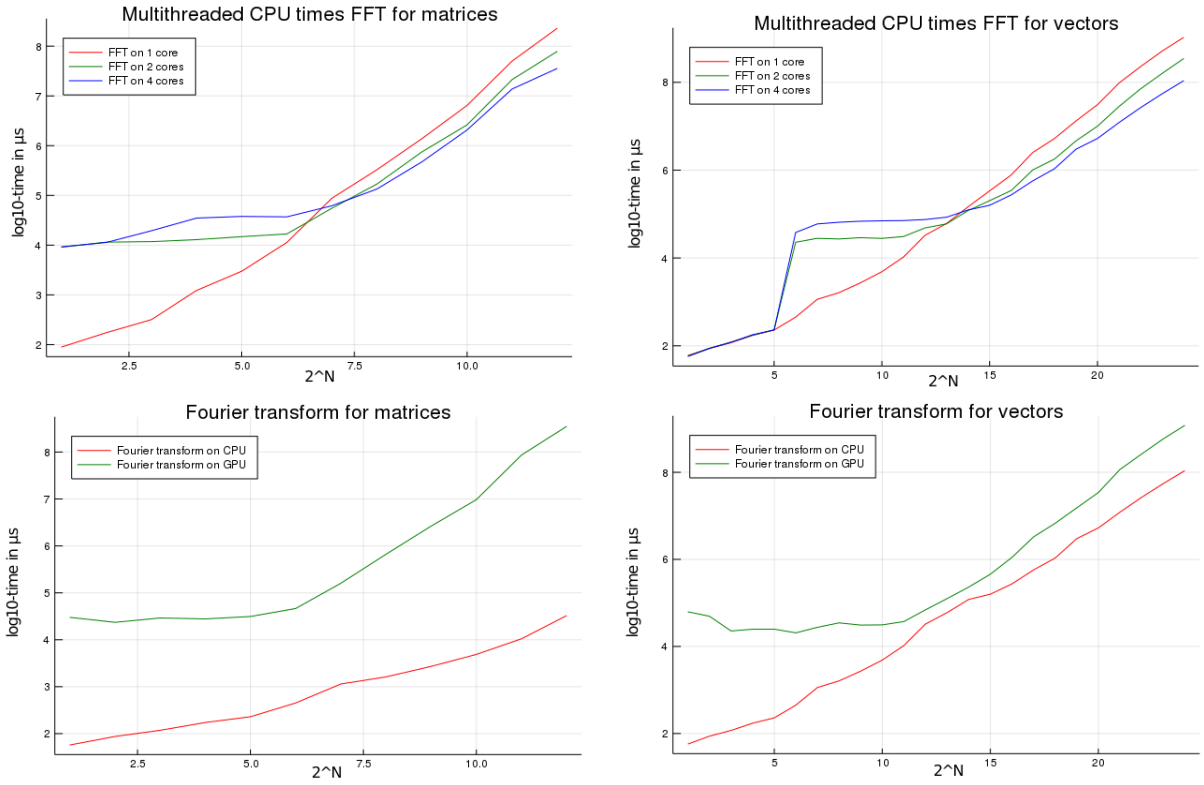


Table 7.24: Results for experiment 2 part 1 on server. The top half shows that using more core results in less computation time after $N = 7$ for matrices and $N = 14$ for vectors. The bottom half shows that the GPU is not a better choice for FFTs in this experiment.

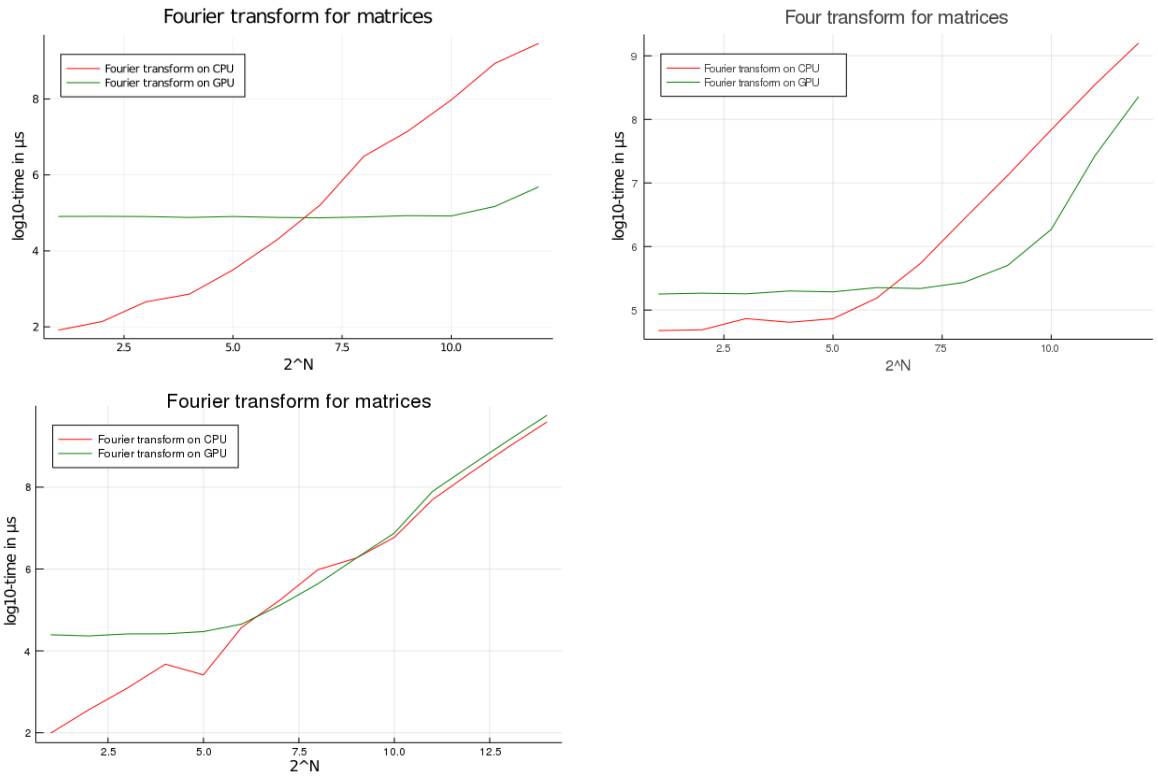


Table 7.25: Results for experiment 2 part 2. On the desktop and laptop the GPU is a better choice for FFTs, on the server the CPU is better by a small margin.

about the same rate as the CPU. This could be attributed to full memory as well as the way BenchmarkTools.jl works. On the server the CPU has, after a certain point, a smaller computation time than the GPU. It is difficult to say why that is; it could be a combination of software not being up to date, the usage of CUDA 9 instead of version 10 or 11, which is more optimised for the Tesla GPU architecture, or the BenchmarkTools.jl package not working properly on the server.

7.3.3 Part 3: Julia Testing

In part 3 of experiment 2 the built-in Julia benchmarking is used to compare the results. The corresponding plots can be found in 7.26. Again the first plot is from the desktop, the second from the laptop and the third from the server.

All three devices show a similar growth for the computation times for both the CPU and the GPU. After size 2^7 the GPU is clearly the better option to determine Fourier transforms. The speed-up is enormous, as the GPU is faster by a factor more than 100.

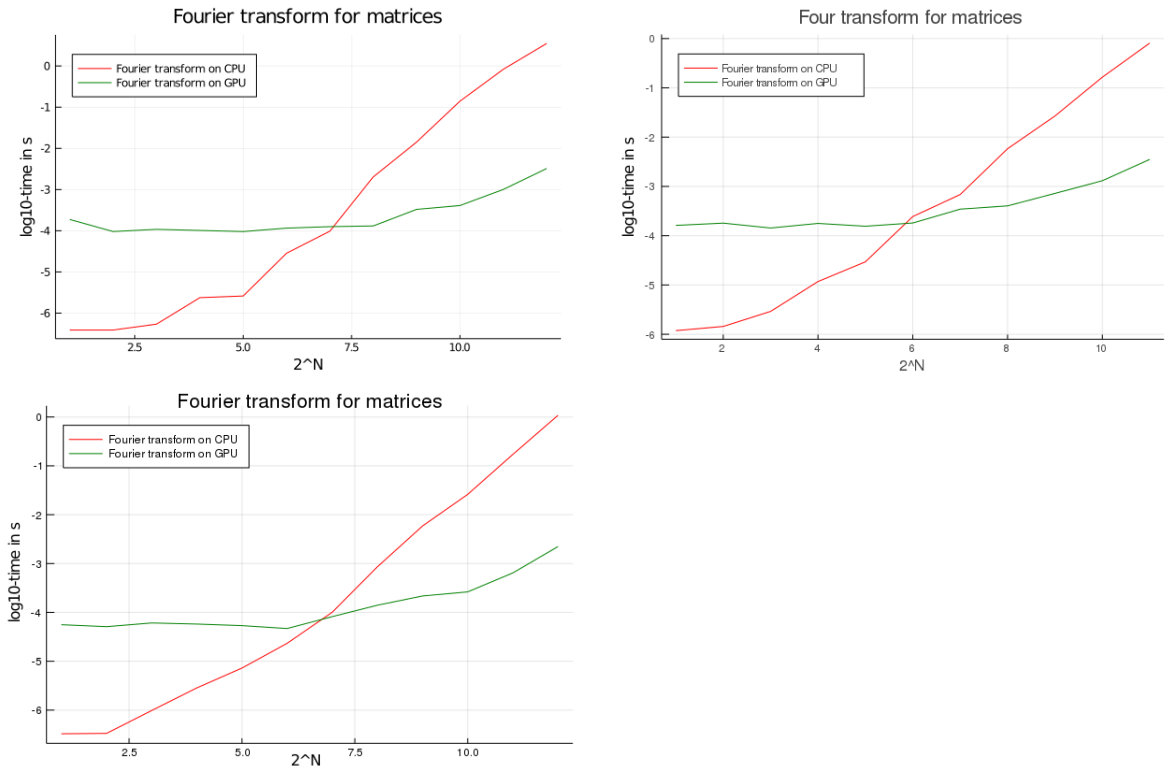


Table 7.26: Results for experiment 2 part 3. For all three devices the GPU is a better choice.

7.4 Exp. 3: PWTD GPU Acceleration

In this section the results of the final experiment are discussed. In this experiment, the PWTD-package was accelerated by coding parts of it and the SampleArrays-package on the GPU. The previous chapter discusses this acceleration in more detail. The actual plot of experiment 3 can be found in figure 7.1. It shows the original wave at the source in blue, and then the resected wave at the observer in orange. The subsignals that make up the wave are also plotted.

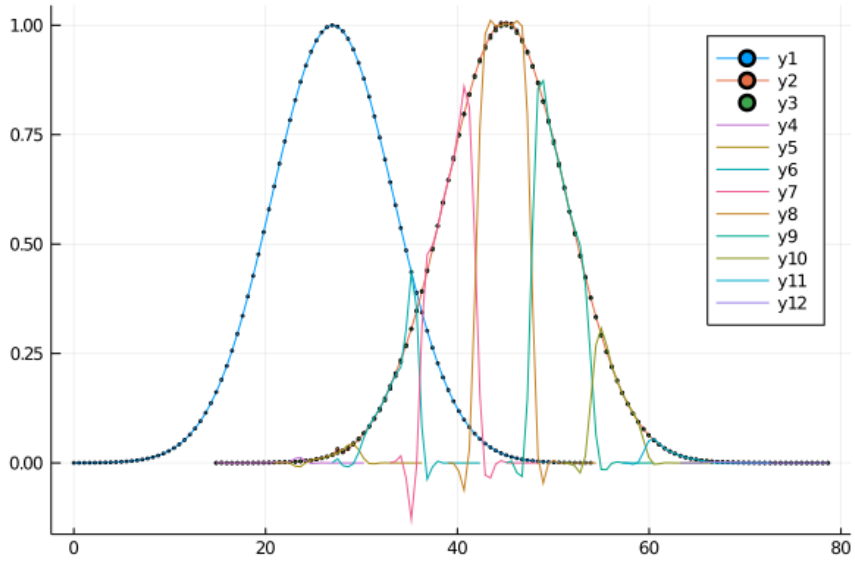


Figure 7.1: Plot of the single source signal problem used in experiment 3.

To test the GPU package against the regular CPU package, the file `singlesource7.jl` was run ten times using both packages on the same machine, after which the averaged time is compared. In this example file a wave is sent from a single source, which is then observed at a different point. Testing was done on the University laptop and on the desktop. Testing was also attempted on the Capgemini server, but the package would not start correctly. Furthermore, it is obvious from prior experiments that in its current state the server does not optimally make use of the GPU.

The results for the CPU and GPU can be found in table 7.27, and are in seconds. The laptop results have about a 6.5 second difference on average; the desktop is very close, with the GPU implementation only 0.32 seconds slower on average.

Run	**CPU Desktop**	**GPU Desktop**	**CPU Laptop**	**GPU Laptop**
Run 1	70.338	69.476	106.687	112.426
Run 2	70.323	70.848	107.177	113.137
Run 3	70.440	70.906	107.329	113.605
Run 4	70.584	71.257	108.372	112.631
Run 5	70.518	70.844	108.937	120.713
Run 6	70.112	71.111	110.482	114.171
Run 7	70.660	71.003	108.215	115.726
Run 8	70.452	71.101	109.589	114.084
Run 9	69.990	70.927	107.819	115.153
Run 10	70.524	69.663	106.747	114.714
Average	**70.394**	**70.714**	**108.134**	**114.636**

Table 7.27: Benchmark times of Experiment 3 on both the CPU and the GPU

Chapter 8

Conclusion

In this chapter the conclusion to the thesis is discussed. This is done by looking at the original thesis questions and discussing the answers by using the results from the previous chapter as well as discoveries made during the research of the thesis. As a reminder, the original thesis questions can be found below.

The main question for this master thesis is as follows:

> *By how much can the computational time of the Plane-Wave Time-Domain algorithm be reduced by using GPUs?*

The sub-questions for this question are:

- *How can the performance of the FFT on GPUs best be optimised?*
- *How can the number of (and concurrently the computational costs and time from) transferals of data between CPU and GPU best be minimised?*
- *By what other means can the PWTD algorithm be optimised?*

First the sub-questions will be discussed, after which the main thesis question will be answered.

How can the performance of the FFT on GPUs best be optimised?
The FFTW documentation mentions planner functionality, which can speed up the computation of multiple FFTs of the same size by predefining the correct algorithm for the FFT. As the planner experiment has shown, using the planner can decrease the computation time for FFTs, therefore the usage of the planner is one of the better ways to improve performance of FFTs on the GPU. Do note that this only works if multiple FFTs of the same size are determined using the planner.

How can the number of transferals of data between CPU and GPU best be minimised?
The best way to minimise the number of transferals between the CPU and the GPU is to incorporate as much as possible into the GPU kernels. This means a transferal of data from the CPU to the GPU as early as possible, with as much of the code running via GPU kernels as possible. Transferring important variables, such as the planner on the GPU, as early is possible is also important.

By what other means can the PWTD algorithm be optimised?
Besides using the GPU, the literature [3] mentions the Windowed Plane-Wave Time-Domain algorithm. It describes a way to shorten the translation functions. Theoretically

this reduces the computation times to $\mathcal{O}(N_t N_s^{4/3} \log N_s)$ and $\mathcal{O}(N_t N_s \log N_s)$ for the two-level respectively the multilevel windowed PWTD algorithm. Due to the added extra implementation complexity this algorithm was not implemented.

Now to answer the main thesis question. From experiment 2, it is clear that the GPU can reduce the Fast Fourier Transform on average by a factor 10, after a certain size matrix or vector is computed. Therefore, theoretically, the cost of step 2 in the two-level and multilevel algorithm is reduced by a factor 10.

Per chapter three, the complexity for the two-level algorithm is $\mathcal{O}(N_t N_s^{1.5})$ in step 1 and $\mathcal{O}(N_t N_s^{1.5} \log N_s)$ in step 2. This can be interpreted as the number of operations being equal to $C_1 N_t N_s^{1.5}$ in step 1 and equal to $C_2 N_t N_s^{1.5} \log N_s$ in step 2, where C_1 and C_2 are contants. After the GPU implementation, the number of operations will be equal to $\frac{C_2}{10} N_t N_s^{1.5} \log N_s$. Thus the total complexity of the two-level algorithm is then $\max\left\{C_1 N_t N_s^{1.5}, \frac{C_2}{10} N_t N_s^{1.5} \log N_s\right\}$ which is smaller than the original complexity of $\mathcal{O}(N_t N_s^{1.5} \log N_s)$. This is assuming no extra complexity due to transfer times from the CPU to the GPU and back.

For the multilevel algorithm, the number of operations in the first step stays equal to $C_3 N_t N_s$. After using the GPU implementation, the number of operations in step 2 will go from $C_4 N_t N_s \log^2 N_s$ to $\frac{C_4}{10} N_t N_s \log^2 N_s$. C_3 and C_4 are again constants. The total number of operations is then equal to $\max\left\{C_3 N_t N_s, \frac{C_4}{10} N_t N_s \log^2 N_s\right\}$, again assuming that there is no additional complexity due to the transferal of data between CPU and GPU. This is again smaller than the original complexity of $\mathcal{O}(N_t N_s \log^2 N_s)$.

Theoretically it is therefore expected that, using the GPU implementation of the FFT, the algorithm will perform quite a bit faster when using a GPU. Actually running the algorithm shows that, with the tested situations, the current GPU implementation is not faster. In the case of the laptop, it is slower by about 6%. On the desktop, however, the GPU implementation is only slower by about 0.003% and is thus nearly on a par with the CPU implementation. If the PWTD algorithm were implemented with GPUs in mind from the beginning instead of implementing it later, a larger speed-gain could be witnessed.

In conclusion, theoretically a (partly-)GPU implementation is faster than a CPU implementation of the PWTD algorithm by a factor of about 10, assuming the correct bundle size for the FFT and no copying from the CPU to the GPU and vice versa. The current GPU implementation is nearly on a par with the CPU implementation on one of the used devices, therefore it is certainly possible to create a full GPU implementation that is faster than the CPU implementation.

Chapter 9

Recommendations and Future Work

In this chapter the results and conclusion are discussed and put into context. Following that, several recommendations are given for the continuation of this research as well as for readers considering using GPUs in their (future) research.

9.1 Recommendations

The results are fairly conclusive: in its current state, the GPU-accelerated Plane-Wave Time-Domain algorithm does not provide the desired acceleration of the algorithm. Though it is not that much slower than the regular algorithm, especially on the desktop, it is still disappointing that no acceleration can be measured, notably as the prior experiments are rather promising and even show an acceleration of a factor larger than 10 for square matrix FFTs using the planner functionality.

It is, of course, important to place the acceleration of experiment 2 in the right context: the operation on the GPU itself is performed faster than on the CPU, however, this does not include the transfer of the CPU-array to the GPU and back before and after the Fourier transform. For small arrays and matrices this can be a problem, as the transfer times can then be larger than the gains made by performing the operation on the GPU. It is very likely that that is what happened here as there are quite a lot of transfers of arrays from the CPU to the GPU and vice versa. It also depends on the speed of the device used. The desktop has faster and more modern hardware than the laptop, and the GPU is nearly on par with the CPU implementation. The larger bandwidth for the GPU, CPU and SSD mean that data can be transferred a lot faster, which most likely contributed to the smaller difference between timings of the CPU and GPU for experiment 3.

As mentioned before, rewriting the axis-functionality for the GPU proved too difficult to implement as not only would the axis-function have to be changed into a GPU kernel, but many other arrays would also have to be copied to the GPU as well. Undoubtedly this would have resulted in many more parts of the code having to be changed to accommodate this, thus requiring almost a complete rewrite of the PWTD package. Building the entire implementation for the GPU from the ground up is a better idea.

Another thing to consider is the nature of the testfile that was used for experiment 3. It contains only a single source, making it one of the easier situations for the algorithm to determine. A more complex situation will require the determination of a more difficult Fourier transform, where the GPU will provide a greater speed-up.

Another thing worth mentioning is the performance of the server. This was, quite frankly, disappointing. This is most likely due to the fact that the software of the server could not be updated easily and the system was therefore stuck at CUDA version 9. Nvidia has made many improvements in versions 10 and 11 of CUDA, especially for the GPUs in the server. It is therefore disheartening that the server could not be used to its fullest potential.

9.2 Future Work

When creating a GPU implementation of an algorithm, it is recommended to do so from scratch and not by trying to improve an already existing package. Though the results are somewhat promising, as for the desktop they are nearly on a par with the CPU, there were several limitations that could not easily be circumvented due to how the algorithm was implemented. Therefore it is better to plan out an implementation with the GPU in mind.

Chapter 10

Appendix

10.1 Code

10.1.1 Experiment 1

```
using CuArrays, CUDAnative, AbstractFFTs, FFTW, Test
using BenchmarkTools
using GPUArrays
using CuArrays.CURAND
using Random
using CSV, DataFrames
using Plots

include("CPUFunctions.jl")
include("GPUFunctions.jl")
include("plotting.jl")

l = 10
A = zeros(l)
B = zeros(2*l)
for i=1:l
    A[i] = 2^i
    B[i] = 2^i
    B[i+l] = 2^(i+l)
end

times_cpu_vec=zeros(length(B), 2)
times_gpu_vec=zeros(length(B), 2)
times_cpu_mat=zeros(length(A), 5)
times_gpu_mat=zeros(length(A), 5)

for i=1:length(B)
    N=Int(B[i])
    @show i
    # Random arrays of length N for CPU computations
    x = randn(Float64,N)
```

```
    y = randn(Float64,N)
    z = zeros(N)
    # Random arrays of length N for GPU computations
    x_d = CuArrays.rand(N,)
    y_d = CuArrays.rand(N,)
    z_d = CuArrays.fill(0.0f0, (N,))

    # Vector-vector addition
    times_cpu_vec[i,1] =
        BenchmarkTools.mean(@benchmark cpu_add_vecvec!($x, $y, $z)).time
    times_gpu_vec[i,1] =
        BenchmarkTools.mean(@benchmark gpu_add_vecvec!($x_d, $y_d, $z_d)).time
    # Vector-vector pointwise multiplication
    times_cpu_vec[i,2] =
        BenchmarkTools.mean(@benchmark cpu_multi_vecvec!($x, $y, $z)).time
    times_gpu_vec[i,2] =
        BenchmarkTools.mean(@benchmark gpu_multi_vecvec!($x_d, $y_d, $z_d)).time
end

for i=1:length(A)
    N=Int(A[i])
    @show i
    # Random arrays of length N for CPU computations
    x = randn(Float64,N)
    y = randn(Float64,N)
    z = zeros(N)
    M = randn(Float64, (N,N))
    K = randn(Float64, (N,N))
    L = zeros(N,N)
    # Random arrays of length N for GPU computations
    x_d = CuArrays.rand(N,)
    y_d = CuArrays.rand(N,)
    z_d = CuArrays.fill(0.0f0, (N,))
    M_d = CuArrays.rand(N,N)
    K_d = CuArrays.rand(N,N)
    L_d = CuArrays.fill(0.0f0, (N,N))

    # Matrix-matrix addition
    times_cpu_mat[i,1] =
        BenchmarkTools.mean(@benchmark cpu_add_matmat!($M, $K, $L)).time
    times_gpu_mat[i,1] =
        BenchmarkTools.mean(@benchmark gpu_add_matmat!($M_d, $K_d, $L_d)).time
    # Vector-matrix pointwise multiplication
    times_cpu_mat[i,2] =
        BenchmarkTools.mean(@benchmark cpu_multi_vecmat_point!($x, $M, $L)).time
    times_gpu_mat[i,2] = BenchmarkTools.mean(
        @benchmark gpu_multi_vecmat_point!($x_d, $M_d, $L_d)).time
    # Vector-matrix regular multiplication
```

```
    times_cpu_mat[i,3] =
        BenchmarkTools.mean(@benchmark cpu_multi_vecmat_reg!($x, $M, $z)).time
    times_gpu_mat[i,3] = BenchmarkTools.mean(
        @benchmark gpu_multi_vecmat_reg!($x_d, $M_d, $z_d)).time
    # Matrix-matrix pointwise multiplication
    times_cpu_mat[i,4] =
        BenchmarkTools.mean(@benchmark cpu_multi_matmat_point!($M, $K, $L)).time
    times_gpu_mat[i,4] = BenchmarkTools.mean(
        @benchmark gpu_multi_matmat_point!($M_d, $K_d, $L_d)).time
    # Matrix-matrix regular multiplication
    times_cpu_mat[i,5] =
        BenchmarkTools.mean(@benchmark cpu_multi_matmat_reg!($M, $K, $L)).time
    times_gpu_mat[i,5] = BenchmarkTools.mean(
        @benchmark gpu_multi_matmat_reg!($M_d, $K_d, $L_d)).time

end

CSV.write("cpu_data_vec.csv", DataFrame(times_cpu_vec), writeheader=false)
CSV.write("gpu_data_vec.csv", DataFrame(times_gpu_vec), writeheader=false)
CSV.write("cpu_data_mat.csv", DataFrame(times_cpu_mat), writeheader=false)
CSV.write("gpu_data_mat.csv", DataFrame(times_gpu_mat), writeheader=false)

plot_vector(times_cpu_vec,times_gpu_vec)
plot_matrix(times_cpu_mat,times_gpu_mat)
```

10.1.2 Planner Experiment

```
using CuArrays, CUDAnative, AbstractFFTs, FFTW, Test
using BenchmarkTools
using GPUArrays
using CuArrays.CURAND
using CuArrays.CUFFT
using Random
#using CSV, DataFrames
using Plots
using LinearAlgebra

function planner_test_cpu(size,runs)
    x = randn(ComplexF64,(size,size))
    y = zeros(size,size)
    p = plan_fft(x)

    for i=1:runs
        y = p * x
    end
    return nothing
end

function fft_test_cpu(size,runs)
    x = randn(ComplexF64, (size,size))
    y = zeros(size,size)

    for i=1:runs
        y = fft(x)
    end
    return nothing
end

function planner_test_gpu(size,runs)
    x_d = CuArrays.randn(size,size)
    y_d = CuArrays.fill(0.0f0, (size,size))
    p_d = plan_fft(x_d)

    for i=1:runs
        CuArrays.@sync y_d = p_d * x_d
    end
    return nothing
end

function fft_test_gpu(size,runs)
    x_d = CuArrays.randn(size,size)
    y_d = CuArrays.fill(0.0f0, (size,size))
```

```
    for i=1:runs
        CuArrays.@sync y_d = fft(x_d)
    end
    return nothing
end

function plotter_cpu(fft_data, planner_data, runs, size)
    p = plot(runs, fft_data, linecolor=:red,
        label=["Regular FFT"], legend=:topleft)
    plot!(p, runs, planner_data, linecolor=:blue,
        label=["Regular FFT","Planner FFT"], legend=:topleft)
    title!(p,"CPU Planner vs FFT for size $size")
    xlabel!(p,"runs")
    ylabel!(p,"time")
    savefig(p, "C:/Users/SID DrW/Documents/Github/MyPkg.jl/
        Experiments/Experiment 2/planner-$size-CPU.png")
    display(p)
    sleep(3)
end

function plotter_gpu(fft_data, planner_data, runs, size)
    p = plot(runs, fft_data, linecolor=:red,
        label=["Regular FFT"], legend=:topleft)
    plot!(p, runs, planner_data, linecolor=:blue,
        label=["Regular FFT","Planner FFT"], legend=:topleft)
    title!(p,"GPU Planner vs FFT for size $size")
    xlabel!(p,"runs")
    ylabel!(p,"time")
    savefig(p, "C:/Users/SID DrW/Documents/Github/MyPkg.jl/
        Experiments/Experiment 2/planner-$size-GPU.png")
    display(p)
    sleep(3)
end

#BenchmarkTools.DEFAULT_PARAMETERS.samples = 100
sizes = [16, 64, 256, 431, 512, 1024]
runs_cpu = [1,2,5,10,15,20,25,30,40,50]
runs_gpu = [1,2,3,4,5,6,7,8,9,10]

data_planner_cpu = zeros(length(runs_cpu), length(sizes))
data_fft_cpu = zeros(length(runs_cpu),length(sizes))

data_planner_gpu = zeros(length(runs_gpu), length(sizes))
data_fft_gpu = zeros(length(runs_gpu),length(sizes))

#CPU Computations
for i=1:length(sizes)
    for j=1:length(runs_cpu)
```

```
        evals = 5
        temp1 = zeros(evals)
        temp2 = zeros(evals)
        for k=1:evals
            temp1[k] = @elapsed fft_test_cpu(sizes[i],runs_cpu[j])
            temp2[k] = @elapsed planner_test_cpu(sizes[i],runs_cpu[j])
        end
        data_fft_cpu[j,i] = sum(temp1)/evals
        data_planner_cpu[j,i] = sum(temp2)/evals
    end
    plotter_cpu(data_fft_cpu[:,i], data_planner_cpu[:,i], runs_cpu, sizes[i])
end

GC.gc()
#GPU Computations
for i=1:length(sizes)
    for j=1:length(runs_gpu)
        evals = 2
        temp1 = zeros(evals)
        temp2 = zeros(evals)
        for k=1:evals
            temp1[k] = @elapsed fft_test_gpu(sizes[i],runs_gpu[j])
            temp2[k] = @elapsed planner_test_gpu(sizes[i],runs_gpu[j])
        end
        data_fft_gpu[j,i] = sum(temp1)/evals
        data_planner_gpu[j,i] = sum(temp2)/evals
        GC.gc()
    end
    plotter_gpu(data_fft_gpu[:,i], data_planner_gpu[:,i], runs_gpu, sizes[i])
end
```

10.1.3 Experiment 2, Part 1

```
using CuArrays, CUDAnative, AbstractFFTs, FFTW, Test
using BenchmarkTools
using GPUArrays
using CuArrays.CURAND
using CuArrays.CUFFT
using Random
using CSV, DataFrames
using Plots
using LinearAlgebra

include("plotting.jl")
BenchmarkTools.DEFAULT_PARAMETERS.samples = 50

function cpu_fft!(x, y, p)
    y = p * x
    return y
end

function gpu_fft!(x, y, p)
    CuArrays.@sync y = p * x
    return y
end

l = 10
A = zeros(l)
B = zeros(2*l)
for i=1:l
    A[i] = 2^i
    B[i] = 2^i
    B[i+l] = 2^(i+l)
end
#Data arrays for vectors
times_cpu1_vec=zeros(length(B))
times_cpu2_vec=zeros(length(B))
times_cpu4_vec=zeros(length(B))
times_cpu_vec=zeros(length(B))
times_gpu_vec=zeros(length(B))

#Data vectors for matrices
times_cpu1_mat=zeros(length(A))
times_cpu2_mat=zeros(length(A))
times_cpu4_mat=zeros(length(A))
times_cpu_mat=zeros(length(A))
times_gpu_mat=zeros(length(A))

for i=1:length(B)
```

```
    @show i
    N = Int(B[i])

    # CPU portion of the experiment
    x = randn(ComplexF64, (N,1))
    y = similar(x)

    FFTW.set_num_threads(1)
    p = plan_fft(x)
    times_cpu1_vec[i] = BenchmarkTools.mean(@benchmark cpu_fft!($x, $y, $p)).time

    FFTW.set_num_threads(2)
    p = plan_fft(x)
    times_cpu2_vec[i] = BenchmarkTools.mean(@benchmark cpu_fft!($x, $y, $p)).time

    FFTW.set_num_threads(4)
    p = plan_fft(x)
    times_cpu4_vec[i] = BenchmarkTools.mean(@benchmark cpu_fft!($x, $y, $p)).time

    times_cpu_vec[i] = min(times_cpu1_vec[i],
        times_cpu2_vec[i], times_cpu4_vec[i])

    # GPU portion of the experiment
    x_d = CuArrays.rand(N,1)
    y_d = CuArrays.fill(0.0f0, (N,1))
    FFTW.set_num_threads(1)
    p_d = plan_fft(x_d)

    times_gpu_vec[i] = BenchmarkTools.mean(
        @benchmark gpu_fft!($x_d, $y_d, $p_d)).time
end

for i=1:length(A)
    @show i
    N = Int(A[i])

    # CPU portion of the experiment
    x = randn(ComplexF64, (N,N))
    y = similar(x)

    FFTW.set_num_threads(1)
    p = plan_fft(x, 1)
    times_cpu1_mat[i] = BenchmarkTools.mean(@benchmark cpu_fft!($x, $y, $p)).time

    FFTW.set_num_threads(2)
    p = plan_fft(x, 1)
    times_cpu2_mat[i] = BenchmarkTools.mean(@benchmark cpu_fft!($x, $y, $p)).time
```

```
    FFTW.set_num_threads(4)
    p = plan_fft(x, 1)
    times_cpu4_mat[i] = BenchmarkTools.mean(@benchmark cpu_fft!($x, $y, $p)).time

    times_cpu_mat[i] = min(times_cpu1_vec[i],
        times_cpu2_vec[i], times_cpu4_vec[i])

    # GPU portion of the experiment
    x_d = CuArrays.rand(N,N)
    y_d = CuArrays.fill(0.0f0, (N,N))
    FFTW.set_num_threads(1)
    p_d = plan_fft(x_d, 1)

    times_gpu_mat[i] = BenchmarkTools.mean(
        @benchmark gpu_fft!($x_d, $y_d, $p_d)).time
    # if i > 7
    #     GC.gc()
    #     GC.gc()
    # end

end

plot_vector(times_cpu_vec, times_gpu_vec)
plot_matrix(times_cpu_mat, times_gpu_mat)
plot_matrix_reg(times_cpu_mat, times_gpu_mat)
plot_vector_cores(times_cpu1_vec, times_cpu2_vec, times_cpu4_vec)
plot_matrix_cores(times_cpu1_mat, times_cpu2_mat, times_cpu4_mat)
```

10.1.4 Experiment 2, Part 2

```
using CuArrays, CUDAnative, AbstractFFTs, FFTW, Test
using BenchmarkTools
using GPUArrays
using CuArrays.CURAND
using CuArrays.CUFFT
using Random
#using CSV, DataFrames
using Plots
using LinearAlgebra

include("plotting.jl")

# CPU computations
times_cpu_mat = zeros(12)
FFTW.set_num_threads(4)
BenchmarkTools.DEFAULT_PARAMETERS.samples = 50

x = randn(ComplexF64, (2,2))
y = similar(x)
p = plan_fft(x)
times_cpu_mat[1] = BenchmarkTools.mean(@benchmark y = p * x).time

x = randn(ComplexF64, (4,4))
y = similar(x)
p = plan_fft(x)
times_cpu_mat[2] = BenchmarkTools.mean(@benchmark y = p * x).time
@elapsed y = p * x

x = randn(ComplexF64, (8,8))
y = similar(x)
p = plan_fft(x)
times_cpu_mat[3] = BenchmarkTools.mean(@benchmark y = p * x).time

x = randn(ComplexF64, (16,16))
y = similar(x)
p = plan_fft(x)
times_cpu_mat[4] = BenchmarkTools.mean(@benchmark y = p * x).time

x = randn(ComplexF64, (32,32))
y = similar(x)
p = plan_fft(x)
times_cpu_mat[5] = BenchmarkTools.mean(@benchmark y = p * x).time

x = randn(ComplexF64, (64,64))
y = similar(x)
p = plan_fft(x)
```

```
times_cpu_mat[6] = BenchmarkTools.mean(@benchmark y = p * x).time

x = randn(ComplexF64, (128,128))
y = similar(x)
p = plan_fft(x)
times_cpu_mat[7] = BenchmarkTools.mean(@benchmark y = p * x).time

x = randn(ComplexF64, (256,256))
y = similar(x)
p = plan_fft(x)
times_cpu_mat[8] = BenchmarkTools.mean(@benchmark y = p * x).time

x = randn(ComplexF64, (512,512))
y = similar(x)
p = plan_fft(x)
times_cpu_mat[9] = BenchmarkTools.mean(@benchmark y = p * x).time

x = randn(ComplexF64, (1024,1024))
y = similar(x)
p = plan_fft(x)
times_cpu_mat[10] = BenchmarkTools.mean(@benchmark y = p * x).time

x = randn(ComplexF64, (2048,2048))
y = similar(x)
p = plan_fft(x)
times_cpu_mat[11] = BenchmarkTools.mean(@benchmark y = p * x).time

x = randn(ComplexF64, (4096,4096))
y = similar(x)
p = plan_fft(x)
times_cpu_mat[12] = BenchmarkTools.mean(@benchmark y = p * x).time

# x = randn(ComplexF64, (2^13, 2^13))
# y = similar(x)
# p = plan_fft(x)
# times_cpu_mat[13] = BenchmarkTools.mean(@benchmark y = p * x).time

# GPU computations
FFTW.set_num_threads(1)
times_gpu_mat = zeros(12)

x_d = CuArrays.randn(2,2)
y_d = CuArrays.fill(0.0f0, (2,2))
p_d = plan_fft(x_d)
times_gpu_mat[1] =
    BenchmarkTools.mean(@benchmark CuArrays.@sync y_d = p_d * x_d).time

x_d = CuArrays.randn(4,4)
```

```
y_d = CuArrays.fill(0.0f0, (4,4))
p_d = plan_fft(x_d)
times_gpu_mat[2] =
    BenchmarkTools.mean(@benchmark CuArrays.@sync y_d = p_d * x_d).time

x_d = CuArrays.randn(8,8)
y_d = CuArrays.fill(0.0f0, (8,8))
p_d = plan_fft(x_d)
times_gpu_mat[3] =
    BenchmarkTools.mean(@benchmark CuArrays.@sync y_d = p_d * x_d).time

x_d = CuArrays.randn(16,16)
y_d = CuArrays.fill(0.0f0, (16,16))
p_d = plan_fft(x_d)
times_gpu_mat[4] =
    BenchmarkTools.mean(@benchmark CuArrays.@sync y_d = p_d * x_d).time

x_d = CuArrays.randn(32,32)
y_d = CuArrays.fill(0.0f0, (32,32))
p_d = plan_fft(x_d)
times_gpu_mat[5] =
    BenchmarkTools.mean(@benchmark CuArrays.@sync y_d = p_d * x_d).time

x_d = CuArrays.randn(64,64)
y_d = CuArrays.fill(0.0f0, (64,64))
p_d = plan_fft(x_d)
times_gpu_mat[6] =
    BenchmarkTools.mean(@benchmark CuArrays.@sync y_d = p_d * x_d).time

x_d = CuArrays.randn(128,128)
y_d = CuArrays.fill(0.0f0, (128,128))
p_d = plan_fft(x_d)
times_gpu_mat[7] =
    BenchmarkTools.mean(@benchmark CuArrays.@sync y_d = p_d * x_d).time

x_d = CuArrays.randn(256,256)
y_d = CuArrays.fill(0.0f0, (256,256))
p_d = plan_fft(x_d)
times_gpu_mat[8] =
    BenchmarkTools.mean(@benchmark CuArrays.@sync y_d = p_d * x_d).time

x_d = CuArrays.randn(512,512)
y_d = CuArrays.fill(0.0f0, (512,512))
p_d = plan_fft(x_d)
times_gpu_mat[9] =
    BenchmarkTools.mean(@benchmark CuArrays.@sync y_d = p_d * x_d).time

GC.gc()
```

```
x_d = CuArrays.randn(1024,1024)
y_d = CuArrays.fill(0.0f0, (1024,1024))
p_d = plan_fft(x_d)
times_gpu_mat[10] =
    BenchmarkTools.mean(@benchmark CuArrays.@sync y_d = p_d * x_d).time

GC.gc()

x_d = CuArrays.randn(2048,2048)
y_d = CuArrays.fill(0.0f0, (2048,2048))
p_d = plan_fft(x_d)
times_gpu_mat[11] =
    BenchmarkTools.mean(@benchmark CuArrays.@sync y_d = p_d * x_d).time

GC.gc()

x_d = CuArrays.randn(4096,4096)
y_d = CuArrays.fill(0.0f0, (4096,4096))
p_d = plan_fft(x_d)
times_gpu_mat[12] =
    BenchmarkTools.mean(@benchmark CuArrays.@sync y_d = p_d * x_d).time

# GC.gc()

# x_d = CuArrays.randn(2^13,2^13)
# y_d = CuArrays.fill(0.0f0, (2^13,2^13))
# p_d = plan_fft(x_d)
# times_gpu_mat[13] =
    BenchmarkTools.mean(@benchmark CuArrays.@sync y_d = p_d * x_d).time

plot_matrix(times_cpu_mat, times_gpu_mat)
```

10.1.5 Experiment 2, Part 3

```
using CuArrays, CUDAnative, AbstractFFTs, FFTW, Test
using BenchmarkTools
using GPUArrays
using CuArrays.CURAND
using CuArrays.CUFFT
using Random
using Plots
using LinearAlgebra
using Statistics

device!(0)

include("plotting.jl")

function cpu_fft!(x, y, p)
    y = p * x
    return y
end

function gpu_fft!(x, y, p)
    CuArrays.@sync y = p * x
    return y
end

N = 11
it = 100

cpu_data = zeros(N)
gpu_data = zeros(N)
cpu_int_time = zeros(it)
gpu_int_time = zeros(it)

#device!(0)


for i=1:N
    @show i
    z = randn(ComplexF32, (2^i,2^i))
    y = similar(z)
    p = plan_fft(z)

    z_d = CuArray(z)
    y_d = CuArrays.fill(0.0f0, (2^i,2^i))
    p_d = plan_fft(z_d)

    x = randn(ComplexF32, (2^i,2^i))
```

```
    @time cpu_fft!(x, y, p)

    x_d = CuArray(x)
    @time gpu_fft!(x_d, y_d, p_d)

    for j=1:it
        x = randn(ComplexF32, (2^i,2^i))
        tc = @timed cpu_fft!(x, y, p)
        cpu_int_time[j] = tc[2]

        x_d = CuArray(x)
        tg = @timed gpu_fft!(x_d, y_d, p_d)
        gpu_int_time[j] = tg[2]
    end

    cpu_data[i] = mean(cpu_int_time)
    gpu_data[i] = mean(gpu_int_time)
end

plot_matrix(cpu_data, gpu_data)
plot_matrix_reg(cpu_data, gpu_data)
```

Bibliography

[1] M. Aboullaite, *Understanding JIT compiler*,
`https://aboullaite.me/understanding-jit-compiler-just-in-time-compiler/`

[2] M. Abramowits and I.A. Stegun, *Handbook of Mathematical Functions*,
Ninth edition, Dover Publications, December 1972

[3] A. Arif Ergin, B. Shanker and E. Michielssen, *The plane-wave time-domain algorithm for the fast analysis of transient wave phenomena*,
IEEE Antennas and Propagation Magazine Volume 41, Issue 4, September 1999

[4] M. Balducci, A.a Choudary, J. Hamaker,*Comparative Analysis Of FFT Algorithms In Sequential And Parallel Form*,
Mississippi State University

[5] J. Bezanson, S. Karpinski, V.B. Shah and A. Edelman, *Why We Created Julia*
`https://julialang.org/blog/2012/02/why-we-created-julia`, February 2012

[6] J. Bolewski, K. Kamieniecki, T. Besard, H. Ranocha, S. and V. Churavy, *JuliaGPU*
`https://github.com/JuliaGPU`

[7] J. Claerbout, *Imaging the Earth's Interior*
Stanford Exploration Project, 31 October 1997
Chapter on Slanted Waves

[8] R. Coifman, V. Rokhlin and S. Wandzura, *The Fast Multipole Method For Electromagnetic Scattering Calculations*,
Proceedings of IEEE ANtennas and Propagation Society International Symposium, July 1993

[9] J.W. Cooley, J.W. Tukey, *An algorithm for the machine calculation of comples Fourier series*
Mathematical Computation 19, page 297-301.

[10] K. Cools, *PWTD.jl package*,
`https://github.com/krcools/PWTD.jl`

[11] G.R. Cooper, C.D. McGillem, *Continuous and Discrete Signal and System Analysis*
Harcourt College Publishers, 1984, page 118

[12] P. Duhamel, H. Hollmann, *Split-radix FFT algorithm*
Electronic Letters 20, February 1984, page 14-16

[13] R. Fitzpatrick, *Quantum Mechanics*
University of Texas Reader, page 15

[14] G. Fourestey, M. Rexroth, C. Schäfer, J.-P. Kneib, *High Performance Computing for gravitational lens modeling: Single vs double precision on GPUs and CPUs.*
Astronomy and Computing, Volume 30, January 2020, 100340

[15] M. Frigo and S.G. Johnson, *The Design and Implementation of FFTW3*,
Proceedings of the IEEE 93(2), pages 216-231, February 2005

[16] M. Frigo and S.G. Johnson, *FFTW Documentation*
`fftw.org/fftw3.pdf`, version 3.3.8, 24 May 2018

[17] E. Heyman, R. Kastner, A. Shlivinski, *Antenna Characterization in the Time Domain*
IEEE Transatlantic Antennas Propagation, Vol. 45, Nr 7, 1997; pages 1140-1149

[18] D.G. Hough, M. Cowlishaw et al, *IEEE Standard for Floating-Point Arithmetic*
IEEE Std 754-2019, IEEE Computer Society

[19] J. van Kan, A. Segal, F. Vermolen, *Numerical Methods In Scientific Computing*
Second Edition, page 5, 6

[20] J. van Kan, A. Segal, F. Vermolen, *Numerical Methods In Scientific Computing*
Second Edition, page 131

[21] H. Karner, C. Ueberhuber, *Parallel FFT algorithms with reduced communication overhead,*
Institute for Applied and Numerical Mathematics, Technical University of Vienna

[22] S. Meiyappan,
Implementation And Performance Evaluation Of Parallel FFT Algorithms,
National University Of Singapore

[23] J. Revels, *BenchmarkTools.jl Documentation*,
`https://github.com/JuliaCI/BenchmarkTools.jl/blob/master/doc/manual.md`

[24] J. Revels, *BenchmarkTools.jl package*,
`https://github.com/JuliaCI/BenchmarkTools.jl`

[25] S.W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*
Chapter 13: Convolution

[26] C. Vuik and C.W.J. Lemmens, *Programming on the GPU with CUDA*,
Reader for a course on CUDA programming on the GPU, January 2019

[27] C. Vuik, F.J. Vermolen, M.B. van Gijzen, M.J. Vuik, *Numerical Methods for Ordinary Differential Equations*,
Delft Academic Press, Second Edition, page 52

[28] C. Vuik, F.J. Vermolen, M.B. van Gijzen, M.J. Vuik, *Numerical Methods for Ordinary Differential Equations*,
Delft Academic Press, Second Edition, page 54, 56

[29] *AbstractFFTs package*,
https://github.com/JuliaMath/AbstractFFTs.jl

[30] *Capgemini Website*,
https://www.capgemini.com/

[31] *clFFT*,
https://clmathlibraries.github.io/clFFT/

[32] *CuArrays.jl Package*
https://github.com/JuliaGPU/CuArrays.jl

[33] *CUDA GPUs*,
https://developer.nvidia.com/cuda-gpus

[34] *Digital Library of Mathematical Functions: Bessel- and Spherical Bessel Functions*,
https://dlmf.nist.gov/10.47

[35] *FFTW package*,
https://github.com/JuliaMath/FFTW.jl

[36] *Intel Math Kernel Library*,
texttthttps://software.intel.com/en-us/mkl

[37] *Julia Documentation On Packages*,
https://docs.julialang.org/en/v1.0/stdlib/Pkg/

[38] *Julia Documentation On Types*,
https://docs.julialang.org/en/v1/manual/types/index.html

[39] *Julia PkgTemplate Repository*,
https://github.com/invenia/PkgTemplates.jl

[40] *JuliaCon 2019 — Keynote: Professor Steven G. Johnson*,
https://www.youtube.com/watch?v=mSgXWpvQEHE, mark at 44:11

[41] *Khronos Group About page*,
https://www.khronos.org/about/

[42] *Nvidia cuFFT*,
https://developer.nvidia.com/cufft

[43] *OpenCL Overview*,
https://www.khronos.org/opencl/

[44] Techpowerup, *Nvidia Quadro M 1200*,
https://www.techpowerup.com/gpu-specs/quadro-m1200-mobile.c2921

[45] Techpowerup, *Nvidia Geforce RTX 2080 Super*,
https://www.techpowerup.com/gpu-specs/geforce-rtx-2080-super.c3439

[46] Techpowerup, *Nvidia Tesla V100*,
https://www.techpowerup.com/gpu-specs/tesla-v100-pcie-32-gb.c3184