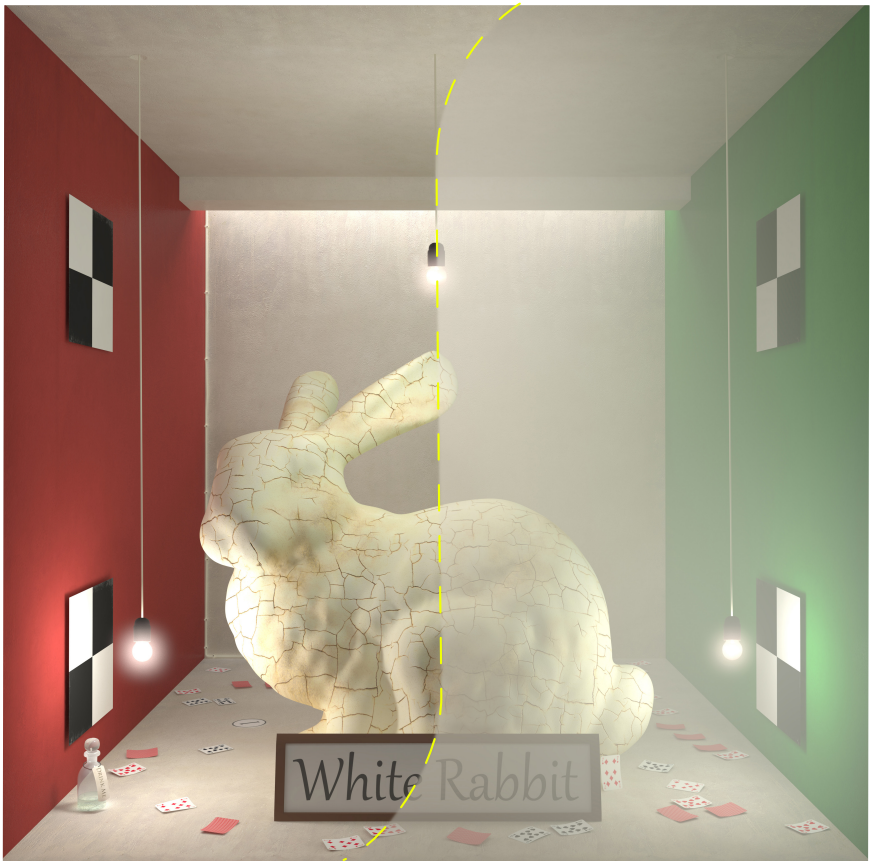
UNIVERSITY OF GOTHENBURG



Study of Convolution Algorithms using CPU and Graphics Hardware

Master of Science Thesis in the Programme Computer Science

Matz Johansson Bergström

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, Sept 2012

Study of Convolution Algorithms using CPU and Graphics Hardware

Matz Johansson Bergström, matz.johansson@gmail.com

Examiner: Ulf Assarsson

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone +46(0)31-772 1000

Cover: Render of the Authors interpretation of the "Cornell box". The scene was modeled and textured by the author using 3D Studio Max. The Stanford Bunny was downloaded from the Stanford Computer Graphics Laboratory. The rightmost half of the render was filtered using VISSLA™.

Department of Computer Science and Engineering
Göteborg, Sweden September 2012

Dedicated to Linda,
you will be missed forever...

Sammanfattning

In this thesis we evaluate different two-dimensional image convolution algorithms using Fast Fourier Transform (FFT) libraries on the CPU and on the graphics hardware, using Compute Unified Device Architecture (CUDA).

The final product is used in VISSLA (VISualisation tool for Simulation of Light scattering and Aberrations), a software written in Matlab. VISSLA™ is used to visualise the effects of cataracts, therefore it is important for our proposed method to be called from within Matlab. The product makes it possible to call graphics hardware from Matlab using the Mex interface.

In this thesis we also explore the optimal usage of memory, and attempt to control allocated memory in a predictable way, to be able to minimise memory-related errors. A novel (hybrid) GPU/CPU algorithm using gpuArray and the *row-column* method is also presented and examined.

Our proposed method speeds up the current computation on VISSLA™ by 3-4 times. Additional proposed optimisations are examined along with the estimated resulting speedup.

Innehåll

List of Algorithms

Listings

Figurer

Tabeller

Acknowledgements

I wish to thank *Björn Löfving* at the department of Ophthalmology at Mölndals Sjukhus for introducing me to FFT, giving me feedback on my work and meticulously proofreading my section on FFT.

Jörgen Thaung, also at Ophthalmology dept. for giving feedback on the introductory section of the report.

Ulf Assarsson, for proofreading the whole report and giving feedback.

Thomas Ericsson, for proofreading the section on previous work and Matlab, and giving general feedback.

Thanks also goes to *Marcus Billeter* for trying CUDA on Unix from Matlab, giving me something to start from.

Per Strömdal for following my work online in the middle of the night via chat.

My father, *Leif Johansson*, for moral support.

Zoran Popović and *Alf Nyström* for interesting discussions over lunch about everything but the thesis.

Also, thanks to *Gunilla Magnusson* for being kind enough to help with a job application and touching up my CV.

SECTION 1

Introduction

In this thesis we study methods of performing large-kernel image convolution using different FFT libraries and algorithms from Matlab.

First, we introduce the definition of convolution and how to speed it up using FFT. To gain further insight, we will briefly discuss the algorithmic details of FFT and give an alternative algorithm in the section *Mathematical background*.

In the section *Implementation*, the main tools used in this thesis is introduced. We describe how to efficiently program in Matlab and how to call CUDA code from Matlab via the MEX interface.

In *Results* we investigate the performance of supposedly fast FFT libraries, and compare them to our own GPU implementation in CUDA (see Section 5). Our results will focus on implementing a fast and memory-efficient convolution algorithm using pre-written FFT routines. We will examine ways to speed each implementation up and to give an accurate and fair comparison of them. We will also assess what limitations each solution has and how to fix them.

We assume the reader has programming experience with C and Matlab. In the thesis we have mainly worked on Windows as a platform, but Unix could be used as well. In the Appendix, we provide the reader with a brief introduction to installing and compiling CUDA under Windows. In Unix, compiling CUDA code is much simpler. Additionally, It is recommended to have completed a course in basic linear algebra to appreciate the algorithms used.

1.1 MOTIVATION

The computational speed of GPUs has exploded in recent years. Research has even indicated that the GPU is two order of magnitudes faster than the CPU for many applications. However, such performance figures have also been shown as unfairly biased [VWLN10]. It is important to compare similar CPU and GPU setups, as well as utilizing all possible optimisations on both the GPU and the CPU [GH11].

We want to investigate how much faster a mid-range GPU is to a mid-range CPU in a real world setting, and what we can do to speed up code. Details on the computer system used to compare the performances can be seen in Section 5.

1.2 PURPOSE

The main purpose of using convolution algorithms in this work is to visualise limitation in human vision during normal ageing and disease. The software and the convolution implementation is used as a tool to assess contrast and detail loss due to optical errors of the eye.

One of the main applications of the software is to work as an aid in the construction of and in the design process of work environments and public spaces. Additional applications include traffic safety where good visual performance is very important.

1.3 BACKGROUND

The eye is a complicated optical organ responsible for receiving photons and converting them to electro-chemical impulses that can be interpreted in the brain as images.

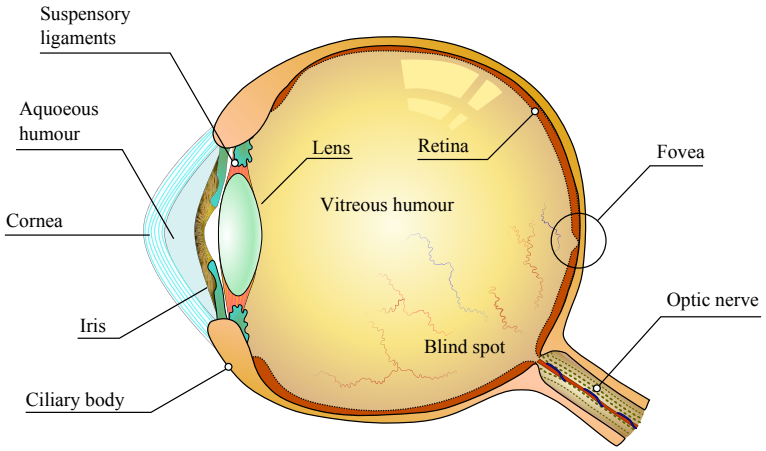


Figure 1.1: Illustration of a cross section of the eye. Light is passed through the lens and focuses light on to the retina.

The retina (see Figure 1.1) is a light-sensitive surface, covering the rear of the eye. The retina [Smi97, chap. 23] consists mainly of three layers of nerve cells responsible for

1. Transferring information via the optical nerve, to the brain
2. Image processing and data reduction
3. Converting photons into neural signals via rod and cone receptors

The *optical nerve* enters the retina at a point called the *blind spot*. It is so called because this area is missing both cones and rods, therefore it is unable to send any information to the brain. At the fovea, however, there is a very high density of rods, which enables a detailed sharp central vision.

The visual quality could also be enhanced in the brain, as a post-processing step. However, the quality of vision is ultimately limited by the optical quality of the eye. The quality of vision is also affected by optical errors, but some of these errors can be corrected without surgically modifying the lens. These errors includes *myopia* (shortsightedness) and *hyperopia* (farsightedness) both could be corrected with ordinary prescribed eye glasses. Another optical error is astigmatism. All but severe cases of astigmatism can be corrected optically.

One of the more common optical errors is light scattering. Normally 5-10% photons are scattered by entering the cornea and the lens before it reaches the retina. The effect is increasing with age. In the case of cataract, the scattering is in the range 50-100%. Cataract is a type of clouding that develops in the lens of the eye, and this clouding introduce glare.

As an example of the prevalence of cataracts in America, it has been reported in 2002 [Cat] that cataracts affect about 20 million Americans of age 40 and older. By the age 80, more than half of all Americans have cataracts.

Cataracts scatters the light, resulting in poor vision with prominent glaring. As an example, Figure 1.2 shows a photo of a lift at Mölndals sjukhus. The light source is unshielded and “Hiss C” is clearly shown. In the lower part of the figure, we see how the photons are focused and scattered

on the retina of a cataract-affected eye (2a). The rightmost photo (2b) shows the simulated cataract which barely makes the text visible.

The illness is progressive, and it is difficult to self diagnose and is possibly also therefore left untreated. The effect of cataracts is seen as a so called a *veiling luminance*. The effect is evident if strong lights without glare shielding are in the direct field of vision. The scattered light affects the vision, even if the light source is peripheral.

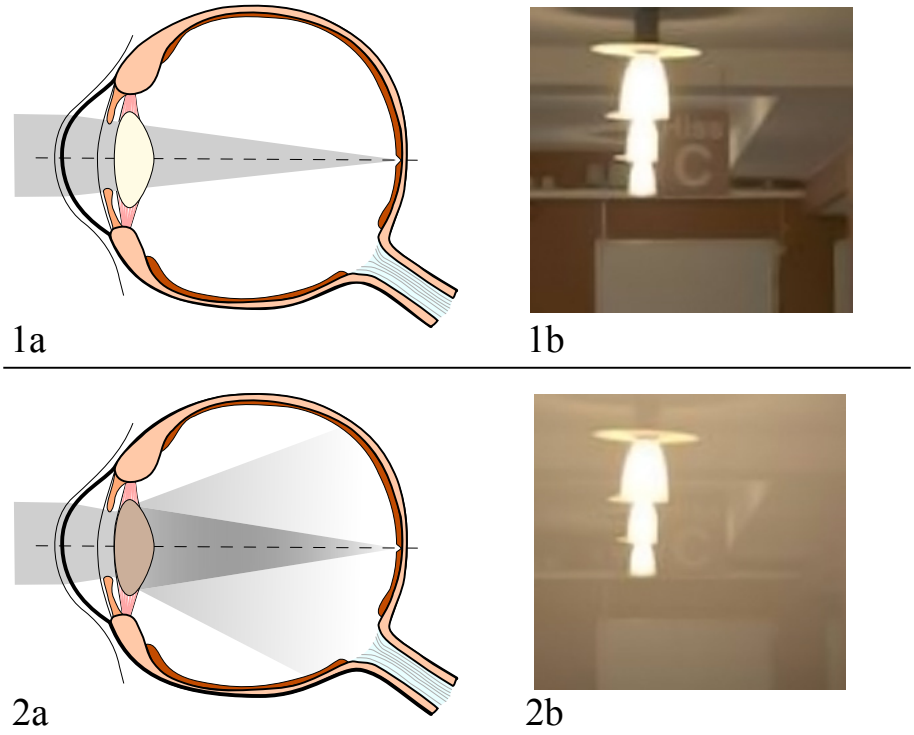


Figure 1.2: Showing the illustration of an unaffected eye and a cataract-ridden eye. The rightmost photo **2b** shows a simulation using VISSLA™ software.

1.4 PROBLEM STATEMENT

The goal for this research is to find a way to utilise the graphics hardware and to speed up (linear) convolution from Matlab. Some of the issues we will resolve are

1. How much faster can convolution be performed on the GPU compared to the CPU?
2. Where is the bottleneck, and how can we speed up the code?
3. Is our research going to be relevant for the next five years?

The last question will be answered as a part of our conclusion section, see Section 6.

1.5 APPLICATION

As previously mentioned, cataract affects vision, especially if strong unshielded light sources are present in the field of view. By using photos, we can simulate the scattering of light, thus also simulate the effect of cataracts. The problem is, we need a very high luminance resolution to get a true representation of the details of objects in dark, as well as in bright areas.

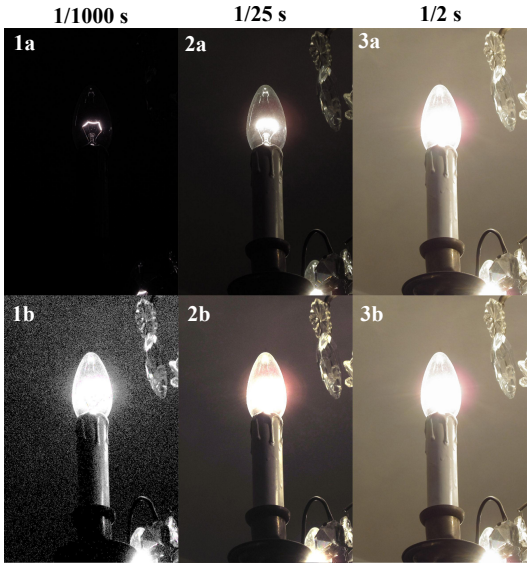


Figure 1.3: Arrangement of photos of an unshielded chandelier light bulb showing the lack of detail of a 8 bit JPEG.

On a side note, recent research by Vitor F. Pamplona and Raskar in [VFPR12] has found a way to increase the readability of digital displays for people with visual aberrations such as far sightedness and even cataract. Their approach is to use a 3D-display, tailored to compensate for the users vision using a grid of *warped light fields* to create the image in-focus. The result is a perceived sharper image. Their research also shows that even people affected with cataracts will perceive a sharper image using the display.

Figure 1.4: The second column in the image arrangement shows the filtered images. The upper row depicts a photo and the lower row shows the render.

The output from a non-professional digital camera is mostly in the JPEG file format. JPEG is a 24 bit format, 8 bit for each color channel (2^8 = 256 intensities). This is not nearly enough intensities to get the dynamic range needed to simulate cataract. For high-end cameras we have the RAW image format, which stores 12-16 bits per channel, (4096-65536 intensities). Even the intensity resolution given by the RAW format is still not enough. There is a format called HDR, which can contain between 16 to 32 bit. For our purposes we need at least 24 bit.

A way to create the HDR photos is to use several exposures as seen in Figure 1.3 and merge them together and store them in the HDR format. As seen in the figure, if we change the exposure of the short-exposure image 1 to the long-exposure image 3, we can clearly see the lack of detail and noise in image (1b). The exposures are indicated in the upper row lower corner of each image.

Another way to visualise the effects of cataract is to use photo realistic rendering software, such as Relux [Rel] or V-Ray rendering plugin [Vra]. These could also be used to create the HDR content.

As an example of practical use of rendering systems to convolve renders is the work of Johansson and Samuelsson [JS12]. The two students created a 3D model and render as a part of a thesis project from Jönkoping University. The resulting renders can be compared to the real images as seen in Figure 1.4. Image **1a** shows a real photo taken in Frölunda, Gothenburg, Sweden. **2a** shows a render created at Vectura with Relux. The images **1b** and **2b** has been convolved by VISSLA™

The work they did was part of an internship at Vectura [Vec] and the image was a result of a render using the Relux renderer. The reference photo was captured by Björn Löfving.

One problem is that the image must be very large to contain peripheral light sources, which is important since they contribute to the image. Another reason for large images is the possibility of studying contrast in the image in detail. Since the HDR images are linearised and absolute calibrated, we are able to measure directly on the image. Additionally, a tone mapping is applied to the image to make it possible to view on a regular 8 bit monitor. The software is used as a tool to study contrast levels and experiment with different light shielding scenarios. The experimental nature of how the software is used makes it therefore of utmost importance to make the visualization computation as fast as possible.

In the next section we will give the necessary background to the development of the convolution algorithms.

1.6 CONVOLUTION

Image processing programs, such as Photoshop, has several filters available to use. Filters, such as "Gaußian blur" could be implemented using a *sliding window* to blur an image. This window is usually referred to as a *kernel*. In this example, an appropriate kernel is defined over two dimensions, using a discretised Gaußian function.

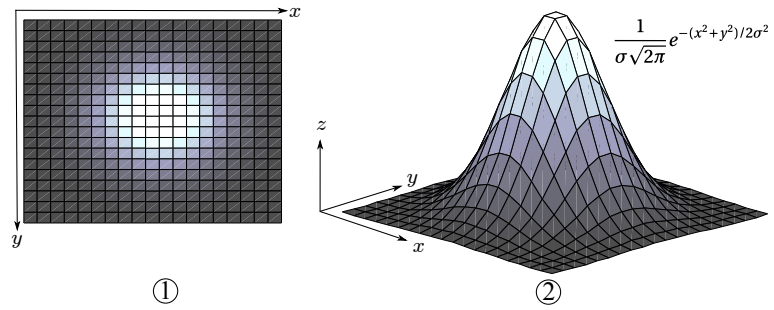


Figure 1.5: (1) Shows the Gaußian kernel as a matrix with intensities. (2) Depicts the shape of the Gaußian kernel in 3 dimensions.

The way we visualise the vision of an eye affected with cataract is similar to the Gaußian blur filter. One important difference is that we need to use a *sliding window* that is as large as the input image. In this way, each pixel will affect all other pixels.

Convolution can be implemented in many ways, either using the definition directly (called "Brute Force") that is a very simple algorithm to implement, but also inherently slow. Convolutions can also be calculated using fast Fourier transform, which is very efficient, but much more complicated to implement. We will see the definition of convolution later. We will also introduce the fast Fourier transform.

1.7 GPU VERSUS CPU

The CPU (central processing unit), is the brain of a computer. The CPU is designed to execute instructions in sequence. It is common nowadays that a computer has 1-8 processors, running instructions in parallel. Even with only one processor, the computer may seem to run tasks in parallel. In fact, the CPU still runs everything sequentially, but switches between tasks fast enough to give the appearance of making programs run in parallel.

The GPU, on the other hand, usually consists of about 500-1000 cores, or "processors". The GPU was specifically designed to execute tasks in parallel, so therefore it consist of many more transistors that are able to compute in parallel, as opposed to the CPU which must maintain a stable OS, cache data and allow user flow of control [Nvi]. In Figure 1.6 we can see the difference in number of Arithmetic Logic Unit (ALU) compared to the CPU.

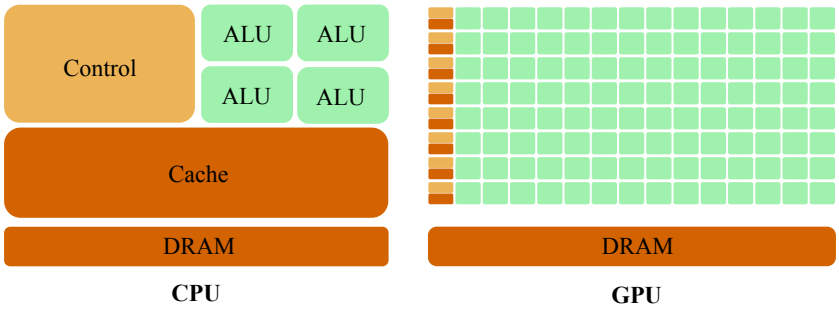


Figure 1.6: CPU compared to GPU. Note the number of ALUs of the GPU compared to the CPU. The figure is based on the *n*Vidia Programming Guide[Nvi]

However, the GPU is limited in both memory and speed, but the ability to execute tasks in parallel makes it faster than the CPU for certain tasks. In 2007, *n*Vidia created CUDA, which is a Application Programming Interface (API) for *n*Vidia GPUs. CUDA makes it possible to execute your own GPU code, taking advantage of its multi-core architecture.

In order to make multi-core programming simpler, CUDA introduces a *thread hierarchy*, to handle the way cores are used. A thread is, put simply, a process and the hierarchy contains threads, grids and blocks. These can be used to, for instance, assign a thread to several elements in a matrix. In this way, we do not need to care about explicit indexing, we simply use the thread hierarchy to map threads to elements in the matrix. To read more about this, see [Nvi, chap. 2.2].

A function that runs on the GPU is called a *kernel*. This should *not* be confused with the term *convolution kernel*. Additionally, the GPU is called the *device*, and the CPU is called the *host*. We will refer to these throughout the report.

In the next section we will introduce previous work made in the research fields of FFT, convolution, on the CPU and the GPU.

SECTION 2

Previous Work

Convolution as a concept has been used in many different areas of research. It can be applied in optics, telecommunications, computer graphics, acoustics, electrical engineering and radio astronomy to name a few. As such, convolution has found itself under many different names, so an exact historical rundown of the use of convolution, before 1940, is difficult. A few examples includes, but are not limited to, Faltung (German word used today), composition product, superposition integral, Carson's integral et cetera [III84].

2.1 FFT

The fast Fourier transform is a mathematical concept that is tightly linked to convolution. In 1942, Danielson and Lanczos published a paper [DL42] on Fourier Analysis which lay the groundwork for the fast Fourier transform. As early as 1805, Gauß invented a interpolation technique resembling the Discrete Fourier Transform, two years before the work of Joseph Fourier [Fou07]. This discovery has been described in [MTHB85]. In 1965, Cooley and Tukey published a paper on "machine calculation of complex Fourier series" [CT65] which revitalized the interest of many researchers. Cooley and Tukey showed how to implement the algorithm using a computer, which sparked the interest of fellow researchers. Modifications of the original algorithm was proposed in the following years. Many of them provided efficient FFT of different input sizes. Some approaches use other methods altogether. Bruun proposed a recursive polynomial approach using *Z transforms* to compute FFT [Bru78]. For more information about Z-transforms, see [Smi97, chap. 33].

Henrik V. Sorensen and Burrus showed in [HVSB87] an FFT algorithm that takes real (power of two) inputs as opposed to complex. The result was a faster and cleaner code, due to the special symmetry and also because of less data shuffling in the code.

For more information, see [Tt], who gives a comparison between a couple of real valued-FFT algorithms. We will discuss the core idea of this speedup, which utilises a special symmetry (*Hermitian*) when transforming real data, see Section 5.2.1.

In 1997, Frigo at MIT developed the FFT library FFTW [Fri99]. The library is open source and contains many of the previously proposed FFT algorithms. The library works by estimating which algorithms are optimal for a specific input, this stage is called *planning* and the resulting output is called a *plan*. FFTW relies on several algorithms to decompose its input and transforming them. In 2007 FFTW released a major update, updating to version 3 (FFTW3). They added in that update, SIMD instructions [FJ05] to the entire library, amongst other modifications. FFTW is today the standard FFT library for programming environments, and is used in Matlab, Scilab [Sci] and Octave [Oct]. Many other FFT libraries follow the same plan approach such as Spiral FFT [Spi], Nukada FFT [Nuk06] and CUDA FFT (CUFFT) [NVI12], but their motives and algorithms differ.

In 1997, Daniel Bernstein created his own FFT routine named DJBFFT and claimed it was faster than FFTW, initially only on Pentium machines. FFTW provides a web page with their own benchmark results [Fftc] comparing known FFT routines with their FFTW. Bernstein raises questions on how these benchmarks are conducted. According to Bernstein, the FFTW authors did not compile his routine using the correct compiler options [Ber].

Another point made by Bernstein is that the FFTW benchmarks only provide the timings on the forward FFT and not on the forward *and* inverse transforms. He also adds that many libraries differ in the way they handle scaling of the transform, some normalise in the forward direction and some in the backward, yielding a faster routine.

Bernstein's DJBFFT was last updated in late 1999 and was, according to Bernstein, developed to prove that creating a faster code than FFTW is possible. Since DJBFFT version 0.70, a convolution routine was also available, but both the FFT and convolution routines are today deemed obsolete.

Another example of FFT library is named *Spiral FFT*. As we will show in Section 5.2.6, Spiral [Spi] FFT is actually *slightly* faster than FFTW, but the FFTW benchmarks (one-dimensional single precision, real data) does unfortunately not include Spiral FFT.

Additionally, in 2003, Intel released their commercial Math Kernel Library [Mkl], containing their own tuned FFT library. According to benchmarks [Ffta], their one-dimensional transforms are only slightly faster than FFTW. For some two-dimensional transforms the library is no faster than FFTW.

In the next section we will explore the history of linear convolution.

2.2 (LINEAR) CONVOLUTION

Worth mentioning is that both convolution and its inverse (deconvolution) has successfully been applied in a wide variety of applications. A special convolution algorithm designed to remove noise from radio signals was developed 1974 by Jan Högbom [Hog74]. The algorithm is part of the CLEAN software library.

Deconvolution or deblurring is another technique where FFT is used. An example of this can be seen in Figure 2.1. The photo (1) was taken while the camera was in motion, producing a blurry photo in the direction of the motion. The photo was then deblurred by PictureSolve (see [Leh]) using special software which in turn uses FFT, clearing up the blur, making the license plate readable.

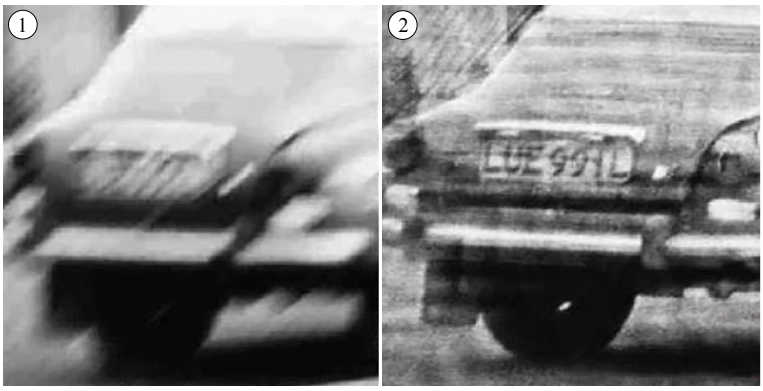


Figure 2.1: The image (1) shows a blurred photo of a licence plate. (2) shows the same image processed by PictureSolve using FFT. Used with permission from PictureSolve.

Convolution, as mentioned before, could be implemented fast using FFT. In 2001, a paper on cyclical convolution using specialised transforms [LZ01] provided an algorithm with a speedup of 40% to 65% compared to convolution using FFT. Unfortunately, we were unable to find any implementation of the algorithm.

To read more about specialised fast convolution algorithms, the reader is kindly referred to [SB99, chap. 8].

Interesting to note is that Werman shows in [Wer03] (2003) that a convolution under special circumstances can be faster than convolution using FFT. The paper gives the condition that if a kernel $\mathbf{K}$ is a linear combination of polynomials, exponentials and trigonometric terms, convolving $\mathbf{K}$ is reduced to only depend on the degree of the polynomials, and not on the dimension of the kernel, as with FFT. The paper further claims that the algorithm is faster than performing FFT on a $N \times N$ matrix for polynomials with degree d smaller than $\log N$. Assuming our input data N is ≈ 8000 we get $\log_2 8000 \approx 13$. Additionally, the extra space required is only $\mathcal{O}(d)$.

2.3 DEVELOPMENT ON THE GPU

Using GPUs to perform FFT computation was first introduced by Moreland and Angel in [MA03] (2003). In 2005, FFT on GPU was used for medical reconstruction and compared their GPU implementation to FFTW on the CPU [PF05]. For most of the tests they saw a speedup of 1.7-2 times. For our purposes the code is outdated.

In November 2006, *n*Vidia introduced a new GPU programming language called CUDA [Cudb], and with their toolkit an implementation of FFT named CUFFT. In late 2008, Open Computing Language (OpenCL) [Ope] was released as a result of collaboration between AMD, IBM, Intel and Nvidia. Nvidia provides support to OpenCL as a part of their GPU computing toolkit.

In a publication from 2010 by Kamran Karimi and Hamze, a comparison between OpenCL and CUDA suggested that the performance of their test kernel code written in OpenCL was 13%-63% slower than CUDA [KKH10]. OpenCL, as opposed to CUDA, is developed to work for many different platforms. Our perception is therefore that CUDA is more matured and, since the target hardware is *n*Vidia only, has better support and a more optimized compiler.

In 2010, Al Umairy et al. published a paper on *small two-dimensional convolutions* using CUDA [AU+10].

Since the first version, CUFFT has been improved several times. Volkov [Vol] provided a speedup for a specific one-dimensional input size. The code has since been part of CUFFT.

SECTION 3

Mathematical background

We define the convolution function *, of two real one-dimensional functions, f and g, according to [Ns06] as

$$(f * g)(t) = \int_{\mathbb{R}} f(\tau)\, g(t-\tau)\, \mathrm{d}\tau. \tag{1}$$

This is the standard definition of convolving two functions defined over a continuous time variable (t).

In this thesis we work on two-dimensional images stored in computer memory in finite accuracy (i.e., floating precision). Therefore we use the discrete two-dimensional version of the above equation. We also introduce terms for $\mathbf{I}$ the input (image) and $\mathbf{K}$ the kernel. Also note that we use spatial coordinates as opposed to Equation 1 which is defined over time. These restrictions are important to establish. They also affect our choice of algorithms later (see Section 7).

A convolution of two matrices, $\mathbf{I}$ and $\mathbf{K}$, defines each coordinate (u, v) of the convolved matrix as

$$(\mathbf{I} * \mathbf{K})(x, y) = \sum_{u=1}^{N_x} \sum_{v=1}^{N_y} \mathbf{I}[x-u, y-v]\, \mathbf{K}[u, v] \tag{2}$$

This function (see also [Smi97, chap. 24]) is thus $\mathcal{O}(N_x N_y)$. The algorithms implementing this equation goes by the name of "brute force", because they are implementing the strict definition, without any optimisations.

For small kernels, we can directly use the brute force Equation 2, to perform convolution. This algorithm (run in parallel) is actually feasible, as seen in [Smi11]. Yet another optimization, which leads to a faster convolution, is if the kernel is *separable*. A kernel is separable if it can be expressed as the outer product of two vectors. This optimization is in the order of 5 times faster than convolving non–separable kernels [MSS00; Pod07]. For this project, we must use *non-separable* large kernels. We will in the next sections describe the general mathematics behind the proposed convolution algorithm. First, we will briefly describe the theory behind the fast Fourier transforms.

3.1 USING THE FAST FOURIER TRANSFORM

In the example above, we presented the theory of convolving a two-dimensional image, sampled at discrete intervals. Assume that we sample a periodic one-dimensional function and store the values in a vector x. The Discrete Fourier Transform (DFT) is defined as a *complex invertible linear transform*. Given the vector x, we can transform it into another vector X with

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i 2\pi k \frac{n}{N}} \Bigg|_{k \in [0, N-1].} \tag{3}$$

The transform decomposes the sampled function into cosine and sine waves. We will digress for a moment, to gain an intuition about the theory, and see an alternative way to look at Equation 3. We know from *Euler's formula* that

$$e^{i\theta} = \cos\theta + i\sin\theta. \tag{4}$$

This is only a compact way of describing points on the complex unit circle using the angle θ. We then see that Equation 3 is a rotation of $\theta \in [0, 2\pi N]$ degrees over the vector x, we use a rotation matrix and we have a new point

$$\begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} \mathrm{Re}[x_n] \\ \mathrm{Im}[x_n] \end{pmatrix} \tag{5}$$

We use this idea to express our first version of a one dimensional DFT, written in Matlab code

Listing 1: DFT

```
RX = 0*x; %real part
IX = 0*x; %imaginary part
N = length(x);

for k=1:N %we perform DFT for the whole of X
    for n=1:N

        theta = 2*pi*(k-1)*(n-1)/N;

        RX(k) = RX(k) + real(x(n)) * cos(theta) + imag(x(n)) * sin(theta);
        IX(k) = IX(k) - real(x(n)) * sin(theta) + imag(x(n)) * cos(theta);

    end
end

X = RX + 1i*IX; %answer in complex format
```

Note: We could have used polar coordinates to describe the same points rotated around the unit circle.

The one-dimensional DFT is easily extendable to work for two-dimensional images where each row is an input vector (see Section 3.2). For more information about two-dimensional DFT the reader is kindly referred to [GW06]. In two dimensions, the algorithm is as slow as the brute force definition in Equation 2.

What makes Fourier transforms feasible in performing convolution is the *convolution theorem*. This theorem link Fourier transforms to convolution in a clever way. In the next section we will see how it is possible to speed up convolution by introducing the fast Fourier transform.

3.1.1 FFT

As we have seen, the fast Fourier transform is an algorithm implementing the Discrete Fourier transform. The FFT is an applied algorithm used in varied branches of science such as optics and audio processing. A recent example of the use of FFT is *Zenph Inc.* [NK06] which are specialised in transforming old piano performances and re-performing them. To do this, they use

FFT to transform audio data into frequencies as a part of their patented software. The reader is recommended to read more about the theory of FFT in [Str03] and [Smi97].

We recall that DFT could be, simply put, interpreted as a summation with special *rotational* weights. This operation effectively transforms a function from the *spatial domain* into a so called *Fourier domain*, which expresses the same data using frequencies instead of spatial data. What makes fast convolution using FFT possible is the *convolution theorem* [Ns06]

$$f * g = \mathfrak{F}^{-1}(\mathfrak{F}(f) \cdot \mathfrak{F}(g)). \tag{6}$$

This theorem states that a convolution in the *spatial domain* is the equivalent to performing a multiplication in *Fourier space*. We only need to take the inverse Fourier transform of the result to transform the function back to the spatial domain.

To appreciate the difference between DFT and FFT we will show the recursive pseudo code in Algorithm 3.1 [Hea02, chap. 12].

In the heart of the FFT algorithm is its symmetries and redundancies that makes the algorithm very efficient. The algorithm is very simple to express recursively, which we will see below.

For simplicity we assume the input x to be 2^m, where $m \in \mathbb{N}$. The output is stored in y and we will use $\omega = e^{-2\pi}$.

Algorithm 3.1 Recursive (Depth first) one-dimensional FFT

function FFT(x, y, N, ω)
  if N==1 **then** ▷ Base case
    $y[0] = x[0]$
  else
    for $k = 0$ to $N/2 - 1$ **do** ▷ Split into even and odd sub-sequences
      $p(k) = x[2k]$
      $s(k) = x[2k + 1]$
    end for

    FFT($p, q, N/2, \omega^2$) ▷ Recursive call
    FFT($s, t, N/2, \omega^2$)

    for $k = 0$ to $N - 1$ **do**
      $y[k] = q[k \mod n/2] + \omega^k t[k \mod n/2]$ ▷ Combine results
    end for

  end if

end function

We compute $\mathcal{O}(\log N)$ levels of recursion, for each of those levels we compute $\mathcal{O}(N)$ arithmetic operations, hence our time complexity is $\mathcal{O}(N \log N)$.

Most modern FFT algorithms use a bit reversal scheme instead of explicit recursion. However, contrary to popular belief, recursive FFT is not inefficient. Xiangyang Liu and Wang did in [XLW09] compare between recursive and iterative FFT. They found that recursive FFT is not necessarily slower than a iterative version. For more information about implementing your own FFT, the reader is referred to the excellent online material written by Johnson and Frigo [JF].

3.1.2 Convolution using FFT

As previously stated (in Section 1), if we convolve two $N \times N$ images $\mathbf{I}$ and $\mathbf{K}$, we are performing what is also called a *linear convolution*. The previously mentioned convolution theorem, makes performing a linear convolution fast, by using FFT.

We will, for clarity, depart from the notation $\mathfrak{F}$, and from here on use FFT_1 and FFT_2 to denote one and two-dimensional fast Fourier transforms respectively. With this new notation, we express a convolution of the matrices $\mathbf{I}$ and $\mathbf{K}$ using the convolution theorem

$$\mathbf{I} * \mathbf{K} = \text{IFFT}_2(\text{FFT}_2(\mathbf{I}) \circ \text{FFT}_2(\mathbf{K}))$$

where $\circ$ is the Hadamard (element-wise) matrix product. This equation also work for 1-dimensional transforms, as we will see in Section 3.2.

By only using the convolution theorem, without any modification to the input data, we are performing a *cyclical convolution*. The cyclical convolution is not what we want, because, per definition, a cyclical convolution wraps the data around, introducing artifacts.

Given two N-vectors $\mathbf{I}$ and $\mathbf{K}$, to get a linear convolution, we need to pad the vectors with zeros [WHP07, chap. 13.1]. The reason for adding zeros is because DFT, and subsequently FFT, assumes the vector is *infinite* and *periodic*, with a period of N elements.

The effect of a cyclical convolution on two 1-dimensional vectors can be seen in Figure 3.1. The linear convolution is performed by padding the data. The cyclically convolved vector is thus accomplished by using no padding.

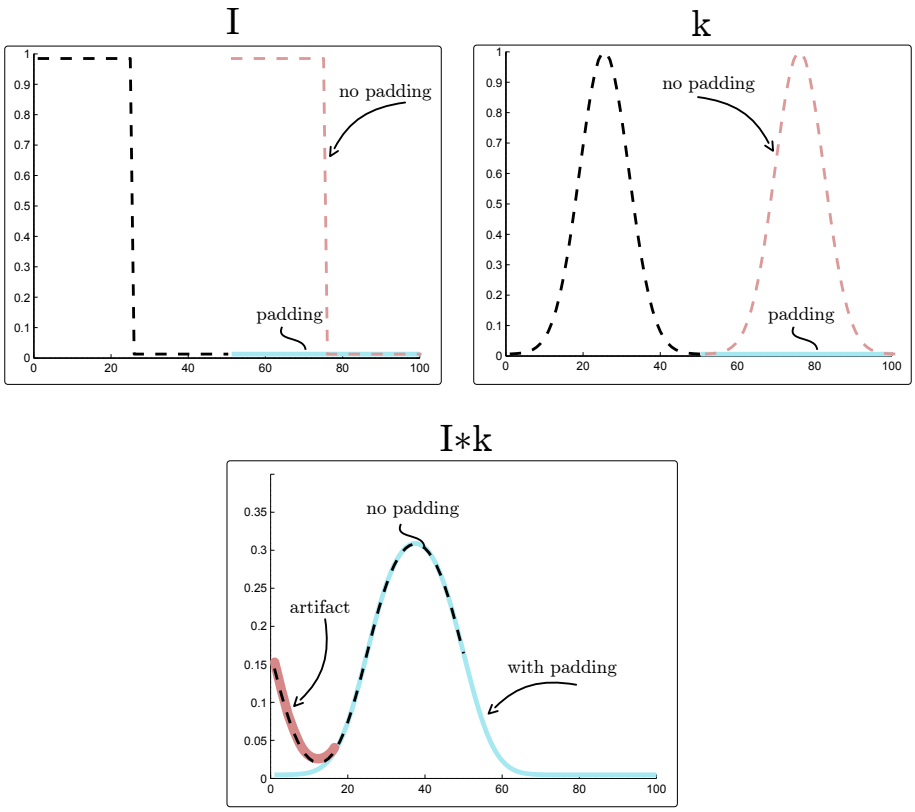


Figure 3.1: Convolution between image $\mathbf{I}$ and kernel $\mathbf{K}$. The functions are assumed to be periodic, because of FFT, which introduces a "bleeding" artifact.

Linear convolution is still cyclical, but by extending the period with N zeros, the period gets just large enough to avoid other periods to affect elements inside the relevant part of the convolution.

One can see it as the padding is providing a type of buffer, so the convolution only picks the zero-padded part, essentially ignoring the values. In Figure 3.1 the relevant part is within the interval $x \in [1, 50]$.

There is also an alternative method of convolving matrices, called *overlap-add* [Smi97, chap. 18]. It works by dividing the image into segments and applying the kernel to those segments, and then adding them together. In this thesis we will only use the ordinary FFT convolution, as described above.

We have so far only described how to pad vectors to produce a linear convolution. To convolve $M \times N$ two-dimensional images, or matrices, we need to add N zeros for each row, and M zeros for each column. In total $3MN$ zeros must be added.

In this thesis we *pad* and *unpad* the data on the CPU. It is possible to send *only* the image and pad on the GPU for a speedier transfer between the host and device. We do not take this into account but we give an estimated speedup using this method in Section 5.5.

3.2 FFT11

The fast Fourier transform could easily be parallelised using the separability of the two-dimensional FFT [CG00]. The following equation is commonly called the *row-column* method. In this report, we are using the method extensively and we therefore choose the short and concise name FFT_{11}, which reference Matlab naming syntax. We can thus express FFT_2 as

$$\text{FFT}_2(\mathbf{I}) = \text{FFT}_1(\text{FFT}_1(\mathbf{I})^\top)^\top \tag{7}$$

Using Equation 7, we can construct a two-dimensional FFT by computing one-dimensional FFTs of each column and then transpose the result and take one-dimensional FFT of each column of the result, and then end with a transpose. This means that we can send blocks of rows, perform FFT on these smaller blocks instead of sending one large image. We will delve deeper into this idea later in Section 5.2.4 and then later, in Section 5.3.3 where we use it for convolution.

3.2.1 FFT11 optimisations

Observe that we can skip calculating many elements in the first pass of the FFT_{11} algorithm, see Figure 3.2. The schematic three pass algorithm is depicted in Figure 3.2 where **B**, **C** and **D** are zeros. We realise that at the first pass we only need to compute half of the FFTs, since FFT of a zero vector is a zero vector. This is simple to see by looking at Equation 3.

The Matlab code in Listing 2 shows how to compute the Fourier transform with this optimization. The size of **I** is $2N \times 2N$.

Listing 2: FFT11 optimization 1

```
O = complex( zeros(N, N, 'single') );
C = complex( zeros(2*N, 2*N, 'single') );

C(1:N, 1:2*N) = fft( [I; O] ).';%transform and transpose into place
C = fft(C).';
```

Tests show that Matlab does not check if the input contains all zeros. Passing FFT with only zeros is a rare occurrence so doing a check would not be feasible. The check itself takes about 5% of the time for a vector of 2^{26} elements. Since we know that half of the FFTs in the first pass

of FFT_{11} consist of zeros (because of the padding), we can skip the extra computations and do the first transpose in a block. We will re-use this idea when we examine convolution using FFT_{11} in Section 5.3.3.

Also observe that the third pass only involves a transpose. As simple as this operation may seem, this step is more complicated to speed up optimally, than one might think. On the GPU, achieving optimal performance has been proved to be very involved [RM09].

The FFT_{11} algorithm is important to utilise because of the memory constraints of the graphics hardware. We will use this technique to fit the data with gpuArray, see Section 5.3.

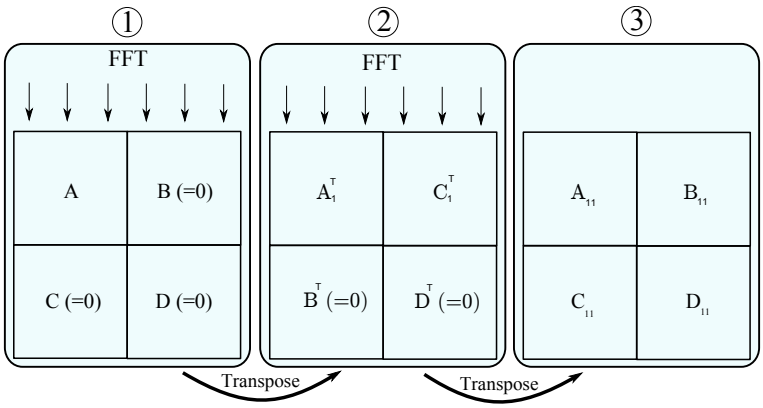


Figure 3.2: A schematic figure of how FFT_{11} is calculated. B, C, D is all zeros in the first step of the algorithm. For the next step, B and D will contain zeros. A transpose is simply moving the zeros into place.

SECTION 4

Implementation

In this section we will introduce the programming environment Matlab (Section 4.1) and compare its performance with C. We will cover calling C functions from Matlab via the Matlab Executable (MEX) interface (Section 4.2), several compilation routes and sample code, which is covered in Section 4.3.

4.1 MATLAB

Matlab is a numerical computing environment with programming capabilities. Matlab was created in the late seventies, initially developed to make use of LINPACK and EISPACK, numerical libraries, written in Fortran. The libraries are provided and maintained by Netlib [Net]. Matlab continued to develop and became a programming environment with many advanced programming capabilities. Because of its origins, the Matlab syntax resembles Fortran in many ways. However, Fortran code is compiled directly into machine code as opposed to Matlab, which is interpreted. One of the disadvantages to interpreted languages is slow execution. To combat this, interpreted languages use a technique called Just-In-Time (JIT). JIT compiles frequently used code to machine code, making the code run faster. We will now see how JIT is used when speeding up loops in Matlab. Later we will study the same code in C and then how to make the code multi-threaded using Open MultiProcessing (OpenMP).

In the Listings 3, 4, 5 and 6 we see examples of how we can speed up the code using pre-compiled functions like `sum`. The codes were timed (not seen in the code) with `tic` and `toc`. In the first code, we have included the pre-allocation of the array, but it is not a part of the timing.

Listing 3: C-style looping

```
N = 1e3;
M = magic(N); %Forming a magic matrix
s = 0;        %the sum of all the elements

for i=1:N
  for j=1:N
    s = s + M(i, j);
  end
end
```

In the first code (Listing 3) we use a C-style looping to sum a matrix M. This is efficient in C but not in Matlab.

Listing 4: Loop interchange

```
for j=1:N
  for i=1:N
    s = s + M(i, j);
  end
end
```

By flipping the indices when accessing the matrix M we make use of the fact that a matrix in Matlab and Fortran is stored column–major linearly in memory [Col]. This technique does not change the performance in a significant way if not JIT is enabled, as seen in Table 4.1.

Listing 5: Vectorising inner loop with sum

```
for j=1:N
  s = s + sum( M(:, j) );
end
```

In Listing 5 we vectorise the inner loop. In this case JIT will most probably optimise the loop and vectorise much like as seen in Listing 6.

Listing 6: Fully vectorised summing with two sums

```
s = sum( sum(M) );
```

Observe that the timing for the un-accelerated vectorised code in Table 4.1 is faster than the accelerated code. When testing this, we used Matlab R2011b. On 2012a, the timings of the un-accelerated code was actually faster. We assume the faster code is due to JIT taking more time to optimise code on-the-fly in version 2012a than 2011b.

Many optimisations have been implemented in Matlab, such as "in place" computation [Shu07] (Matlab R2006b [Get09]), for conserving memory during computation. Calling `A = fft(A)` would invoke the in place version of `fft` and conserve memory.

Another achievement in making Matlab faster was the introduction of multi-threaded computation in Matlab 2007a [Get09]. The multi-threaded computations and in place memory conserving technique are probably one of the most important optimisations made to Matlab.

The following table shows the performance from JIT acceleration using `feature accel off` and `on`. The speed factor is the timing divided by the fastest timing Fully vectorised with JIT activated.

	JIT	no JIT
Code	Duration (factor)	Duration (factor)
C–style sum	48 s (263)	299 s (1635)
Flipping rows and columns	4.3 s (23)	253 s (1388)
Vectorising inner loop	0.7 s (3.9)	0.74 s (4)
Fully vectorised	0.18 s (1)	0.18 s (0.98)

Table 4.1: Table of performance of different codes in Matlab.

To compare with a compiling language we will use Gnu Compiler Collection (GCC). We use GCC when testing FFTW (see Section 5.2.1) and Spiral FFT (see Section 5.2.6) later because it is easier to compile than using the MEX compiler.

We implemented the same code as above in C and compiled with GCC with different optimization flags. Table 4.2 shows normalised values of the timings.

The fastest code (using optimization flag `-O3` and SSE optimization) is twice as fast than the fastest code on Matlab (the value with factor 1). The reason for this modest speedup is due to the simplicity of the code. In a more complex situation, compiled C code will provide better speedup than we see here.

To get a speedier code in Matlab, it is always a good idea to vectorise the code as much as possible. In some cases, using the `function` Matlab keyword may accelerated the code even more. We saw this especially when testing FFT_{11} convolution in Section 5.3.3.

An important part of JIT is that it may speed up code but it is difficult to predict what optimisations may be inferred by the accelerator. One reason for this might be because JIT and Matlab changes with each version. To get code to run fast relying on that JIT does its job well and vectorises the code when possible is an important factor.

flag	Duration (factor)
-	1 s (12.5)
O	0.354s (4.3)
O1	0.35 s (4.2)
O2	0.34 s (4.3)
O3	0.17 s (2.12)
O3 SSE	0.08 (1)

Table 4.2: Table of performance of different flags passed to the GCC compiler. SSE = `-march=native -mfpmath=sse`

Even if the code is vectorised, compiled C code is still faster. To be able to get the fastest FFT code possible, we have to use compiled code. We also saw that the use of optimization flags are also important. Fortunately, Matlab provides a programming interface to call C code called MEX. This means that we are able to create our own function in C, compile it, apply optimizing flags to our compiler and run it directly from the Matlab environment. Another advantage using Matlab is *multi-threading*. To accomplish this in C we can use an open source library called OpenMP. The OpenMP library makes it very simple to parallelise code, see Listing 7.

Listing 7: Threaded code, using OpenMP

```
//data is defined as an array of integers
int s = 0;

#pragma omp parallel default(shared) private(i, j)
{
#pragma omp for reduction(+:s)
  for(i=0; i<N; i++)
    for(j=0; j<N; j++)
      s += data[i][j] //equivalent to *(data + i*N + j)
}
```

In Matlab, as mentioned earlier, contiguous data is stored column-by-column (called column-major). Because indexing of memory is cached ahead, it is therefore more efficient to access the matrices with the inner loop stepping in row-by-row. For the C language, the data is internally stored linearly row-by-row (row-major), accessing and updating data by columns is more efficient. Since we have these two ways of internally representing a matrix, it is important to note that by sending data between these programming languages we actually transpose the matrix. Another thing to note: a complex matrix in Matlab is stored internally as two 1-dimensional arrays (matrices). One array contains the real elements and the other contains the imaginary elements. We will have to convert from this data structure when we use CUFFT, as we will see later.

4.2 CALLING C CODE FROM MATLAB WITH MEX

The previous section showed Matlab as a relatively fast (if the programmer is careful) programming environment. Matlab provides a simple programming language with a clever accelerating interpreter called JIT.

Unfortunately, we also saw an instance when the JIT actually made the wrong decisions and ended up trying to accelerate a code that did not need further acceleration and that the accelerations that work for one version of Matlab may not work for the next.

MEX is a way to call functions written in C (and other languages like C++ and Fortran) using a MEX library and a generic function name

```
#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
```

We can name our arguments as we please but for sake of consistency we keep the recommended *mathematical* terminology in accordance with Matlab. `Nlhs` means "number of left hand side (output arguments)", `nrhs` "number of right hand side (input arguments)" `plhs` means "pointer to left hand side (output)" `prhs` "pointer to right hand side (input arguments)".

A function `f` is thus defined as

```
[plhs[0], plhs[1], ..., plhs[nlhs-1]] = f(prhs[0], prhs[1],..., prhs[nrhs-1])
```

The pointers are pointing to `mxArray` which is a contiguous array layed out linearly in memory. In C, a matrix is stored *row-major*, meaning the elements are layed out row after row, as in Equation 8.

$$\text{mxArray } array_{row} = \underbrace{\{1,2,3,4,5,6,7,8\}}_{\text{Memory}} = \underbrace{\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}}_{\text{Interpretation}} \tag{8}$$

In Matlab and Fortran, the memory is layed out column-by-column, as in Equation 9. This is important to note when sending data to C via MEX, especially when calling libraries, as we will see in Section 5.2.5.

$$\text{mxArray } array_{col} = \underbrace{\{1,5,2,6,3,7,4,8\}}_{\text{Memory}} = \underbrace{\begin{pmatrix} 1 & 5 \\ 2 & 6 \\ 3 & 7 \\ 4 & 8 \end{pmatrix}}_{\text{Interpretation}} \tag{9}$$

4.3 COMPILATION WORKFLOW

Compiling CUDA code withMEX can be accomplished in several ways. In Figure 4.1 we see the different routes that could be taken to create CUDA-enabled code from Matlab. Our flow of compilation is indicated in the figure.

We can compile CUDA code (.cu) with NVCC creating a PTX code. PTX is a type of CUDA assembler code, which can be executed within Matlab. The problem is, we cannot use any external libraries to link which is what we need.

There are also CUDA-enabled MEX files which lets the user code write the code into a single CUDA file and handle calls to CUDA kernels and libraries. Special scripts has been created to compile with MEX and NVCC [Liy08]. Unfortunately, the different versions of Matlab and VS (Microsoft Visual Studio) did not work together, so we used another route.

Our method consists of a CUDA file consisting of kernel calls, and a MEX file with the interface to Matlab. We compile the CUDA file using NVCC and create an object file (.o) which we link and compile with MEX in the second step of the compilation process. The result is a `mexw64` file. This file can then be called from Matlab like any other function call.

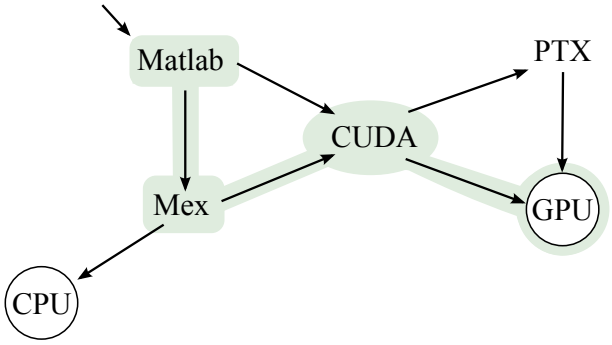


Figure 4.1: Data flow over compilation stages to GPU and CPU, starting from Matlab. The longest valid path is 3 jumps long. The only path that is not valid, and is longer than 3 jumps is Matlab → Mex → CUDA → PTX → GPU is not valid.

We have defined the variables `mexfile`, `cudafile` and `objfile`. These contain C/C++ code, CUDA code and the intermediate Obj file respectively. The resulting file is named after `mexfile`, with suffix `mexw64`.

Listing 8: Compiling CUDA and C files

```
str = ['!del ', objfile]; %if nvcc fails we force Mex to stop
eval(str)

!setEnv.Cmd /x64

try
  cudart = ' -lcuda -lcudart -lcufft';
  VS = '"C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\lib"';
  %NVCC STAGE:
  str = ['!nvcc -c ', cudafile, ' -o ', objfile, ' -arch compute_20 -I. -I...
      "', getenv('CUDA_INC_PATH'), '" -L', VS, '-L"', getenv('...
      CUDA_LIB_PATH'), '" -lcudart -lcufft']
  eval(str)
  %MEX STAGE:
  str = [' mex ', mexfile, ' ', objfile, ' ', timing, ' -I"', getenv('...
      CUDA_INC_PATH'), '" -L"', getenv('CUDA_LIB_PATH'), '" -lcudart -...
      lcufft']

  eval(str)
catch exception
  rethrow(exception)
end
```

The basic steps of the compilation workflow we used are

1. Write the MEX function (C or C++), used as an entry point to CUDA function
2. Write your CUDA code (.cu)
3. Compile and link C++ code and CUDA code

The compilation step, as seen in Listing 8 assumes the NVCC, linker and compiler is setup correctly. For more information, see Section 8.2.

SECTION 5

Results

In this section we will compare between FFT and convolving matrices on the GPU and the CPU using Matlabs gpuArray, built-in FFT functions and a selection of libraries. For the first set of tests, we only attempt to speed up the FFT computation and assess how to speed it up more, using GPU and CPU. The second set of tests are focused on the convolution algorithm itself. In those tests, we will implement a real world convolution algorithm on one channel and compare it to other implementations.

For all tests in this report we use a mid range system running 64 bit Windows 7, Matlab 64 bit 2011b, Intel Core i5-2500K 2.2 GHz, *n*Vidia Geforce GTX 570, and 8 GB DDR3 RAM (1600 MHz). This system is balanced, scoring overall $7.5 - 7.9$ units using Windows Experience Index. The CUDA display driver that was used in all of the benchmarks was 302.59 CUDA 5.0 Release candidate (released June 2012).

In the application VISSLA™, the workflow forces the convolution to be performed just before a point in time. The application supplies the user with buttons and sliders which modifies the simulation of the eye, and hence the shape of the kernel just until convolution is to be performed.

One speedup, which we have not included in our comparison, is to generate the kernel data on the GPU, instead of sending it.

5.1 COMPARING PERFORMANCE

To time our code correctly and reliably, we used high resolution timing in C with Windows `QueryPerformanceTimer`. Users on forums has voiced their concern about the accuracy of the timer on dual processor systems [Nee]. To time code execution is thus not trivial. We found that the best way to get accurate timings, where overhead in the API calls to the timing function would be negligible, is to place the timed code in a loop. By timing the loop, we could get an accurate mean timing. This is also especially useful when timing short parts of a code, for instance when we benchmark Spiral FFT (see Section 5.2.6). For more information about timing tools in Matlab, see [KCM11]. For GPU timing, we used `gtoc`, which was provided in the test suite `gpuBench`, see Section 5.2.3. For CUDA we used *CUDA events*, see Section 5.2.5. In the test code in Listing 9, we check the return types of the functions if they fail, which is omitted in this code for brevity.

Listing 9: Timing CPU code using performance counter

```
#include <Windows.h>

LARGE_INTEGER ticksPerSecond;
LARGE_INTEGER start_ticks, end_ticks, cputime;

void tic()
{
  QueryPerformanceFrequency(&ticksPerSecond);
  QueryPerformanceCounter(&start_ticks);
}

double toc()
{
  double time;

  QueryPerformanceCounter(&end_ticks);
  cputime.QuadPart = end_ticks.QuadPart - start_ticks.QuadPart;
  time = ((double)cputime.QuadPart / (double)ticksPerSecond.QuadPart);

  return time;
}
```

5.2 FFT

In this section, we will benchmark FFT using both CPU and GPU libraries. We will try out different approaches to speed up our code. The most promising techniques will be used in the convolution part of the results (see Section 5.3).

FFT in Matlab is built on the FFTW library. The library itself is written in C. We want to investigate to make sure the threading of FFTW is optimal, so we downloaded the dlls for FFTW. In the next section we will examine FFTW for C/C++.

5.2.1 FFTw

FFTW is an open source library specifically optimized for performing FFT. The Matlab FFT function is built on FFTW, and in newer Matlab versions, the FFT function is threaded to run in parallel. Tests show that running one-dimensional FFT on 4 cores is about twice as fast than running on one core (using `singleCompThread` flag). Performing two-dimensional FFT is about 1.87 times faster on 4 cores than on a single core. Thus the threaded FFT efficiency is about 46-50% for the above examples. We know that FFT is easily parallelisable using FFT_{11}. We can also use a special symmetry on one-dimensional transforms called the *Hermitian symmetry*

$$X_{N-k} = \overline{X_k} \tag{10}$$

This equation states that an FFT of a real valued input is symmetrical, apart from a complex conjugate.

In Table 5.1, we compare different techniques and compiler flags for one-dimensional transforms. To see the speedup of using Equation 10, we can change the plan to transform "real to complex" data. However, FFTW will only return half of the output, we have to "untangle" the output for ourselves. It is easy to modify the output to be identical to the ordinary complex output (which is

required for convolution). We modified the plan to use "real to complex" (`R2C`) and "complex to complex" (`C2C`) transforms. Additionally, we tested the double precision version of FFTW. The mean of 2000 runs were collected, where $N = 8192$. In all tests we used the `estimate` planning.

According to the FFTW manual [Fri99], power-of-two sizes are generally faster to transform, and sizes with large factors are very inefficient to transform[1]. We will revisit this idea later when we compare convolution algorithms on the GPU and the CPU, in Section 5.3.

Code	Duration (factor)
Matlab (avg)	51 μs (2.04)
FFTW	115μs (4.6)
C2C double	61 μs (2.44)
float	34 μs (1.36)
-O3	40 μs (1.6)
R2C -O1	32 μs (1.28)
-O2	31 μs (1.24)
-O3	25 μs (1)

Table 5.1: Table of performance of FFTW from C and Matlab using different compiler options, precisions and plans.

Below we will briefly see how we can modify a plan to speed up the FFT computation.

FFTW plan The FFTW library internally store information about the transform to be executed. This is called a *plan*. By using the function call `fftw_print_plan(plan)` for a one-dimensional real single precision transform plan, we get the following printout

```
(dft-ct-dit/4
  (dftw-direct-4/12 "t1fuv_4_sse2")
  (dft-vrank>=1-x4/1
    (dft-bluestein-503/1024
      (dft-ct-dit/32
        (dftw-direct-32/32 "t3fv_32_avx")
        (dft-buffered-32-x32/32-6
          (dft-direct-32-x32 "n2fv_32_avx")
          (dft-r2hc-1
            (rdft-rank0-iter-ci/64-x32))
          (dft-nop))))))
```

The output shows that FFTW plans are interpreted as a kind of recipe, telling the main algorithm how to divide the data and which transformation code to use. We can see that the name Bluestein is used, which is just another FFT algorithm. Note also that FFTW also use SSE instructions, as is also seen in the printout.

FFTW plans are part of an advanced planning structure that is related to FFTWs "wisdom". A wisdom is a type of data base that is collected using a *Planner*. When a wisdom is created, the planner is trying different plans, based on the provided input size and method [Wow]. The different types of methods available for Matlab FFTW are: `estimate`, `measure`, `patient`, `exhaustive` and `hybrid` [Matb]. The fastest method is guaranteed to be found and added to the wisdom if the planner uses the `exhaustive` method. The planner takes longer to execute compared to e.g., `estimate`. What makes this efficient is that a wisdom data base can be saved to a file and reused at a later time for a faster executed transform.

[1] For instance 8192 (2^{13}) is efficient, but the slightly smaller size $2^{13} - 1$ is a (Mersenne) prime number, thus a very inefficient dimension to transform, even though a slightly smaller amount of data has to be transformed.

Duration	percent speedup
1.505 s	1.177
1.278 s	1.009
1.266 s	1.028
1.232 s	1.011
1.218 s	1.020

Table 5.2: Table showing the repeated runs to FFT inside Matlab. The right column shows the decreasing quotient of two consecutive runs.

The speedups provided with planning could be something to explore in more detail. However, we will not delve more into this subject in this thesis.

FFT libraries such as CUFFT use a similar plan data structure as FFTW. As we will show in Section 5.2.5, there are differences in how these plans are used.

In the next section we will investigate Matlab FFT and how to use the aforementioned Hermitian symmetry speedup.

5.2.2 Matlab FFT

We have observed that the performance of the FFT function within Matlab got slightly faster after we ran it twice. As seen in Table 5.2, the performance of a one-dimensional FFT (2^{26}) using four threads is shown to be improving after the first run. This is referred to as a *warm up*. We will use the same idea in our tests with FFT using gpuArray, in Section 5.2.3, and Jacket in Section 5.3.5.

An *in-place algorithm*, or function is defined as a function which transforms the input data using constant memory. Often, the input is overwritten by the output. This is what we mean when we say "in-place".

We found that performing an in-place FFT_2 on real data was not faster than performing the same computation on complex data. The result can be seen in Listing 11.

Listing 10: Comparing complex to real 2d transform in Matlab

```
>> N=8e3; A=rand(N, N, 'single'); tic; A=fft2(A); toc
Elapsed time is 1.463705 seconds.

>> A=complex(rand(N, N, 'single')); tic; A=fft2(A); toc
Elapsed time is 1.418840 seconds.
```

Obviously, the real matrix should be faster to transform, thanks to the Hermitian symmetry. Because the transform outputs a complex matrix, we conjectured that the code has to allocate extra memory and copy the result in the new matrix and back to $\mathbf{A}$. However, if we let $\mathbf{A}$ be a real matrix, and store the output in a complex matrix $\mathbf{B}$, we get the same performance.

Since there is a discrepancy in the result, we tried to speed up the FFT_2 by using FFT_{11}, as first described in Section 3.2 on page 23. By repeating the experiment in Listing 11 we notice that FFT_{11} is about 16% faster than FFT_2. The difference between transforming real and complex is about 4%. This depends on the size of the input.

Listing 11: Comparing complex to real 2d transform in Matlab using the row-colum method

```
>> A = complex(rand(N,N,'single')); tic; A = fft(fft(A).').'; toc
Elapsed time is 1.270736 seconds.

>> A = rand(N,N,'single'); tic; A = fft(fft(A).').'; toc
Elapsed time is 1.212485 seconds.
```

We will use this technique on the GPU in the next section, and also in the convolution algorithm, see Section 5.3.3.

We also explored the threading management of FFT_2. The number of threads that FFTW use depend on how many processors are present on the system. In fact, Matlab does no longer allow users to control how many threads to run on [Matd]. This is because Matlab can potentially call other libraries and has therefore not total control over the number of threads used by those processes, which is preferable. The command `maxNumCompThreads(N)` does not affect FFT_2 and will be deprecated in future versions of Matlab (but it is still there, as of Matlab version 2012a).

However, we can limit Matlab and its sub-processes by starting Matlab with the flag `-singleCompThread`. By doing this, we limit the number of threads to one. We get that an FFT on a vector of 2^{25} single precision elements, on roughly 2.5 s on one thread, on a four core system. The same calculation using four threads takes 1 s. We get 2.5 times faster code with four threads as opposed to one. This is only a 60% speedup.

5.2.3 gpuArray

By using the Matlab *Parallel Computing Toolbox* [Par], the user is presented with new accelerated built-in functions using the GPU from Matlab. Some of the functions that are GPU-accelerated includes, but are not limited to: `fft`, `fft2`, element-wise multiplication (`.*`) and matrix transpose (.'). We will use these functions in our implementations later.

One issue that arises when using gpuArray is that a complex matrix packaged in gpuArray allocates 5 times more memory than a real matrix (all single precision). After applying FFT_2, the memory of the same matrix is 7 times larger than the real matrix. We also found that by clearing the GPU memory after each call, the memory of a complex matrix is 4 times larger than a real matrix and FFT_2 of that matrix requires 3 times more memory. Our conclusion is that GpuArray manages memory in an unpredictable way, which severely limits the use of GPU memory.

We also tried to find the exact maximum memory allocatable by gpuArray. By doing a *binary search*, we found that only 33% of the memory could be successfully allocated. The code can be seen in Listing 12.

Listing 12: Binary searching peak memory allocation on gpuArray

```
reset(parallel.gpu.GPUDevice.current())
gr = gpuDevice(1)

totalMemory = gr.FreeMemory/2^20;
lastMem  = 0;
imax     = totalMemory;
imin     = 1;%minimum memory
N        = 0;

%Binary search
while imax >= imin

  imid = ceil((imin + imax)/2); %try new allocation size
  skip = false;

  try
    N  = round( sqrt( imid*2^20/(2*4*2) ) );
    %(N^2*4*2)/2^20 => N = sqrt( M*2^20/(2*4) )

    Ag = gpuArray( complex(rand(N, N, 'single')) );
  catch exp

    skip = true;
  end

  if (skip) imax = imid-1; else imin = imid+1; end

  lastMem = gr.FreeMemory/2^20;
end

fprintf(1, 'Maximum memory utilisation: %i percent\n',  round(100*imid/...
    totalMemory) )
```

We have so far been unsuccessful in finding any resources that discuss the usage of GPU memory for gpuArray.

Since we find gpuArray to be a very simple way to access the GPU, we will continue and attempt to benchmark it.

To be able to benchmark our codes using gpuArray, we used a script called `gpuBench` [Tor12]. Using gpuArray in that setting, we get a 10-20 times speedup compared to FFT using the CPU. We soon found out gpuBench did not account for sending and retrieving the data from the GPU, which is important for our purposes. To accomplish this, we need to use the function `gather`. In Figure 5.2, we compare CPU with the `gather` version of gpuArray. Computing with gpuArray without `gather` is about 4-6 times faster than with `gather`, giving us comparable timings to the benchmark.

An issue with only using Matlabs `tic toc` for GPU application is that a GPU call is *asynchronous*. Calling a GPU function returns the CPU the control directly after calling it, not after it is completed on the *GPU*. We can see the effect in Figure 5.1.

For a synchronous call, the host will wait for the kernel to finish, and `toc` will work as expected. In the case of asynchronous calls, host and device continue to run their code, and thus, the ordinary `toc` will not work. We must use `gtoc` to get the correct timing, as seen in Figure 5.1.

In Matlab 2011b and prior versions, the gpuArray calls are synchronous. In Matlab 2012a,

Mathworks switched to asynchronous calls. We will use this feature to speed up the convolution in Section 5.3.7.

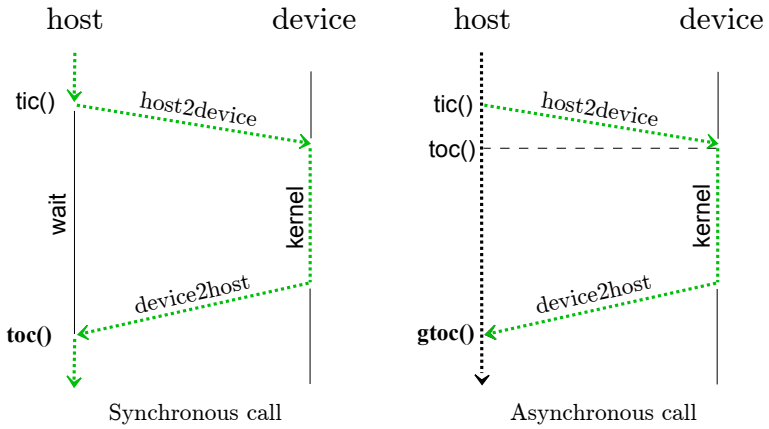


Figure 5.1: Synchronous and asynchronous calls to `tic toc` and `tic gtoc`.

In Figure 5.2, we compare the computation of FFTs on matrices of size 2^n, where $n \in \mathbb{N}$. The circles show separate tests using sizes that we know FFT is optimal.

We found that the FFT routine on the GPU is much slower for the first run compared to subsequent runs. This resembles the speedup we saw in Section 5.2.2. In order to fix this, we used a *warm up*-function, which runs a small computation using the GPU before running the real tests. We will see this idea again in Section 5.3.5.

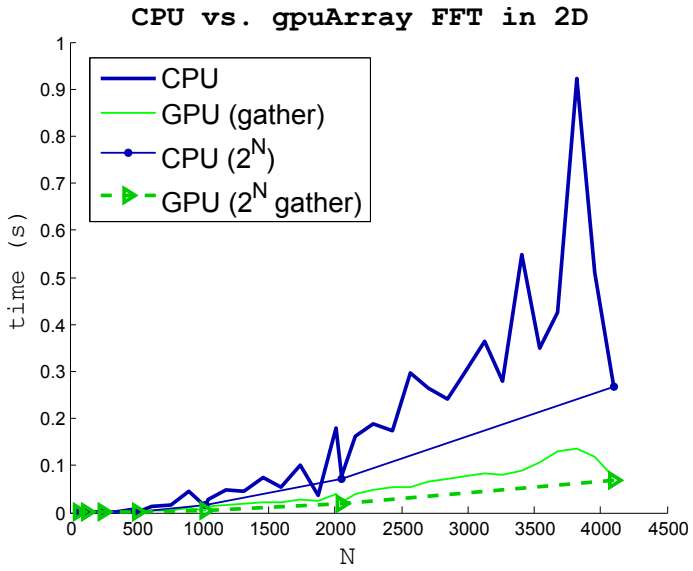


Figure 5.2: Performance between FFTW (Matlab) and FFT using gpuArray. Notice that the timings of the FFT is coinciding with the pure 2^n sizes of 1024 and 2048.

We found that FFTW in Matlab is difficult to benchmark because it uses an internal wisdom data structure (see Section 5.2.1 in page 32) but also because of Matlabs JIT accelerator. We believe it is very difficult to measure performance within Matlab, because the tests made earlier in the same script file could possibly affect the performance of the code. To combat this, we tried to run the tests several times to give unbiased results.

In the next section we will present the results on using FFT_{11} with gpuArray.

5.2.4 FFT11 using gpuArray

Since GPU is limited in memory we could use FFT_{11} to compute the FFT. This section will only show the result of the transform, to see the source code see Section 5.3.4.

Using gpuArray and transforming a 8192×8192 matrix is not possible because of lack of memory. By using FFT_{11}, we can send smaller blocks that fit GPU memory to accomplish a full two-dimensional transform. Figure 5.3 shows how we send different block sizes of data to be transformed. The larger the blocks are, the faster the transform becomes.

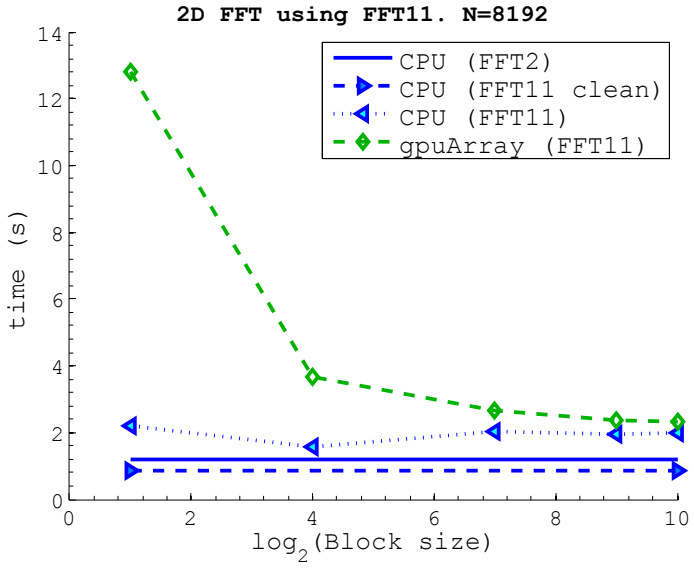


Figure 5.3: Result of running gpuArray on different sized blocks of matrices. As the chunk size increases, the speed increases, but the size allocated is also increased.

In the next section we will use CUFFT to perform our transforms. We will see different approaches to speeding up the code. We found that some techniques speeded up the code but where unfortunately not consistently faster.

5.2.5 CUFFT

Our setup consists of calling CUFFT library using MEX as an interface. This allows us to call the CUDA FFT routine directly from within Matlab.

We program using CUDA C, a programming language similar to C that exposes the functionality of GPU programming. The difference between CPU programming and GPU programming is that the GPU is multi core, consisting of between hundreds to thousands of cores. The programming style proposed by CUDA programming guide is based on using a block, grid and thread hierarchy to identify threads. For more information, the reader is kindly referred to [Nvi]. For an example of how threads are setup, see Listing 13. Later on in this section we will see how to execute a kernel using threads, see Listing 16 on page 42. First, we will examine the way CUFFT manages memory.

Listing 13: Thread, grid and block hierarchy

```
int block_size_x = 32; //MAX_THREADS=1024, sqrt(1024) = 32
int block_size_y = block_size_x;

dim3 dimBlock(block_size_x, block_size_y, 1);
dim3 dimGrid(M/dimBlock.x, N/dimBlock.y);

if (M % block_size_x !=0 )
  dimGrid.x+=1;

if (N % block_size_y !=0 )
  dimGrid.y+=1;
```

One important issue with GPU computation is the management of memory. To send data from Matlab via MEX, we need to allocate memory on the device first. We use the variable name `tmp_d`, a $M \times N$ matrix, which resides on the device. To allocate, we call `cudaMalloc((void **) &tmp_d, sizeof(float)*M*N)`.
To send the data residing on the host, from `tmp_h` to `tmp_d`, we call `cudaMemcpy(tmp_d, tmp_h, sizeof(float)*M*N, flag)` where `flag` is `cudaMemcpyHostToDevice`. To send the data back to the host, we use the flag `cudaMemcpyDeviceToHost`.

For convenience, we will introduce a measure of memory that we will call a *unit*.

Definition. *1* unit = $M \cdot N \cdot 4/2^{20}$ *MB for input matrix of size* $M \times N$.

This metric was not chosen haphazardly. It is the smallest measure of memory for the real pointer to the MEX structure, half of `MX_COMPLEX`. This measure can also be used to describe the allocated memory of a `R2C` plan. We will also use this metric when estimating the amount of memory allocated by a GPU code. This is important, since we only have a limited amount of memory on the GPU. We will explore this in detail in Section 5.3.2.

CUFFT works much like FFTW, using a plan to describe to the FFT routine, and what type of transform we want to perform. The following code sets up a two-dimensional plan for a $(M \times N)$ matrix $\mathbf{A}$

```
cufftHandle plan;
cufftPlan2d(&plan, N, M, CUFFT_C2C)
```

The second and third arguments denotes rows and columns in $\mathbf{A}$, we have to state N and M since we effectively transpose the data when sending it to C from MEX. A transpose is thus unnecessary, we only need to switch the dimensions.

In CUDA, the plan allocates memory, as opposed to FFTW that creates algorithmic recipes. Our research show that the amount of memory that is allocated somehow depends on the factoring of the input size. As previously stated in [DKE10], the allocated memory of a CUFFT plan is between 1 and 4 times the size of the input.

Additionally, according to [FJ12], the time it takes to compute a transform depends on the factoring of the size of the input. If the input size can be factored into large prime factors, the computation will take longer. The reason is the FFT routine use general purpose routines for these sizes. As previously stated, FFTW is fast for matrices whose size can be factored into small primes, such as power-of-two.

To gain more insight about the memory consumption of a CUFFT-plan, we tested different input sizes and measured the memory before and after plan creation using

```
cudaMemGetInfo(&memfree, &memtotal);
```

By plotting the allocated memory of a couple of plans, we noticed an *upper* and *lower* bound of the memory allocated. According to [KS11] the lower bound is given by the size of the input, i.e., 2 units, and the upper bound is 4 times larger. We found that the upper bound is defined by $2 \cdot (N + \delta)^2 \cdot 4 \cdot 2/2^{20}$, where $\delta = 121$, see Figure 5.4. The reason for using our new estimate is, there is a discrepancy of about 18 MB for large matrices, and this might overflow memory resulting in an error.

As previously stated, FFTW is slower for near-prime size inputs. This however, is not true for CUFFT. We did not find any common denominator between the factors of input size and the allocation size of the plan. Additionally, we found that $\delta \approx 160$ for non-quadratic sizes.

As an example, the space of a plan for transforming 6800^2 is about 350 MB; increasing the dimensions to 6801^2, the plan require in excess of 1.1 GB ($\approx$ 1500 according to our calculations).

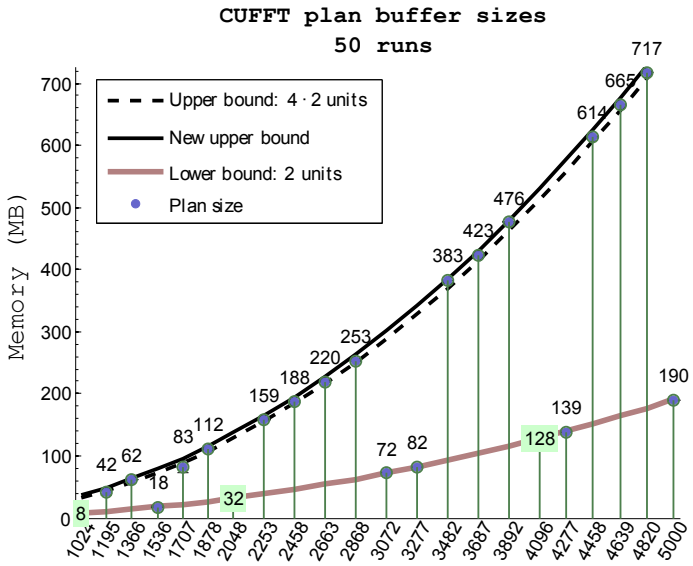


Figure 5.4: CUFFT Plan memory sizes for different quadratic matrices with dimensions N. In the figure we propose a new upper bound of the memory required.

We believe the reason the plans takes up so much memory is because the plan actually contains buffers for the calculation. In the documentation for CUFFT [NVI12] it is mentioned that FFT routine is "in place" FFT. However, because of the plan buffer, the routine still use outside buffers, and is thus not strictly speaking "in place".

We found that the simplest way to find a small plan size is to *always use dimensions that are multiples of 100*. This equation is very simple, but for the benefit of referencing it later, we will define this as a function. Given the dimensions of the input M and N, the function *optimal_dim* returns the dimensions of a matrix where the plan size is minimal

$$\text{optimal\_dim}(M, N) = \{100 \cdot \lceil M/100 \rceil, 100 \cdot \lceil N/100 \rceil\}. \tag{11}$$

FFT using CUFFT Since the code is written in C, we use the *Windows Performance Query* to time the CUDA function calls. The placement of the timing can be seen in Figure 5.5.

An important part of the code consists of allocating memory and copying data from via MEX to the CUFFT native complex array structure, called `cufftcomplex`. `cufftComplex` is an

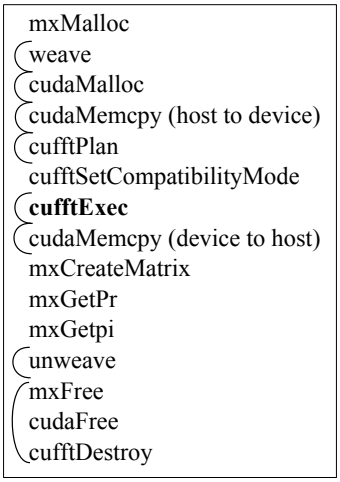


Figure 5.5: Layout of the CUFFT code and the placement of timers as indicated by arcs.

interleaved array of real and imaginary data. When using MEX, the complex class (or type) is called `MX_COMPLEX`. it is a struct consisting of two separate pointers to arrays. Hence, we need to "weave" the data into the `cufftComplex` data structure, before calling the FFT routine, and unweave the data back into two arrays, after the routine is done.

Figure 5.5 shows the positions of the different timers shown in Figure 5.6. The latter shows the relative timings of the different parts of the code for different sizes of input. It is important to point out that weaving and unweaving the data takes a relatively long time to finish.

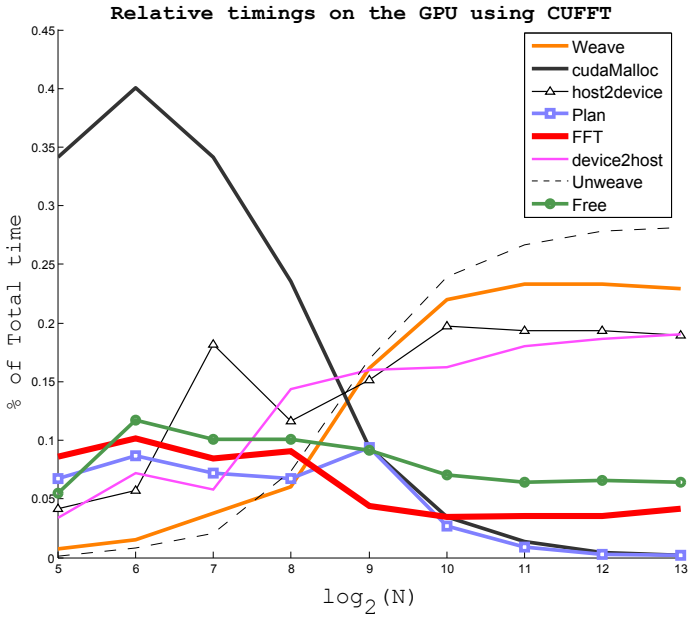


Figure 5.6: CUFFT_1 relative timings. Note that *weave*, *unweave* and transfer functions takes the most of the time for larger input sizes

This version of FFT on the GPU is approximately 1.6 times faster than FFT_2 using Matlab. The GPU calls are made *asynchronously*, meaning that control returns to the CPU immediately after a kernel launch. In Matlab versions before 2012a, the GPU calls are made synchronously [Matc]. With CUFFT, we are also able to do the computation in-place (compare to gpuArray). We call `cufftExecC2C` by referring to pointers `idata` and `odata`, input and output respectively. If we send the same pointer as input and output, we get an in-place FFT (compare to

gpuArray).

```
cufftExecC2C( cufftHandle plan, cufftComplex *idata, cufftComplex *odata,
              int direction );
```

To use the Hermitian version we could use `cufftExecR2C`. This version would only give us the non-redundant Fourier coefficients (compare to Section 5.2.1). This routine is possibly twice as fast as `C2C`. After some tests with this routine, and a kernel which rearranges the data into a full complex matrix, the result was not correct. We did not pursue this idea further. The important fact to note is that a `R2C` only requires one unit, but at the expense of having to write your own "untangle" kernel.[2]

Even better, by sending the whole matrix to the GPU and performing FFT_{11}, we could use a one-dimensional plan, which is then the size of the input. For an $N = M = 8192$ matrix, we need 512 MB for the ordinary two-dimensional FFT. If we use the FFT_{11} on the device, we only need 2 units which is less than 1MBin the one-dimensional case. Research [DKE10] show that performing FFT_{11} on the device is $\approx 35\%$ faster than `C2C` two-dimensional CUFFT. In the same paper, they also transform larger matrices by sending data in blocks to the device using the FFT_{11} algorithm. We will try out these methods later using CUDA and gpuArray.

To ensure the timings are correct, we need to use a similar function as `gtoc` (see Section 5.1). We call `cudaThreadSynchronize` which works as a barrier, waiting for all threads on the device to finish before continuing.

Speeding up CUFFT We revisit the MEX file and the results seen in Figure 5.6. We see that the packing and unpacking portion of the program is relatively slow. First, we will try to speed up the packing by using OpenMP and threading the code in different ways, see Listing 14.

Listing 14: Threading weaving using OpenMP

```
#pragma omp parallel default(none) private(i) shared(Ntot, input_re, ...
    output_float)
{
  #pragma omp for schedule(static)
  for (i = 0; i < Ntot; i++)
  {
    output_float[i].x = input_re[i];
    output_float[i].y = 0.0f;
  }
}
```

We have also attempted to parallelise `mxMalloc` and `cudaMalloc` using two threads (of the four).

We found that weaving could be sped up 20% using 4 threads. Unweaving using threads was 48% faster. These results were recorded in our tests, but for an unknown reason, the results were not reproducible.

Even though threading on the CPU-side did not work (nor did we really expect it to), the GPU consists of thousands of threads and thus should be very fast. This method is mentioned in [Too, slide 41] (in the function `real2complex`), but they only provide code for quadratic input sizes. With a little extra effort, we managed to get the code to work for any size input, see Listing 15.

[2] We used an "untangle kernel" provided on the *n*Vidia forums. Unfortunately, because the forums has been down the last couple of months, we cannot cite to the specific thread.

It might be interesting to note that [Mud09] discuss methods of rearranging data on the GPU. However, we failed to find any implementations of this library openly available.

Listing 15: Weaving data on a GPU kernel

```
__global__ void weavecomplex (cufftComplex *c, float *a, int M, int N)
{
int idx = blockIdx.x*blockDim.x + threadIdx.x;
int idy = blockIdx.y*blockDim.y + threadIdx.y;

  if(idx<M && idy<N)
  {
    int index = idx + idy*M;
    c[index].x  = a[index];
    c[index].y  = 0.f;
  }
}
```

We call with the code in Listing 16.

Listing 16: Calling the kernel function to weave data

```
weavecomplex<<<dimGrid, dimBlock>>>(rhs_complex_d1, a_d, M, N);
```

This code is a standard method of indexing using CUDA threads and the *thread hierarchy* mentioned in Section 5.2.5. To measure the time spent in the CUDA device we used *Cuda Events*. The code in Listing 17 shows that we can use the code in the same way as with the Performance Query timer code

Listing 17: Timing GPU code using events

```
cudaEvent_t start, stop;

void tic()
{
  cudaEventRecord(start, 0);
}

float toc()
{
  float elapsedTime=0.0f;

  cudaEventCreate(&stop);

  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);
  cudaThreadSynchronize();
  cudaEventElapsedTime(&elapsedTime, start, stop);

  return elapsedTime/1000.0;
}
```

The reason `cudaMalloc` takes time might be because CUDA, much like Matlab, needs time to initialize code. CUDA has a JIT accelerator which compiles code at run-time [Nvi].

We followed the advice in [Van06] of defining environment variables `CUDA_DEVCODE_CACHE`, but we found no difference in speed when doing this. We also tried to see if initialization was the problem by following the suggestion in [Sta]. By calling a CUDA run time function,

e.g., `cudaFree`, a new CUDA context will be initialised. We did not detect any noticeable delay due to CUDA initialization, and therefore, we conclude that the time of initialization is negligible.

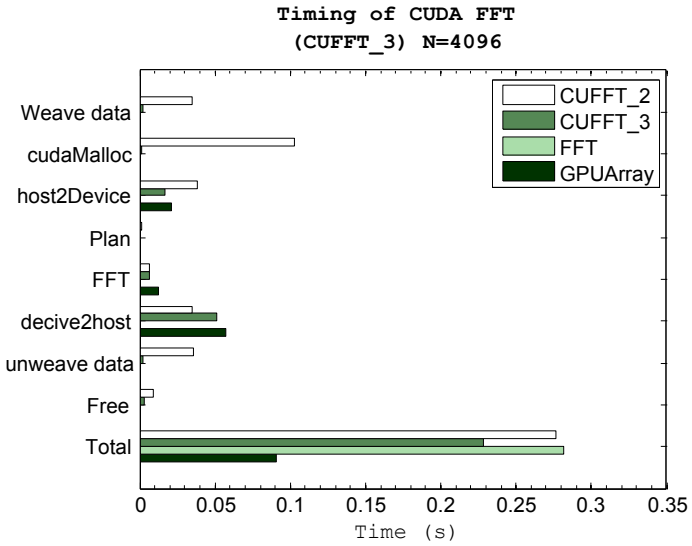


Figure 5.7: Comparison between gpuArray, FFT_2, `CUFFT_2` and `CUFFT_3`. The new version of our CUFFT code weaves and unweaves the data on a customised kernel.

We observed that the total time spent on the device is about 0.084 s, about the same time as gpuArray. However, timing the device function shows that the total time is spent between the device function call and the MEX function.

To confirm the results, we can calculate the speed of transmitting data from host to device. We send $(4096^2 \cdot 4)/2^{20}$ MB of data in 0.0021 s. The transfer is thus 3047 MB/s, compared to 3780 MB/s as CUDA Software Development Kit (SDK) 5.0 bandwidth benchmark show. The weaving performance is about 15 times faster on the GPU than on the CPU. The reason `host2Device` is twice as fast on `CUFFT_3` is because we only send the real part of the input, since we rearrange the data on the GPU and initialize the imaginary part with zeros. We will briefly focus on another FFT library for the CPU, which is very fast and might be helpful for others.

5.2.6 Spiral FFT

Spiral FFT is a library developed at Carnegie Mellon University, Pennsylvania. The FFT library consists of power-of-two hard coded input data, which were generated from an algorithmic framework described in [MPV11].

The data in Spiral is interlaced, like `cufftComplex`. On the CPU side. We have enough memory to perform FFT on two images concurrently, and we also do not need to communicate between device and host, which is costly. In Spiral we have, just like FFTW, special plans taking real input data. The one-dimensional FFT of real data follows the *Hermitian* symmetry, given by Equation 10.

By using the Real to Complex (`R2C`) version of Spiral, we double the speed of the transform for $N = 8192$, see Table 5.3. Tests show that the extra time spent on copying data is negligible. If we compare the real Spiral FFT to Matlab FFT (using a single thread), the Spiral FFT is 4-5 times faster.

Flag and method	Duration
-O3 C2C	$35\mu s$
-O3 R2C	$18\mu s$

Table 5.3: Comparison between different runs of Spiral. The timing includes copying data, to give identical outputs for both methods.

Because of the limited input sizes, only one-dimensional transforms and comparable speedup to FFTW, we chose not to pursue this alternative any further.

We found it particularly difficult to find any instructions on how to compile our code. We found out that SSE instructions were used in the source code. We then solved the compilation problem by using (in GCC) `-march=native -mfpmath=sse`.

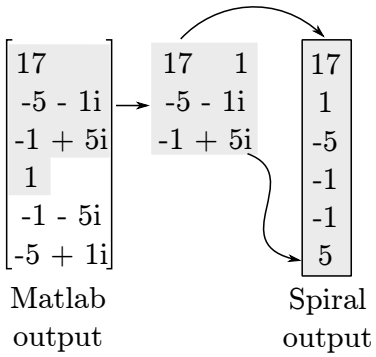


Figure 5.8: The output data layout from Spiral for a 1-dimensional real FFT.

5.2.7 O-Matrix

O-Matrix [Oma] is a language and programming environment with a very similar syntax to Matlab. O-Matrix is built specifically for High performance. The benchmarks, provided on the homepage of O-Matrix, gives a comparison to other software such as Matlab. According to the homepage [Oma] (under benchmarks), O-Matrix is running twice as fast as Matlab. Unfortunately, the benchmarks are old (from 2008), and there is no indication that more current benchmarks, comparing to the latest version of Matlab is going to be posted.

We downloaded the trial version called *O-Matrix Light* and called FFT_2 and compared it with Matlab. The result was, surprisingly, that O-Matrix was on average 4 times **slower** than Matlab. We do not know if O-Matrix Light is limited in any way, compared to the full version of O-Matrix. We notice via the task manager that the execution is in fact multi-threaded.

We emailed the developer for support and explanation, to no avail.

Listing 18: FFT2 in O-Matrix

```
interrupt(5)
clear
clock

N = 3000
A = real(rand(N,N))
A = fft2d(A);
```

The Listing 18 took 0.57 s using O-Matrix, and 0.09 s using Matlab. Matlab is therefore ≈ 6.3 times faster than the O-Matrix FFT routine.

We also tested GNU/Octave, which is also similar to Matlab. We found that it is twice as slow as Matlabs FFT, although Octave calls the FFTW library, as is evident in the installation directory.

In the next section we will list other libraries that were considered, but ultimately not tested.

5.2.8 Other libraries

We found many free libraries that looked promising, but as reported by the FFTW benchmarks [Fftc], or simply not currently updated, such as Bernstein DJBFFT, these were not tested:

1. IPP, Intel® Integrated Performance Primitives. According to benchmarks on FFTW homepage [Fftc] this library is slower than FFTW for our purposes.
2. MKL (Math Kernel library) [Mkl] is a library for the CPU, mentioned in many benchmarks [Fftc; Gov+08] but are according to FFTWs benchmark [Fftc] slower than FFTW.
3. Nukada FFT is a GPU code, which seemed promising at first but was ultimately not tested because of time constraints. According to the benchmark on their web page [Nuk11] the benchmarks has not been updated since January 2011 and cannot thus reflect its performance accurately.

We eventually tested all the FFT libraries that were either not listed in any benchmark or was faster than FFTW, for our purposes.

5.3 FAST CONVOLUTION

In the previous section about FFT, we focused on speeding up the FFT routine on the CPU and GPU. We concluded that the GPU is much faster in computing FFT but was slow in transferring data and limited in memory. In this section, we will attempt to implement convolution of two matrices using CUDA, gpuArray and compare to FFT on Matlab. We will minimise data transfer and memory consumption to produce a fast convolution algorithm. Additionally, we will revisit the FFT_{11} algorithm and propose improvements and compare the gain in speed using the classical approach.

We also concluded that FFT on the CPU generally is faster for input sizes where the factors are small. This is especially important when we compare CUDA to CPU convolution, see Section 5.3.2.

Since we test many different hardware and algorithmic solutions, we have to be mindful of the quality of the result. We generally found (unless otherwise stated) that the relative error of the convolution algorithms we tested were within the ϵ_{mach} value, which is $\approx 1 \cdot 10^{-6}$ for single precision floats. We used the relative error metric, as seen in [Fftb]. We define the relative error η as

$$\eta = \frac{||\tilde{\mathbf{A}} - \mathbf{A}||_\infty}{||\mathbf{A}||_\infty}$$

for the approximation $\tilde{\mathbf{A}}$. We use as exact solution $\mathbf{A}$, which was calculated using the double precision `fft`. Note: The relative error is only good to use when the elements of the matrices are approximately the same size. In our case the matrix elements are all $\in [0, 6]$.

We will use square matrices of carefully picked sizes. These sizes can be seen, with their allocated memory in CUDA, with their factors in Table 5.4. The sizes require a minimum amount of memory in CUFFT. We found these sizes by running and storing all different CUFFT plan sizes and picking the local minimums using `[p inds] = findpeaks(-planmem, 'minpeakdistance', ws)`. The variable `ws` is the *window size*, the assumed number of elements apart, these minimum plan sizes are. We then picked 20 of them to use in the comparisons.

Size	Memory	Factors	Size	Memory	Factors
1056	8	$2^5 \cdot 3 \cdot 11$	5115	199	$3 \cdot 5 \cdot 11 \cdot 31$
1339	13	$13 \cdot 103$	5150	202	$2 \cdot 5^2 \cdot 103$
2365	42	$5 \cdot 11 \cdot 43$	5334	217	$2 \cdot 3 \cdot 7 \cdot 127$
2580	50	$2^2 \cdot 3 \cdot 5 \cdot 43$	5432	225	$2^3 \cdot 7 \cdot 97$
2712	56	$2^3 \cdot 3 \cdot 113$	5661	244	$3^2 \cdot 17 \cdot 37$
3007	68	$31 \cdot 97$	5733	250	$3^2 \cdot 7^2 \cdot 13$
3596	98	$2^2 \cdot 29 \cdot 31$	6586	330	$2 \cdot 37 \cdot 89$
3750	107	$2 \cdot 3 \cdot 5^4$	7168	392	$2^{10} \cdot 7$
3808	110	$2^5 \cdot 7 \cdot 17$	7462	424	$2 \cdot 7 \cdot 13 \cdot 41$
4118	129	$2 \cdot 29 \cdot 71$	7854	470	$2 \cdot 3 \cdot 7 \cdot 11 \cdot 17$

Table 5.4: The different sizes, allocated memory for each, and the factors.

5.3.1 CPU

Convolving images using the CPU is very straight forward. We tested convolving images using plan sizes that requires minimum memory in CUFFT, to be able to compare the GPU and the CPU for large sizes. We found that the performance of FFTW does not depend on the same criteria as CUFFT. This is even more clear in the convolution algorithm, as opposed to using a single FFT routine as in Section 5.2.1. We perform the simple convolution using the convolution algorithm as seen in Listing 19.

Listing 19: Convolution using CPU

```
A   = zeros(N, N, 'single');
tmp = phantom('Modified Shepp-Logan', N/2);
A(1:N/2, 1:N/2) = tmp; %padding image
B   = fspecial('gaussian', N, 10);

C   = ifft2(fft2(A).*fft2(B));

%Below was omitted in the comparisons
C   = fftshift(C);
C   = C(1:N/2, 1:N/2);
```

According to the Matlab documentation [Mata], `fft` is slower for inputs with large prime factors, as seen in Figure 5.9. Since the decomposition of the dimensions are $6586 = 2 \cdot 37 \cdot 89$ and $7168 = 2^{10} \cdot 7$, this explains the time difference between the methods. We wanted to know if the reason for the method being slower depends on memory usage, as we hypothesised is the case of CUFFT.

On a side note, in Matlab, using the profiler, we can get the timings of functions. Additionally, by adding the following *undocumented feature* [Und] `profile('-memory','on');` we can get memory allocation. Unfortunately this feature does not show usage over time, which would be helpful.

We can compare the CPU and memory usage for the two inputs, 6586 and 7168. The latter contains 18% more elements but is 76% faster than the former. FFTW is not using the same techniques as CUFFT as can be seen by observing the memory usage. While executing these codes, Process Explorer [Pro] was run to produce the graph in Figure 5.10. The graph was "beautified" and vectorised. The software provides an update frequency of 0.5 s and can thus provide a high resolution of the CPU and memory usage of an application over time. We observe a slight increase of CPU activity depicted in Figure 5.9 as a red area. This is the kernel CPU

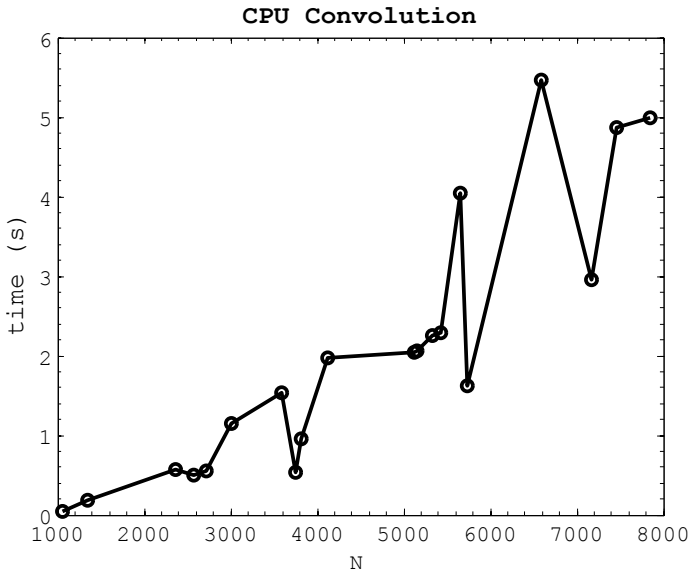


Figure 5.9: CPU Convolution performance. Note the difference of timing for different input sizes.

usage, which means that functions very close to the hardware is run at that time. We believe that FFTW makes calls to special accelerated hardware, making it faster to execute.

In Figure 5.10 we can clearly see that there are very little difference in CPU, and memory usage compared to what we observed in CUFFT (Section 5.2.5). We see that 7168 requires more memory, though completes faster.

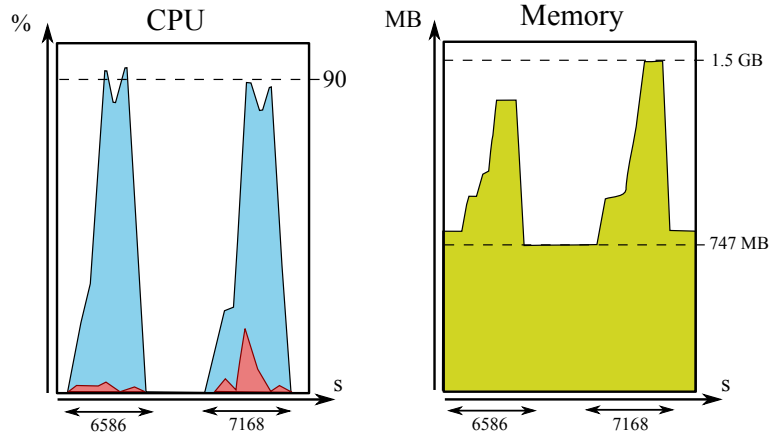


Figure 5.10: The CPU usage and memory footprint of Matlab during Convolution provided by the *Process Explorer*. The time to perform a convolution of 7168 is faster than 6586 but it is not evident in the CPU usage or memory.

In our comparison table (see the Appendix in page 67), we did not take this information into account when comparing our proposed CUDA Convolution code with the convolution algorithm on the CPU.

5.3.2 CUDA

As our previous attempts at speeding up FFT show, sending data and computing only one FFT on the GPU proved to be time consuming and did not speed up the code significantly. By sending all the data once and performing convolution on the GPU and then sending it back, we could save time.

One issue that arises is memory management on the GPU. We can see in Table 5.5 and 5.6 the pseudo code of the convolution algorithm with respect to memory requirement. The table use the *unit*, defined in Section 5.2.5 on page 38.

We do not show the full code mainly because no additional memory is allocated after the plan creation. The table is the source code, row by row, where the columns describe the number of units allocated.

One way to combat the constraints of the GPU memory is to switch unnecessary allocations and reuse memory and arrange the allocation in a way to minimise the maximum allocated memory. The theoretical minimum amount of memory allocated in our algorithm is 6 units. We need to perform FFT on both images, we need 4 units for the data and 2 units for the plan. This is the minimum amount of memory (we can zero-pad by using Equation 11 on page 39 to get minimal plan sizes for all sizes). To convince us of this, we need to realise that transferring data require 2 units for cufftComplex and 1 unit for the float. No matter how we rearrange the code, we always need to transform the other matrix and keep the transformed first matrix in memory. The total minimum memory is thus 6 units. The complete code for both MEX and CUDA can be seen in Section 9.

Pseudo code	Diff	Total
Allocate real data a_real	1	1
Allocate real data b_real	1	2
Allocate a_complex	2	4
Allocate b_complex	2	6
Copy a to a_real	0	6
Copy b to b_real	0	6
Weave a_real to a_complex	0	6
Weave b_real to b_complex	0	6
Create Plan	2	8

Table 5.5: Deconstruction of the CUDA algorithm memory pattern, version 1. This is the "naïve" way of organising the code.

Pseudo code	Diff	Total
Allocate real data a_real	1	1
Copy a to a_real	0	1
Allocate a_complex	2	3
Weave a_real to a_complex	0	3
Copy b to a_real	0	3
Allocate b_complex	2	5
Weave a_real to b_complex	0	5
Free a_real	-1	4
Create Plan	2	6

Table 5.6: Deconstruction of the CUDA "Mem-save" version, memory pattern version 2. Note that we reuse `a_real` when sending `b` from host to device.

We can create plans and destroying them just as we compute the FFT, but this is not necessary, and will take additional time. The code described in Table 5.6 is optimal with respect to reusing data.

The two versions are equally fast, as seen in Figure 5.11. However, for some reason, when the GPU memory is almost full, the speed of both the methods severely decrease. We can also see a table of the performance between the CPU and our proposed convolution algorithm in 5.17 on page 63. In that figure, we test non-square convolution, and we notice a similar pattern of a sudden decrease in speed when the memory is almost full.

We sometimes experienced that the code crashed. We conclude that this was due to our trying to allocate too much memory. We simply solved this by only running our code when we know for sure the data would fit on the GPU. If CUDA tries to allocate too much memory using

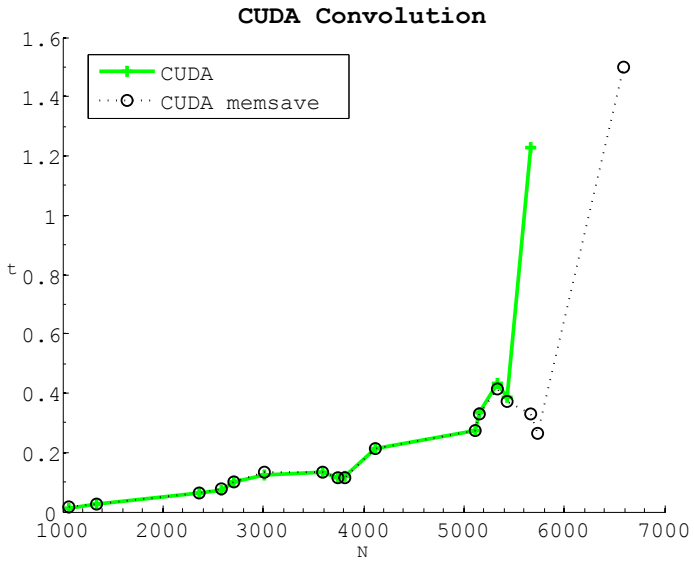


Figure 5.11: CUDA Convolution performance comparing the "memory saving" code and our first version. We can see that the performance of the methods decrease just before the memory is full.

`cudaMalloc`, CUDA will end up in an error state, from which it cannot escape. All subsequent attempts on executing the code will fail.

To fix this, our code is wrapped inside a Matlab function which tests how much memory is left on the device and returns an error if we try to allocate too much memory.

We also found that if we try to allocate a plan that is larger than the available memory on the device, CUDA will not end up in an error state and simply do nothing. We take care of this information and simply stop the computation, preventing the CUDA from ending up in the "error cycle".

To further ensure our code is stable, we implemented detailed CUDA error messaging, which is passed forward using MEX. See Section 9 for the code.

Another interesting fact is that the speed of the convolution in Figure 5.11 is remarkably linear, compared to FFT on the CPU. We did not include power-of-two sizes, because we wanted the sizes to only minimise the CUFFT plan buffer. However, these sizes, as indicated in Figure 5.9, still provide very fast transform for some sizes.

One important thing to also note is that CUDA seems to propagate old errors. This was also reported by Parrish in [Par12]. Our code is very stable because it is careful not to allocate too much memory.

5.3.3 Revisiting FFT11

The speedup provided by the FFT_{11} method is, as we have previously seen, very limited on the GPU (see Section 5.2.4). However, using FFT on the CPU has been proved to be faster than using FFT_2, as shown in Section 5.2.2. We will test using FFT_{11} for convolving images on CPU and in the next section using gpuArray. We do not really expect any major speedup using gpuArray compared to the CPU but we will examine these in detail to make sure.

As we continued researching the FFT_{11} method, we realised that some of the transposes could be removed. The optimization is simple and as it turned out, has been found before by Al Umairy et al. (2010) in [AU+10]. However, in their paper, they only describe the optimization briefly.

We will give this idea a proper introduction and compare this to the original convolution method along with other speed up techniques.

The idea is, if we simply apply FFT_{11} to the convolution algorithm on the two $M \times N$ matrices $\mathbf{I}$ and $\mathbf{K}$, we have

$$\mathbf{I} * \mathbf{K} = \text{IFFT}_2(\text{FFT}_1(\text{FFT}_1(\mathbf{I})^\top)^\top \circ \text{FFT}_1(\text{FFT}_1(\mathbf{K})^\top)^\top). \tag{12}$$

The inverse FFT_2 can also be separated such as in the *row-column method*. Equation 12 is thus expanded to

$$\mathbf{I} * \mathbf{K} = \text{IFFT}_1(\text{IFFT}_1(\text{FFT}_1(\text{FFT}_1(\mathbf{I})^\top)^\top \circ \text{FFT}_1(\text{FFT}_1(\mathbf{K})^\top)^\top)^\top)^\top). \tag{13}$$

We will introduce the generic (complex) matrix symbols $\mathbf{A}$ and $\mathbf{B}$ to describe the identities we apply. We return to Equation 12 and apply the following identity

$$\mathbf{A}^\top \circ \mathbf{B}^\top = (\mathbf{A} \circ \mathbf{B})^\top. \tag{14}$$

We apply Equation 14 to 12 and label the result with $\mathbf{A}$ and carry a transpose, we get

$$\mathbf{I} * \mathbf{K} = \text{IFFT}_2(\underbrace{[\text{FFT}_1(\text{FFT}_1(\mathbf{I})^\top) \circ \text{FFT}_1(\text{FFT}_1(\mathbf{K})^\top)]^\top}_{\mathbf{A}^\top}) \Leftrightarrow \text{IFFT}_2(\mathbf{A}^\top). \tag{15}$$

We use the following identity

$$\text{IFFT}_2(\mathbf{A}^\top) = \text{IFFT}_2(\mathbf{A})^\top, \tag{16}$$

which is easy to see is valid by the following argument: The row-column method separates the FFT_2 into one-dimensional computation of the rows and then the columns. The one-dimensional transform can therefore be applied to columns then rows, instead of rows and then columns, because of the transpose, we only swap the summation order.

If we expand the inverse FFT from the right hand side of Equation 15, and apply 16, we get

$$\text{IFFT}_1(\text{IFFT}_1(\mathbf{A}^\top)^\top)^\top \Leftrightarrow^{16} \text{IFFT}_1(\text{IFFT}_1(\mathbf{A})^\top)^{\top\,\top} \Leftrightarrow \text{IFFT}_1(\text{IFFT}_1(\mathbf{A})^\top). \tag{17}$$

We have by the identities 14 and 16 removed half of the transposes from the convolution algorithm using FFT_{11} and the inverse FFT_{11}. The final equation is thus

$$\mathbf{I} * \mathbf{K} = \text{IFFT}_1(\text{IFFT}_1(\text{FFT}_1(\text{FFT}_1(\mathbf{I})^\top) \circ \text{FFT}_1(\text{FFT}_1(\mathbf{K})^\top)^\top). \tag{18}$$

Since FFT_{11} is (sometimes) faster than FFT_2 (Section 5.2.2), we will now compare several variations of FFT_{11} used for convolution. We will combine the *stripped* method, as seen in Equation 18. We will also revisit the *pad trick*, first introduced in Section 3.2.1.

The codes can be seen in Listings: (naïve) 20, (stripped) 21, (stripped and padding trick 1) 22 (stripped and padding trick 2) 23. The two last methods are very similar.

All the codes are wrapped within a `function`. By wrapping the code in a function we saw a performance speedup difference between 30% and 44%. The detailed performances are shown below.

Listing 20 will convolve the $N \times N$ matrices $\mathbf{A}$ and $\mathbf{B}$, which are already zero-padded.

Listing 20: FFT11 ordinary version

```
%Initialization:
tmp1 = complex(0*A);
tmp2 = complex(0*A);

tmp1 = fft( fft(A).' ).';
tmp2 = fft( fft(B).' ).';
tmp1 = tmp1.*tmp2;
tmp1 = ifft( ifft(tmp1).' ).';
```

The above code was 44% faster by using `function`.

Listing 21: FFT11 stripped

```
tmp1 = fft( fft(A).' );
tmp2 = fft( fft(B).' );
tmp1 = tmp1.*tmp2;
tmp1 = ifft(ifft(tmp1).');
```

The stripped version shows a marginal speedup using `function`.

The next two listings (22 and 23) shows two variations on the stripped FFT_{11} using the *zero pad trick*. This works best if the padding is required for both the image *and* the kernel (this might not always be the case). The variables `Asub` and `Bsub` are the $N/2 \times N/2$ sub-matrices containing the non-zero elements. We also introduce the zero matrix `O`.

Listing 22: FFT11 stripped + zero pad trick 1

```
O    = 0*Asub; %zero matrix
tmp1 = fft( [fft([Asub; O]).'; O, O] );
tmp2 = fft( [fft([Bsub; O]).'; O, O] );
tmp1 = ifft( ifft(tmp1.*tmp2).' );
```

The stripped zero pad version 1 showed a 34 % speedup using `function`.

The last method stores the result in a zero matrix, instead of adding zeros, as in Listing 22. This method is also the fastest. All the codes in Figure 5.12 were wrapped in the `function` syntax. The last code showed little or no speedup using the `function` syntax.

Listing 23: FFT11 stripped + zero pad trick 2

```
tmp1(1:N, 1:2*N) = fft([Asub; O]).';
tmp1             = fft(tmp1);

tmp2(1:N, 1:2*N) = fft([Bsub; O]).';
tmp2             = fft(tmp2);

tmp1             = tmp1.*tmp2;
tmp1             = ifft(ifft(tmp1).');
```

We have shown that the fastest method is at its best 2.7 times faster than the ordinary FFT_2 convolution by simply reducing the applying. Please note that these results are approximate. We

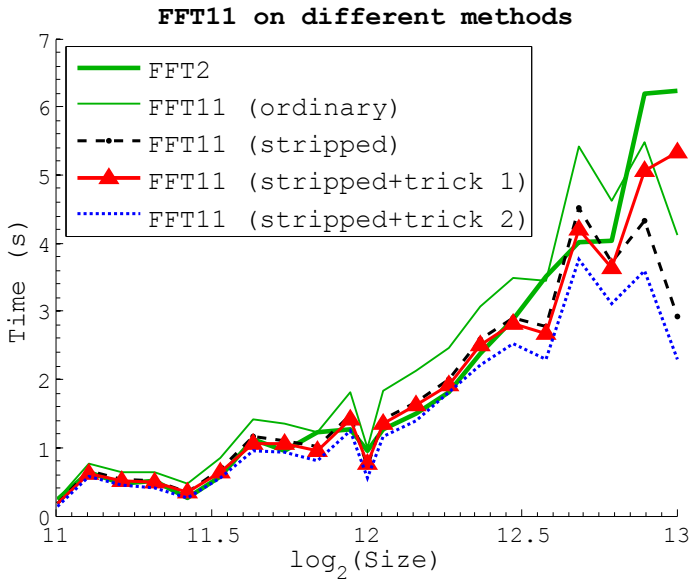


Figure 5.12: The figure shows a comparison between several methods of FFT11

found that the performance of these methods fluctuate, but we believe that the indicated speedups are relatively accurate.

The question of quality is important when using FFT_{11}, since we perform FFT on the rows and columns. We found that the three stripped FFT_{11} algorithms were of the same accuracy. This means that by not computing FFT_{11} of zero matrices, we do not increase the accuracy of the solution at all. The average relative error η of FFT_2 was 4.7630e-007 compared to 3.7779e-007, which is very accurate.

Since our first test did not present any substantial speedup, we chose to digress from the strict comparison of convolution algorithms and try to speed up the code as much as possible. By padding both the image *and* the kernel, we depart from the way it is performed in VISSLA™, where we only pad the image. This padding scheme is however the most generic method, as we see it.

Using the FFT_{11} algorithm on the CPU did provide us with a speedup. The speedups are modest until the matrices reaches $N = 2^{13}$. We have shown that speedup of a efficient code, such as FFT could be faster by using better memory management.

We will now return to the comparisons and use the FFT_{11} algorithm with gpuArray and see if it can speed up the computations.

5.3.4 Matlab gpuArray

When we use gpuArray to perform convolution on the GPU, we can either fit both matrices in memory, or use FFT_{11} to send blocks of data to the GPU. By using FFT_{11}, we can get larger convolutions, but at the price of sending data to the GPU several times, which is slow.

In 2011 Karas and Svoboda compared different methods of sending blocks of data to perform FFT. They modified the FFT algorithm to allow sending of frequency domain data in blocks.

We will use the previously used method (Section 5.2.3) of sending columns of data and applying one-dimensional FFT on the columns. Unfortunately, we still have the issue with memory management of the gpuArray, which means we cannot allocate as much memory as one would

think. We found that gpuArray seem to use double precision results from FFT. We exchange the 4 to an 8 and get

$$m_{data} = \frac{2 \cdot 8 \cdot MN}{2^{20}} \tag{19}$$

We can compute the "block size", the number of columns that can fit on the GPU using the following

$$\text{blocksize} = \left\lfloor \frac{m_{dev} 2^{20}}{2 \cdot 8M} \right\rfloor \tag{20}$$

where m_{dev} is the amount of device memory left. If *blocksize* does not divide N, the rest of the columns are

$$\text{rest} = N \bmod \text{blocksize} \tag{21}$$

The following code (Listings 24 and 25) shows the FFT_{11} algorithm using blocks to convolve the matrices $\mathbf{A}$ and $\mathbf{B}$. We use the newly acquired transpose trick, introduced in Section 5.3.3.

Listing 24: FFT11 on gpuArray sending blocks of data

```
function C = gpuArray_FFT11(A, B)

try
  [M N] = size(A);

  w = floor(2^24/N);

  if w>N, w = N; end

  A  = subfft(A, w).';
  A  = subfft(A, w);

  B  = subfft(B, w).';
  B  = subfft(B, w);

  C = subifft( subifft(A.*B, w).', w);

catch exception
  rethrow(exception)
end

end
```

Listing 25: Perform fft of the columns on gpu in the block size specified

```
function res = subfft(A, blocksize)

[R C] = size(A);
res   = 0*A;

for k=1:floor(C/blocksize)
  res(:, (k-1)*blocksize+1:k*blocksize) =...
     gather( fft( gpuArray(A(:, (k-1)*blocksize+1:k*blocksize)) ) );
end

rest = mod(C, blocksize);
res(:, end-rest+1:end) = gather( fft( gpuArray(A(:, end-rest+1:end) ) ) );
end
```

In Figure 5.13 we see the performances between the ordinary convolution algorithm and FFT_{11} using gpuArray. In this figure we show the impact of a "cold start" i.e., without warmup. Had we used a warmup, the speed for the first size would be as expected. This effect is only evident when we use FFT_2 and not FFT_{11}. This could be because FFT_2 is the first function call and also very time consuming, compared to the first call to FFT_{11}.

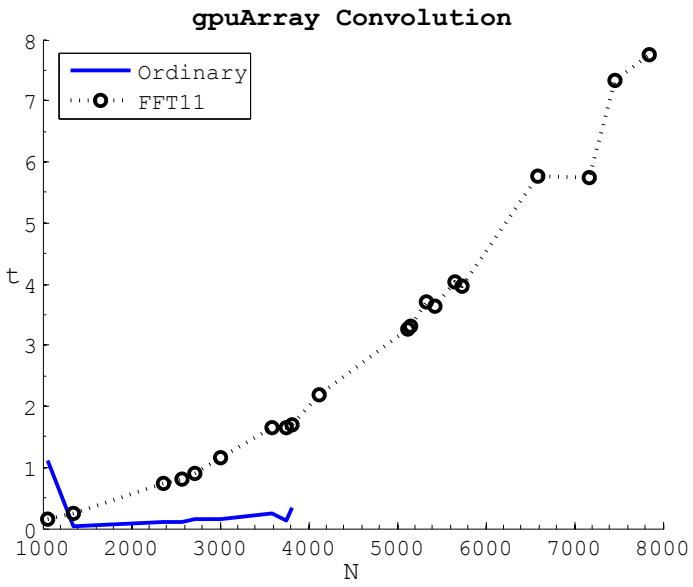


Figure 5.13: Convolution with FFT_{11} and ordinary using gpuArray. Note the significant delay in the first computation.

We will later reuse this code to get a *hybrid* method, using both CPU and GPU resources, see Section 5.3.7. In the next section, we will try a GPU library which claims to be several times faster than gpuArray and in some instances hundred times faster than the CPU for selected functions.

5.3.5 Jacket

Jacket is a GPU library, developed by Accelereyes. Jacket provides a simple integration of GPU code, callable from Matlab. This makes it possible to utilise GPU acceleration without the need to

create your own CUDA kernel code. We downloaded Jacket v2.2 (build 77be88c) by AccelerEyes (64-bit Windows) from their homepage [Jac].

We wrote our test code using code shipping with Jacket. The timing function in the Jacket benchmark code is `timeit`, this is the timer AccelerEyes recommend us to use. By looking in the installation folder of Jacket, we found that the timer function is a `.p` file, which is a binary file created from Matlab, meaning, we cannot see what it actually does.

The goal of all benchmarks in this thesis, is to provide a fair comparison between convolution algorithms using the GPU and the CPU. The code will be used in a specific real-world setting, repeatedly run from a GUI. It is therefore important to replicate a similar setting for all benchmarks, to be able to compare them.

In [TLM], AccelerEyes provides a set of recommendations to run Jacket. One of these recommendations is the warming up of the code. If this warmup is not performed, the delay of the first call is in the range of 10-13 seconds. It is therefore recommended to run the function to benchmark, once, on a small input size before actually running it. Another recommendation is to change the Matlab priority from the task manager to "real time". We did not need to use `gsync`, as stated in [TLM], we return the result with `single`, and we also verify that Jacket computes the correct value (not included in the code here).

`Timeit` is a function which takes a function as an argument. `Timeit` provides an easy way to time GPU, and CPU code by running the argument in a loop and returning the mean elapsed time. In this way we are given a fair timing according to [Vis11]. However, In our application, we will not run the code in a loop. Additionally, as indicated in the example benchmarks, and most probably in the Jacket benchmarks, the sending and returning of data from the GPU is not included in the comparisons. The sending and returning of GPU data is important in our application.

We found that the Jacket routines were unstable. We tried on both Matlab 2011b and Matlab 2012a. During the trial run of Jacket, we encountered propagated errors from CUDA including:

1. `CUFFT failure (invalid plan)`
2. `Encountered locked FFT cache`
3. `CUDA runtime error: unknown error`
4. `An existing CUDA context was found. Aborting initialization.`

These errors are manifested through Figure 5.14. A total of 7 tests were run to get the results. For some reason, Jacket could not free any memory if the memory ran out. To be able to run the tests again, we had to restart Matlab to free the memory. Each run was saved in a separate file. We saved the Matlab version and the sizes that were successfully computed. As seen in Listing 26 we also timed sending data to GPU (call to `gsingle`) and to CPU (call to `single`), this was included in the figure.

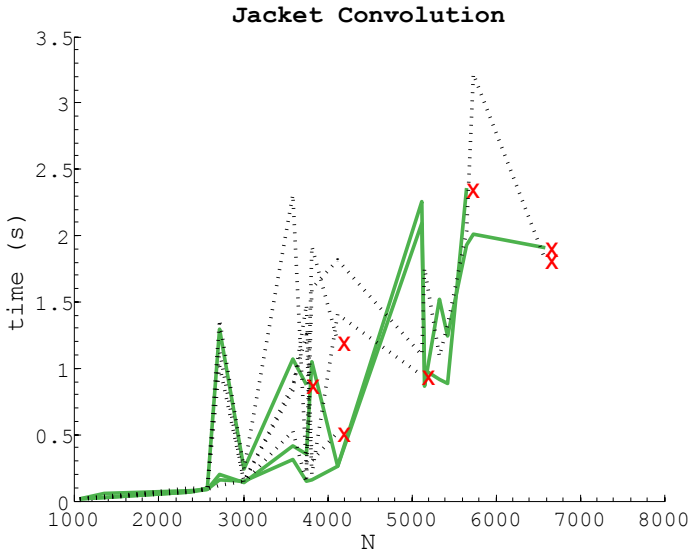


Figure 5.14: Convolution using Jacket. The crosses mark failed execution runs. We performed in total 7 trials. The dotted lines show runs using Matlab 2012a and the continuous lines Matlab 2011b.

Listing 26: Matlab using Jacket

```
S = mfilename('fullpath'); S = S(1:end-length(mfilename));
gcache('load', [S 'cache.jkt']);

ag = grand(2, 2);
bg = grand(2, 2);

single( circ_conv(ag, bg) ); %warmup

gsync %wait for GPU

tic;
ag = gsingle(A);
bg = gsingle(B);

ag = conv(ag, bg);
C = single( ag );
toc

%convolution function
function out = conv(a, b)

a = fft2(a);
b = fft2(b);
a = a .* b;

out = ifft2(a);

end
```

Benchmarks that are available on the AccelerEyes homepage [Acc] comparing GPU (with Jacket) and CPU are also possibly not taking into account the time of sending the data to the GPU. The benchmark comparison is also run on an Intel 920, 2.66GHz 4 (released Q4 2008) with a Tesla C2050 (Nov 2010) — a “high end” GPU. The GPU comes with 3GB of RAM and gives the comparisons towards Jacket a possibly unfair advantage.

5.3.6 gpuMat

The gpuMat library [Gpu] is built on CUDA 4.2 and allows us to call CUDAs FFT routine directly from Matlab without explicitly interfacing MEX. The library is open source and consists of functions to interface CUDA using MEX.

GpuMat makes it possible to code with an API similar to CUDA directly in Matlab without the involvement of MEX functions or kernel coding. The CUFFT routines also make it possible, just like with CUFFT, to create your own plan and execute them. This makes gpuMat the more flexible alternative on the market. Jacket and gpuArray both hide the calls to CUDA, which makes them simpler to use, but at the same time very inflexible.

While testing gpumat we found that it failed to complete computation for all but small convolution sizes. The returning error is memory allocation error.

5.3.7 Hybrid

Because of the computational power of the CPU, we considered using the CPU and the GPU to compute the convolution. Here, we deduce a simple model to predict the potential speedup by delegating the computations between the CPU and GPU using FFT_{11}.

When we send computations to the GPU, we usually wait for the result to return and send the rest of the data, hence leaving the CPU idle for the duration of the computation. A way to make use of the CPU during this time, to speed up the computations, could be to allocate memory, transpose data between passes and compute FFT alongside the GPU.

To make sure we do not slow down the overall computation, we need to divide the number of tasks performed by the CPU and GPU so they finish at the same time. We can express this condition by the following equation

$$n_{rows} \cdot t_{GPU} = (N - n_{rows}) \cdot t_{CPU} \tag{22}$$

n_{rows} contains the number of rows to be computed by the GPU. Variables t_{GPU} and t_{CPU} contain the time it takes to complete a fixed set of FFT computation on the GPU and the CPU respectively. N is the number of columns of the input matrix ($M = N$). In this simple model, we assume that the number of rows sent to the device is optimal, so the communication between the device and host is minimal. We can thus estimate the number of rows as

$$n_{rows} = \frac{N}{1 + t_{GPU}/t_{CPU}}.$$

The speedup gained from using this method depends on the quotient t_{GPU}/t_{CPU}. For example, say $t_{GPU} = 0.2s$ and $t_{CPU} = 1$ s. If we compute the full problem on the GPU, we have $8192 \cdot 0.2 \approx 1638s$. If we do the same, divided between the CPU and the GPU, we get $n_{rows} = 8192/(1 + 0.2/5) \approx 6827$. We now have $6827 \cdot 0.2 \approx 1365$ (same time on the CPU because of Equation 22). The factor speedup compared to only the GPU is thus $1638/1365 \approx 1.2$, which is simply the quotient t_{GPU}/t_{CPU}. We gain a 20% speedup compared to using only the GPU.

The method of distributing a problem over GPUs and the CPUs are only efficient if both are about the same speed. We saw in Figures 5.9 and 5.13 that the speed to perform convolution of a 8000×8000 takes 5 s and 8 s respectively. We should, according to the above argument,

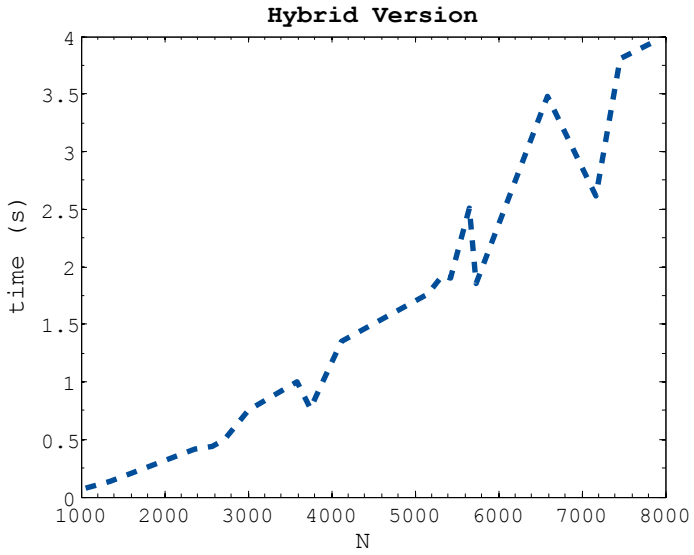


Figure 5.15: Convolution with the GPU/CPU hybrid version.

gain a 60% speedup using the hybrid algorithm. We propose the code in Listing 28 together with 27.

Since Matlab 2012a, asynchronous calls were introduced to gpuArray. However, this was not available for us, because of a limitation with the licensing of Matlab. We can thus only give an *estimated* performance. We do this by simply halving the timing for the hybrid method. The performance can be seen in Figure 5.15.

Listing 27: Benchmarking CPU and GPU

```
function [t_gpu, t_cpu] = benchmark(tmp, xs)
  test_work = min(1000, xs);%reasonable benchmark size

  tic;
  tmp(:, 1:test_work) = fft(A(:, 1:test_work));
  t_cpu = toc;

  t = tic();
  tmp(:, 1:test_work) = gather( fft( gpuArray(A(:, 1:test_work)) ) );
  t_gpu = gtoc(t);
end
```

Listing 28: FFT11 on gpuArray sending blocks of data

```
function A = hybrid(A, B, hadamard, bypass_benchmark)
mem       = 230; %MB
[ys, xs] = size(A);

%Benchmarking hybrid method:
%Assume GPU and CPU takes same time for larger sizes
t_gpu = 1; t_cpu = 1;

if ~bypass_benchmark [t_gpu, t_cpu] = benchmark(A, xs); end

gpu_work = ceil( xs/(1 + t_gpu/t_cpu) ); %columns on GPU
cpu_work = xs - gpu_work; %columns on CPU

%Block size used to send to GPU
bs = floor(2^24/ys); %seems like a safe block size
%special case
if bs > xs, bs = xs; end

for i=1:2
  if i==2, A = A.';
  A(:, 1:gpu_work)      = subfft(A(:, 1:gpu_work), bs);
  A(:, gpu_work+1:end) = fft(A(:, gpu_work+1:end));
  barriersynch(); %wait for GPU
end

for i=1:2
  if i==2, B = B.';
  B(:, 1:gpu_work)      = subfft(B(:, 1:gpu_work), bs);
  B(:, gpu_work+1:end) = fft(B(:, gpu_work+1:end));
  barriersynch(); %wait for GPU
end

A = A .* B; %Hadamard calculated on host

for i=1:2
  if i==2, A = A.';
  A(:, 1:gpu_work)      = subifft(A(:, 1:gpu_work), bs);
  A(:, gpu_work+1:end) = ifft(A(:, gpu_work+1:end));
  barriersynch();
end

end
```

GpuArray `fft` internally use CUFFT. To be able to see more of a speedup from the hybrid method, we could use CUDA to compute larger transforms than gpuArray. As we mentioned previously in Section 5.2.4, gpuArray has an unknown memory management, as discussed in section 5.2.4, this severely limits the input size and speed of the routines.

There are alternatives to programming using both GPU and CPU called Thrust [Thr]. Thrust is said to be similar to C++ STL (Standard Template Library) thus provides with a complete library which lets you code using OpenMP and CUDA in a simple manner.

We have shown a simple method of using both GPU and CPU within Matlab. With this method we saw a way to speed up convolution using CPU and GPU if the performance are about the same.

In the next section we will superimpose all the previous methods, the fastest from each category. For Jacket we have chosen the fastest runs of all tests, although it crashed for many of them.

5.4 COMPARING ALL THE METHODS

In this section, we will compare all the aforementioned methods of convolution. Figure 5.16 shows the fastest performances of the convolution algorithms on the CPU, the GPU (gpuArray, Jacket) and the hybrid method.

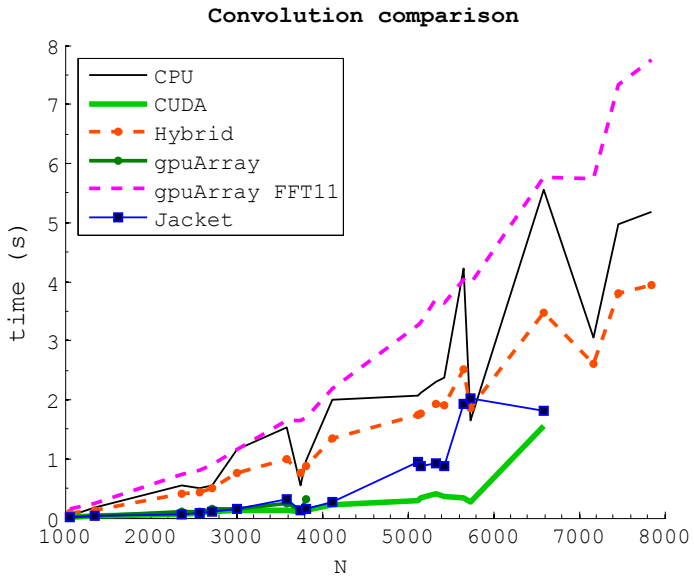


Figure 5.16: Final comparison.

In this comparison, we first need to make it clear that we compare between methods of convolution using the simple convolution algorithm as before in this section. In VISSLA™, we do it slightly differently, using 3 channels for the image and only one for the kernel. To get a fair comparison between the CUDA code and the CPU code we have had to add extra complexity to our CUDA code, which we decided not to do because of time constraints. However, the relative difference in speed between the methods are still valid, but the speedups are more modest than indicated, as we will see in Figure 5.17. This table will work as a schematic layout of the speedup between CUDA and the real VISSLA™ convolution algorithm on the CPU. The input sizes indicated in the table will also give the smallest CUFFT buffers.

Our comparison plot features Jacket although it did not pass our tests. The values in the plot features the fastest runs out of 7 trials of Jacket. We can see that our CUDA code is faster for all input sizes compared to Jacket and gpuArray.

We tested several versions of convolution using gpuArray. The main issue with gpuArray is the memory management, which results in crashes, similar to Jacket. GpuArray using FFT_{11} manages to convolve larger matrices but at the cost of sending the data several times to and from the GPU. At some point, this method is even slower than the CPU.

Our proposed *hybrid* method was only marginally faster than the CPU due to the limited memory allocation of gpuArray. By using CUDA instead, we could have a faster method. Unfortunately because of Equation 22, we would only expect a modest speedup compared to only using CUDA. Another way to speed up the method is by using several identical GPUs.

We also saw that our proposed CUDA code is faster than Jacket and gpuArray, and can convolve just as large images.

5.5 CUDA CODE ON VISSLA

In the previous section we found that our CUDA code was superior to other alternatives including the CPU.

In this section we will compare our CUDA code to the CPU code used in VISSLA™. The computation is similar to the previous section, but differs in that the image consist of three channels. This difference should still yield a speedup comparable to the aforementioned methods, but we only use one channel in the kernel. This is an important difference because the speedup is not quite as large as the above comparison might indicate.

The CPU code we used in this comparison can be seen in Listing 29. The CUDA code is identical. The complete CUDA code can be seen in Section 9.

Listing 29: VISSLA™ CPU convolution

```
function z = ConvFFT(g, h)
%g is the 3-channel padded image, h is the 1-channel full kernel
[gy, gx, ~] = size(g);
ffth = fft2(h);
%This differs from the CUDA code, which, for simplicity, calculates the ...
    kernel three times

for i = 1:3
  z(:, :, i) = ifft2(fft2(g(:, :, i)).*ffth);
end

z = fftshift(z);
z = z(1:gy, 1:gx, :);
```

In order to show the speedup, we chose to include not only quadratic sizes, but all of the different sizes as seen in Table 5.4. The resulting figure (Figure 5.17) consists of numbers indicating: memory requirement, size (Mega pixels) and speedup for the different input sizes. The information is enclosed in cells in shades of green. These cells work as a map of the speedup. Lighter shades indicate a larger speedup, and dark shades means slower. The plan sizes are also indicated using contour lines.

The reason we are getting different speedups for different sizes is because FFTW can transform input sizes with small factors faster than others. The sizes with large factors (or worse, if the size is prime) are the slowest. Because we want small CUFFT buffer sizes, and these seem to not depend on the factors as FFTW does, we get different speedups for different sizes.

To get an idea of how fast the CUDA code really is, we picked the size that produces the largest speedup: 5661×5661. This size has the factors 3, 17 and 37, so it is not obvious as to why we get such a large speedup. On the CPU (via VISSLA™) it took 10.5 s compared to 2.42 s using our CUDA code. If we look at our earlier experiment convolving matrices using the CPU on page 47, the performance in Figure 5.9 show that for some input sizes, the convolution algorithm is very slow. This concludes that the speedup is clearly the case of FFTW being slow, rather than the GPU is just much faster.

Also note that attempting to perform convolution using all GPU memory is not recommended. CUDA seems to slow down just before the GPU is full. We can see a dip in speedup when the available memory on the GPU is at 85%.

Note that the plan memory contours are wavy, instead of smooth curves, because the cells are not equidistant. Also note that, according to the poster, the speedups are *almost* symmetric. Generally,

but not always, the input sizes where the dimensions where $N > M$ (under the dashed *symmetry line*) are faster than the cells where $M > N$ (over the dashed line). The speedup between the halves could be up to 40%.

5.6 FURTHER SPEEDUPS

To speed up the code further, we will show an estimated speedup using two rather simple optimisations. We will use as an example a matrix of size 4750×4750. This sized input takes 1.67 s to convolve using the CUDA code. The image require $3 \cdot 86$ MB of memory, the same as the kernel. In total, the image and kernel requires 516 MB of memory. We also need to send the data back, which is $3 \cdot 86$ MB. In total 774 MB is transmitted. Assuming the speed of transfer is 3047 MB/s (see Section 5.2.5 for details), the transfer takes 0.25 s for all data, both input and output.

We will use the VISSLA™ CPU code as a guide and only compute the kernel once. The implications are that we only need to send the kernel once. That is, we only need to send $4 \cdot 86$ MB (to GPU) + $3 \cdot 86$ (from GPU), totalling to 602 MB, which takes 0.197 s to send.

But, we also only need to compute the kernel once. In order to estimate the speedup, we assume that `ifft` and `fft` takes about the same time and the Hadamard product takes a negligible amount of time. We then remove the transfer time of our proposed CUDA code, which is 1.67-0.25 = 1.42 s. We estimate each `fft` by dividing with 9 and multiplying with 7 (our new number of `fft`s). The new computation takes 1.11 s. The new optimization takes 1.11 (7 `fft`) + 0.197 (transfer) = 1.29 s which is 28% faster than before.

We can also pad using the GPU. In this case, the only part of the computation that is affected is the transfer. We get down the transfer to 0.134 s. The speedup is now 35%.

It is difficult to speed up the computation part of the code any more than we suggested, but, transfer could be reduced even more. For instance, if the kernel is created using a simple function (e.g., Gaußian function) we could send only the parameters to create the function and create it on the GPU.

It is also possible to use streams to be able to parallelise tasks that could be performed in parallel [Nvi, chap. 3.2.5]. This is reportedly important when using pinned memory.

Additionally, if we could *asynchronously* send data, we could transform blocks of data at the same time as we send the data, thus hiding the memory transfer within calculations.

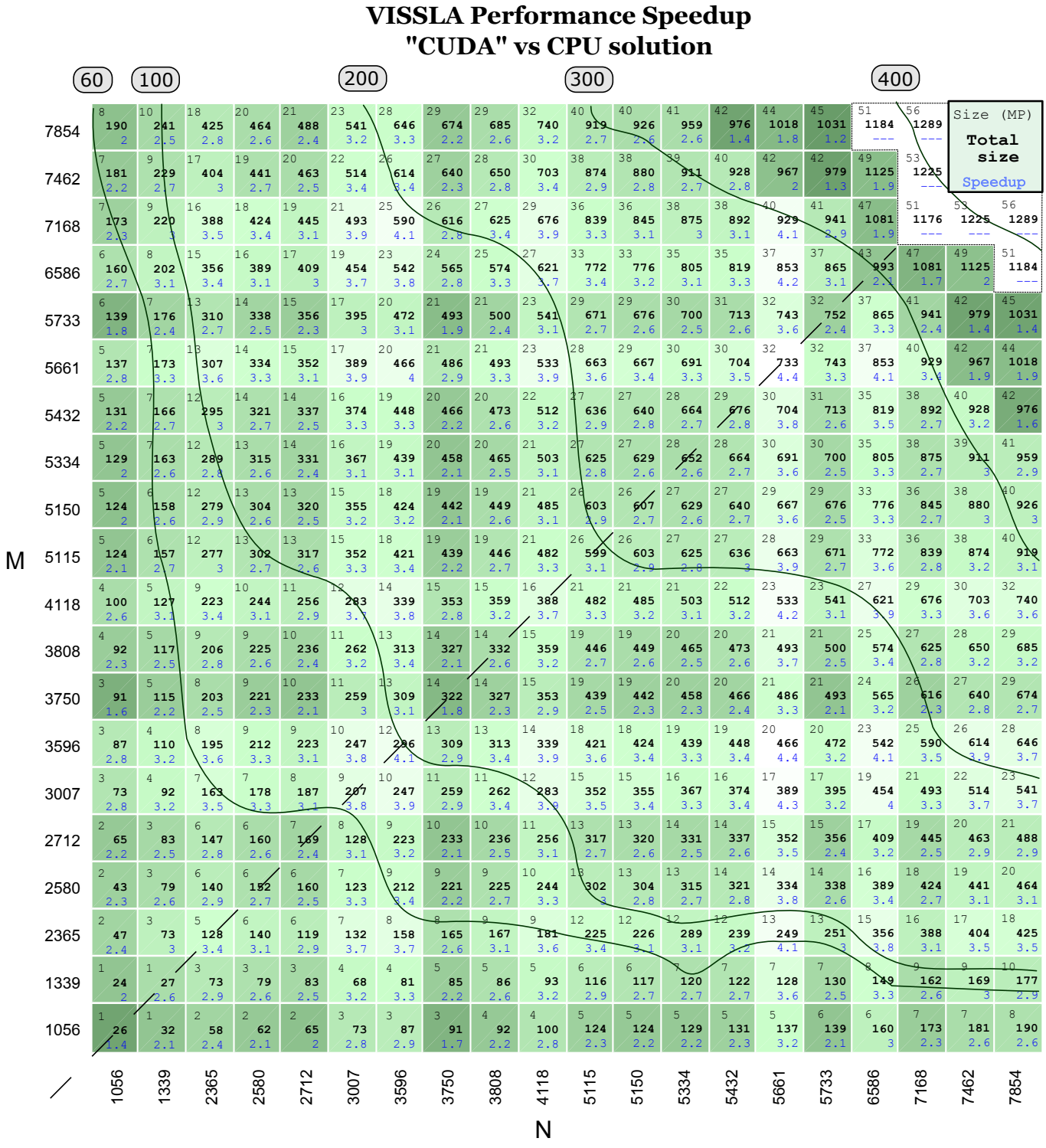


Figure 5.17: A table of the speedup of CUDA Convolution algorithm compared to the VISSLA™ CPU code. Speedup is indicated using numbers and brightness of the cells. The sizes on the axes are not equidistant.

SECTION 6

Discussion and Conclusions

We looked at some CPU and GPU libraries to convolve matrices. We concluded that using the GPU to convolve large matrices provides a (modest) speedup, compared to using the CPU. In VISSLA™ we saw a speedup of 3-4 times compared to using the CPU.

However, our proposed solution could be sped up more, by computing the kernel once and pad the data on the GPU. These optimisations was estimated to lead to a speedup of at least 35%. Hopefully, a real implementation would provide an even better speedup than we predict.

In our comparisons we used a mid-range, balanced system, to compare the performance between the CPU and the GPU. We bought the system in March 2012 and it cost approximately 10000 Swedish kronor (approximately $1500). This provided us with a GPU with 1250 MB on-board memory, as stated in Section 5. We conclude that the recommended sizes to convolve using such a GPU is stated in Figure 5.17 as no more than 85% of the available free memory.

Associated with memory usage is the stability of our CUDA code, this applies to the other alternatives also. By avoiding to allocate more memory than is available, we eliminated the "memory crashing issue".

To speed up the code even more, we saw that it is possible to refine the algorithms and only use FFT_{11} on the GPU, as mentioned in [DKE10]. We believe this would substantially reduce the memory required by the CUFFT plan. If we want to convolve even larger images, a way to provide faster transfers is to use so called "pinned memory" or page-locked memory [Nvi] feature. This will however require more host-side memory. However, because of the MEX interface, we are forced to use `mxMalloc`, and there are no way to call `cudaMallocHost` to provide pinned memory transfers for the current version of Matlab (2012a).

GpuArray proved to be a simple way to call functions and send them to the GPU, without the need to use MEX to interface to CUDA. One problem is the unpredictable memory management used by gpuArray. The speed of the transfer is also lacking. We believe the "pinned memory" option should be available within gpuArray.

CUDA coding still lacks a stable debugging environment. We believe a stand-alone debugger and a simple profiler would help greatly in bringing in programmers to use Windows as a platform for CUDA.

We tried using Visual Studio and nSight, along with Visual Profiler. At some point they would crash the computer, or not be able to profile Matlab code correctly. In Unix, there are tools to debug CUDA. Linking and compiling is also much easier to do on the Unix platform.

We also examined a CPU-only methods of convolving matrices using the row-column method. We provided with a solution (although we pad the kernel) with a speedup of 2.7 times, using the row-column method.

An alternative way to speed up convolutions is to change the distribution of the workload in the GUI. By computing the FFT of the image as soon as the image is loaded from disk, we only need to calculate the FFT of the kernel. The kernel cannot be transformed until modifications to it has been made.

We can also note that possibly performing convolution on a part of the image would be possible using the *overlap/save* and *overlap/add* methods [Smi97].

Timing code and showing performance can be deceptive. The fact that Matlab is doing some optimisations to accelerate code can also affect the results. The best way to check a result is to test the code separately for a few cases, to ensure integrity of the results. For the convolution benchmarks we instead ran the different tests separately and saved the results on disk, to reduce contaminating GPU contexts.

Benchmarks are difficult to code, because we do not really know in what way the code is to be used. Do we use it repeatedly, accounting for the "warm up", or pause between runs? Many commercial codes tries to use benchmarks as a tool to provide the customer with proof that their product is faster. We have discussed controversies, for instance Bernstein versus Frigo and Johnson (see Section 2), Jacket (see Section 5.3.5), gpuBench (see Section 5.2.3). For those interested, see [VWLN10] and [GH11].

We believe that the CUFFT library is a very nice API to enable powerful math operations on the GPU. However, additional functions would make the API even more useful, such as a fast transpose of `cufftcomplex` data structure. Additionally, we wish for a kernel which untangles the output of the `R2R` transform. These additions would make FFT_{11} using CUDA much simpler to implement.

To answer the Problem statement point 3 in Section 1.4, *is our research going to be relevant for the next five years?*. This is a difficult question to answer. The speedup we showed indicated that the GPU is at least 4 times as fast as the CPU. We believe that the number of cores on the GPU will increase, hence making the convolution computation more viable in the future. Additionally, more on-board memory and speedier cores will only give the extra edge the GPU needs in order to outperform the CPU by several orders of magnitudes.

SECTION 7

Future Work

One concern about using FFT to apply convolution is the assumption that the input image is periodic. To be able to convolve a non-periodic image using FFT, we padded our image with zeros (section 3.1.2). Using a panorama image, we map a 360 degree image on a cylinder. This would make the horizontal vector periodic. One problem with this approach is that we still need to pad the vertical dimension.

Additional speedup, except from using "Real to Complex" transform could be accomplished using proper FFT planning. Since FFTW planning in C takes a relatively long time and could increase the performance of CPU FFT. A way to speed up the plan creation could be to store all the possible plans VISSLA™ could transform. This would possibly speed up the convolution even more.

Some methods of convolution were not considered, such as *overlap/save* and *overlap/add* [Smi97]. These methods are similar to FFT_{11} but differs in that they perform convolution of parts of the image using the full kernel, which is useful for real time convolution. This method might be useful if we only are interested in a part of the convolution, such as zooming into a part of a large image.

We intend to continue with our research with padding and creating the kernel on the GPU. Additionally, it would be interesting to see how FFT_{11} on CUDA would eliminate plan size for the benefit of convolving larger matrices. It is also interesting to see how much it affects performance, if at all.

SECTION 8

Appendix

This section will describe a couple of the errors we encountered, and how to fix them. We start with the different problems encountered while creating our CUDA code and then follow with general tips on how to setup the Mex and CUDA environment.

8.1 PROBLEMS ENCOUNTERED, DEBUGGING CUDA

The following bugs were difficult to find and may help others when first coding in CUDA. I spent many hours trying to fix these bugs. Many of the bugs were trivial and are left out of this list. Some bugs in the list below may seem trivial, but they were very difficult to find.

We also indicate the severity of the error, when available. Sometimes, however, we found that CUDA reports the error (often unspecified) *after* it has occurred. CUDA (or Matlab) seems to stack the errors and report old errors at times.

Sometimes Matlab is forced to restart. That happened often, and is very time consuming and forced us to change the way we code. We tried Visual Studio with *n*Vidia nSight. While we tried out nSight, the computer system rebooted twice within a period of 10 minutes. We abandoned nSight.

We also tried to use *n*Vidia Visual Profiler (VP), shipping with CUDA (called `nvvp`), to profile our code. We found examples on how to call Matlab from VP, but after a couple of tries, we abandoned this idea. VPS runs the code several times and will give a report on the performance, but for some reason the Matlab instances would not exit Matlab and just bogged down the system for some reason.

We used the following flags inside VP:

```
-wait -nojvm -nosplash -r script, exit
```

where `script` is the script name without the m suffix.

The first version of our code was compiled and run i VS. We gained lots of experience by doing this. We even got the executable file profiled by nSight at one time.

After we got a debugged core code and we felt confident enough to add code to it, we changed from VS to using only Matlab (which has no debugger).

This forced us to change our way of coding. Instead of employing a quick “test and see” coding, as is the way of coding we always use, we changed to coding “slow and careful”.

We found that the following errors we made are difficult to spot some times.

1. Freeing twice causes crash
2. Using malloc and not mxmalloc
3. Cudamalloc does not return NULL if it succeed
4. Block size must be correct
5. Uninitialized thread variable
6. Changing all the code from double to float is important

7. If dimensions of plan are swapped (should be N, M), the result will look like "clipping"
8. Random crashing

Point 1 was solved by creating a function "release" which hade it impossible to free a pointer twice. In this way we could place a Error label at the end of our code, which takes care of every error. If we encounter an error early on in the code the error label frees memory which has not been used. By using this release function we are sure it will not leave CUDA in another "error state".

Point 2 This took time to notice but is very logical. `mxMalloc` is used by Matlab and the Mex interface and should not be used as device allocated code. This will most definitely cause a crash.

Point 3 Was also very difficult to notice, until we looked carefully in the CUDA API Reference Manual [Cuda, Chap. 5.28.3.3]. We are used to coding in C where a successful allocation of a pointer returns a non-NULL address. If we compare the return from cudaMalloc, NULL is mapped as 0 and thus the first element in the enumeration cudaresult, which is `OK`.

Point 4 Block size must be carefully calculated. This is pretty simple, but the way to compute it was not straight-forward, until we found a good source.

Point 5 Uninitialized thread variable could possibly crash the code, since we index within a matrix. The way to code in CUDA is different from the ordinary for-loop structure, so you tend to forget that you actually have to initialize the index variable.

Point 6 Changing types is very important. When we started with CUDA and FFTW, the example codes were in double precision. We only use single precision, therefore we had to change all the allocation sizes (sizeof) and casting to `single`. This will take time but could pass compilation stage, that is why we added this here.

Point 7 As a part of the construction of our CUDA Convolution code, we decided to add the non-quadratic sizes. We noticed that the data is transposed when it is sent from Matlab to MEX (and subsequently CUDA). Therefore we needed to create a plan of $N \times M$ (instead of $M \times N$). Since we created a $M \times N$ plan, we got a convolution which looked like a clipping artifact.

Point 8 Random crashes are *very* difficult to debug. One tip is to make sure that the kernel calls are synchronised by simply calling `cudaThreadSynchronize()`.

Additionally, error messages from CUDA, reported by Matlab will often not make any sense. One of these are the following:

```
 An unexpected error occurred during CUDA execution.
The CUDA error was:  setting the device when a process
is active is not allowed.
```

We found that this problem could be fixed by resetting the CUDA context. It is enough to call `cudaDeviceReset()`.

8.2 SOME STEPS ON THE ROAD TO CUDA/MATLAB INTEGRATION

First, you need to install the VS (Visual Studio) to be able to compile C code. Unfortunately, the Express version of VS (which is free) does not contain the 64 bit compilers we need. So, we have to download the Windows SDK 7.1 also[3].

[3] http://www.mathworks.se/support/solutions/en/data/1-ECUGQX/

When these are installed, we need to tell Matlab where they are, to be able to compile our codes.

Type in the Matlab prompt

```
mex -setup
```

This will start a text wizard that will guide you to choose the compiler you want. We need to choose Windows SDK 7.1 as the compiler.

The setup script will create a batch file called `mexopts.bat` (Mex options). This file contains variables and program calls to the compiler cl.exe and linker link.exe which we need to compile Mex code.

We need to correct `VSINSTALLDIR` to point to the path where Visual Studio is, if necessary.

Tip: Permanently set Matlab to be "run as administrator". Otherwise we will get error messages when we try to compile our code. Right click on the Matlab shortcut, choose "Compatibility" and tick the "Run as Administrator" radio button.

Additionally, we need to update the linker directory (`LINKERDIR`) to point to the Windows SDK. Note: We need to point to Windows SDK 7.1, 7.0 does not work.

Also note that the paths in `mexopts` must *not* have space between variable and path, so:

```
LINKERDIR= C:
```

is not the same as

```
LINKERDIR=C:
```

We also need to make sure the *nvcc.profile* in CUDA toolbox contains path information for the CUDA compiler, we need to point this to Visual Studios `include`

```
INCLUDES += ``-I$(TOP)/include" "-I$(TOP)/include/cudart''
  ``-IC:/Program Files (x86)/Microsoft Visual Studio 10.0/VC/include''
  $(_SPACE_)
```

We need to modify the file `mex.pl` (perl code) in `Matlab/R2011b/bin/`. Right before `$ENV{'ARCH'} = $ARCH;` add `$ARCH = ``win64'';`, forcing a 64 bit compilation.

Hopefully this short section will provide with enough help to get the reader "on the road" to code in CUDA and call it from Matlab.

SECTION 9

Codes

This section contains the codes used to convolve on the GPU. The first code is the C/C++ code bridging Matlab via Mex. We will include all the files that were used, so readers could learn and try them out for themselves. We use C++ because at some time, we needed to compile C++ to be able to get a full set of compiler optimisations, the code could just as easily compile with a C compiler. No special C++ specific keywords are used in these codes.

9.1 ERRCODES

This section contain the error code header and source file. This is important, especially when finding out bugs. However, in the final product, the error messages are very unlikely to occur.

9.1.1 Header

This is the header file for the errCodes.cpp.

```
void mexprintError(int x);

typedef enum {
  ERR_OK = 0,
  ERR_PLAN,
  ERR_CUFFT,
  ERR_FFT_FORWARD,
  ERR_FFT_INVERSE,
  ERR_MALLOC,
  ERR_COMPAT,
  ERR_FAILSAFE,
  ERR_COPY
} ERR_CODE;
```

9.1.2 Source

```
#include "errCodes.h"
#include "mex.h"


void mexprintError(int x)
{
  char * mess;

  switch( x )
    {
    case (enum ERROR_CODE) ERR_PLAN:
      mess = "The plan is wrong goddamnit\n"; break;
    case (enum ERROR_CODE) ERR_FFT_FORWARD:
      mess = "Forward FFT could not be executed.\n"; break;
```

```
    case (enum ERROR_CODE) ERR_FFT_INVERSE:
      mess = "Inverse FFT could not be executed.\n"; break;
    case (enum ERROR_CODE) ERR_MALLOC:
      mess = "Malloc was not successful.\n"; break;
    case (enum ERROR_CODE) ERR_COMPAT:
      mess = "Compatibility Mode was not set.\n"; break;
    case (enum ERROR_CODE) ERR_COPY:
      mess = "Copying data was not successful.\n"; break;
    case (enum ERROR_CODE) ERR_FAILSAFE:
      mess = "Listen, the code didn't run correctly, what can I tell ya...
          ...\n"; break;
      break;

    }

  if (x>0) //added
    mexErrMsgIdAndTxt( "MATLAB:mexcallmatlab:CUFFT", mess);
}
```

9.2 C CODE

The C++ code interfacing Mex and calling the CUDA routine.

9.2.1 Source

```
/*
Matz JB 2012
 This code peforms the convolution between two (MxNx1) matrices
 Required memory: 6 units
*/
#include <mex.h>
#include "errCodes.h"
#include "CConv.h"
#include <math.h>

void myExitFcn()
{
  mexPrintf("MEX-file is being unloaded\n");
}


void mexFunction( int nlhs, mxArray* plhs[], int nrhs, const mxArray* prhs...
    [] )
{

  int M, N;

  if (nlhs!=1)
    mexErrMsgTxt("The number of outputs should be 1\n");


  if(nrhs==0)
    {
      mexPrintf("Error: Provide with input data please\n");
      return;
    }

  if(nrhs!=2)
    {
```

```
      mexPrintf("Error: Provide two matrices please\n");
      return;
    }

  float * indata1;
  float * indata2;
  float * outdata;
  int tmpM, tmpN;


  tmpM = (int)mxGetM(prhs[0]);
  tmpN = (int)mxGetN(prhs[0]);

  M = (int)mxGetM(prhs[1]);
  N = (int)mxGetN(prhs[1]);

  if (M!=tmpM)
    mexErrMsgTxt("The number of rows in the inputs must match.\n");

  if (N!=tmpN)
    mexErrMsgTxt("The number of columns in the inputs must match.\n");

  if (DEBUG)
    mexPrintf("Dimension of inputs: (%dx%d)\n", M, N);

  if (nrhs != 2)
    mexErrMsgTxt("Hello, I want 2 inputs.\n");
  else if (nlhs != 1)
    mexErrMsgTxt("I want a single output.\n");

  indata1 = (float*) mxGetPr(prhs[0]);
  indata2 = (float*) mxGetPr(prhs[1]);

  plhs[0] = mxCreateNumericMatrix(M, N, mxSINGLE_CLASS, mxCOMPLEX);
  outdata = (float*) mxGetPr(plhs[0]); //must cast

  int err;

  //Launching kernel
  Convolution(indata1, indata2, outdata, M, N, &err);

  mexprintError(err); //if an error occured, we pass it on to Matlab

  if(mexAtExit(myExitFcn))
    {
      mexPrintf("Error unloading function!\n");
    }

}
```

9.3 CUDA CODE

This section contains the CUDA code. Note: change the macro `NR_THREADS` depending on the graphics hardware the code is run on.

9.3.1 Header

The header was tricky to get right. Omitting the `extern C` keyword will result in an error. Trial and error directed us here.

```
extern "C" void Convolution(float *a, float *b, float *c, int M, int N, ...
    int *err);
```

9.3.2 Source

```
//Matz JB June 2012

//This code performs the convolution between two images
//IFFT( FFT2(A).*FFT2(B) ) where A:(MxN), B:(MxN)
//Memory requirement: 6 units

#include <stdio.h>
#include <math.h>
#include <cufft.h>
#include "errCodes.h"
//errCodes contains the hidden error message to mexprint

#define MAX_THREADS 1024 //change as appropriate


void sync()
{
  cudaThreadSynchronize();
}

//This is only here for historical reasons. Freeing a variable in CUDA
//which has not been allocated seems to cause no problems. In C/Mex,
//this will cause a crash
void cudaRelease(void *ptr)
{
  if ( ptr != NULL )
    cudaFree(ptr);
}

//Quad indexing using threading from:
//http://gpgpu.org/wp/wp-content/uploads/2009/06/03-Toolkit.pdf We can
//get non-quad matrix indexing using idx and idy

__global__ void weavecomplex (cufftComplex *c, float *a, int M, int N)
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  int idy = blockIdx.y*blockDim.y + threadIdx.y;

  if(idx<M && idy<N)
    {
      int index = idx + idy*M;
      c[index].x  = a[index];
      c[index].y  = 0.f;
    }
}

//We only need to unweave to real
__global__ void unweavecomplex2R (float *a, cufftComplex *c, int M, int N)
{

  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  int idy = blockIdx.y*blockDim.y + threadIdx.y;
  volatile float2 c2;//force vector load, increase memory coalescing
```

```
  if(idx<M && idy<N)
    {
      int index = idx + idy*M;

      c2.x = c[index].x;
      c2.y = c[index].y;
      a[index] = c2.x;
    }
}

//Scaling is embedded in Hadamard product instead of inside the
//"weaving" functions
__global__ void hadamard(cufftComplex * a, cufftComplex * b, int M, int N)
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  int idy = blockIdx.y*blockDim.y + threadIdx.y;
  float scaling = 1.0f/sqrt(1.0f*M*N);
  int index;
  volatile float tmp;

  if(idx<M && idy<N)
    {
      index = idx + idy*M;

      a[index].x *= scaling;
      a[index].y *= scaling;
      b[index].x *= scaling;
      b[index].y *= scaling;

      tmp = a[index].x;

      //Naive complex multiplication, 2 additions + 4 multiplications
      a[index].x = tmp*b[index].x - a[index].y*b[index].y;
      a[index].y = tmp*b[index].y + a[index].y*b[index].x;
    }
}


//Convolves a and b and stores the result in c
extern "C" void Convolution(float *a, float *b, float *c, int M, int N, ...
    int *err)
{
  //Device data, only used on the device:
  //These declarations must be first since we risk to encounter errors for ...
      which we
  //just goto "Error"
  cufftComplex *rhs_complex_d1 = NULL;
  cufftComplex *rhs_complex_d2 = NULL;

  float *a_d = NULL;

  //Setting up Block and Grids for the thread mappping:
  int block_size_x = (int) sqrt( (float) MAX_THREADS);
  int block_size_y = block_size_x;

  dim3 dimBlock(block_size_x, block_size_y, 1);
  dim3 dimGrid((M/dimBlock.x), (N/dimBlock.y));

  if (M % block_size_x !=0)
    dimGrid.x += 1;

  if (N % block_size_y !=0)
    dimGrid.y += 1;
```

```
cufftHandle plan;
*err = ERR_FAILSAFE;

//1 unit
if( cudaMalloc((void **) &a_d, sizeof(float)*M*N) != cudaSuccess )
  {
    *err = ERR_MALLOC;
    goto Error;
  }

if( cudaMemcpy(a_d, a, sizeof(float)*M*N, cudaMemcpyHostToDevice) != ...
    cudaSuccess )
  {
    *err = ERR_COPY;
    goto Error;
  }

//3 units
if( cudaMalloc((void **) &rhs_complex_d1, sizeof(cufftComplex)*M*N) != ...
    cudaSuccess )
  {
    *err = ERR_MALLOC;
    goto Error;
  }

weavecomplex<<<dimGrid, dimBlock>>>(rhs_complex_d1, a_d, M, N);

//5 units
if( cudaMalloc((void **) &rhs_complex_d2, sizeof(cufftComplex)*M*N) != ...
    cudaSuccess )
  {
    *err = ERR_MALLOC;
    goto Error;
  }

//reuse a_d
if( cudaMemcpy(a_d, b, sizeof(float)*M*N, cudaMemcpyHostToDevice) != ...
    cudaSuccess )
  {
    *err = ERR_COPY;
    goto Error;
  }

weavecomplex<<<dimGrid, dimBlock>>>(rhs_complex_d2, a_d, M, N);
sync();//wait for kernel

//4 units
cudaRelease(a_d);

//At least 6 units. Notice the order of the 2:nd and 3:rd arguments
if (cufftPlan2d(&plan, N, M, CUFFT_C2C) != CUFFT_SUCCESS)
  {
    *err = ERR_PLAN;
    goto Error;
  }

if (cufftSetCompatibilityMode(plan, CUFFT_COMPATIBILITY_NATIVE) != ...
    CUFFT_SUCCESS)
  {
    *err = ERR_COMPAT;
    goto Error;
  }
```

```
  if (cufftExecC2C(plan, rhs_complex_d1, rhs_complex_d1, CUFFT_FORWARD) != ...
      CUFFT_SUCCESS)
    {
      *err = ERR_FFT_FORWARD;
      goto Error;
    }

  //same plan to perform FFT on the other matrix
  if (cufftExecC2C(plan, rhs_complex_d2, rhs_complex_d2, CUFFT_FORWARD) != ...
      CUFFT_SUCCESS)
    {
      *err = ERR_FFT_FORWARD;
      goto Error;
    }

  hadamard<<<dimGrid, dimBlock>>>(rhs_complex_d1, rhs_complex_d2, M, N);

  if (cufftExecC2C(plan, rhs_complex_d1, rhs_complex_d1, CUFFT_INVERSE) != ...
      CUFFT_SUCCESS)
    {
      *err = ERR_FFT_INVERSE;

      goto Error;
    }

  cudaRelease(rhs_complex_d2);

  if( cudaMalloc((void **) &a_d, sizeof(float)*M*N) != cudaSuccess )
    {
      *err = ERR_MALLOC;
      goto Error;
    }

  unweavecomplex2R<<<dimGrid, dimBlock>>>(a_d, rhs_complex_d1, M, N);

  //Pick only real part and send back to host code
  unweavecomplex2R<<<dimGrid, dimBlock>>>(a_d, rhs_complex_d1, M, N);

  cudaMemcpy(c, a_d, sizeof(float)*M*N, cudaMemcpyDeviceToHost);

  *err = ERR_OK;
  //We reached this point, thus everything went ok,
  //otherwise we have ERR_FAILSAFESAFE, which should never happen


  //Catch all error cases and clean up:
 Error:

  cudaRelease(a_d);
  cudaRelease(rhs_complex_d1);

  cufftDestroy(plan);
}
```

Glossary

*n*Vidia	*n*Vidia Corporation. Developer of GPU and CPU chipsets.
AccelerEyes	Accelereyes, the company that developed Jacket.
API	An Application Programming Interface (API) is a particular set of rules and specifications that a software program can follow to access and make use of the services and resources provided by another particular software program that implements that API.
FFT_{11}	An alternative algorithm to compute the FFT_2 in two passes. The method is commonly referred to as the "row-column method".
FFT_2	The 2-dimensional FFT.
gpuArray	MATLABs internal GPU data structure, lets a user use GPU accelerated functions.
gpuBench	Matlab GPU benchmark suite.
Jacket	A GPU library and interface to CUDA-accelerated code from Matlab.
nSight	nVidia profiler and debugger for Windows.
Octave	A programming environment, similar and easily portable to Matlab.
RAW	A file format usually used in professional digital cameras.
Relux	A open source modeller and rendering software specialised in light simulation.
Thrust	A library providing a simple API for GPU and CPU coding.
V-Ray	High performance photo realistic rendering plugin by Chaosgroup.

Acronyms

ALU	**A**rithmetic **L**ogic Unit. A digital circuit that performs arithmetic and logical operations.
API	**A**pplication **P**rogramming **I**nterface. See glossary item API.
CPU	**C**entral **P**rocessing **U**nit.
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
CUFFT	**CU**DA **FFT**, part of the CUDA toolbox.
DFT	**D**iscrete **F**ourier **T**ransform.
dll	**D**ynamic–**L**ink **L**ibrary, a Microsoft–specific version of shared libraries.
FFT	**F**ast **F**ourier **T**ransform.
FFTW	**F**astest **F**ourier **T**ransform in the **W**est, an FFT library written in C.
GCC	**GNU C**ompiler **C**ollection, a collection of compilers. `gcc` is the command that invokes the GCC C compiler.
GPU	**G**raphics **P**rocessing **U**nit.
GUI	**G**raphical **U**ser **I**nterface.
HDR	**H**igh **D**ynamic **R**ange Image file format (e.g., .HDR, .EXR).
JIT	**J**ust **i**n **T**ime, a compiler mode optimizing code just before executing.
JPEG	**J**oint **P**hotographic **E**xperts **G**roup (JPEG).
Matlab	**Mat**rix **lab**oratory is a scientific numerical computing and visualization environment.
MEX	**M**ATLAB **Ex**ecutable, a way to call functions written in C, C++ or Fortran directly from Matlab.
NVCC	**NV**IDIA **C**UDA **C**ompiler.
OpenCL	**Open C**omputing **L**anguage. A framework for writing programs executing on the GPU.
OpenMP	**Open** source **M**ulti-**P**rocessing.
PTX	**P**arallel **T**hread **E**xecution, a pseudo assembly language in CUDA.
SDK	**S**oftware **D**eveloper **K**it. A programming package that enables a programmer to develop applications for a specific platform.
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata. Hardware supporting execution of multiple processing elements with the same operation on multiple data simultaneously

SSE	**S**treaming **S**IMD **E**xtensions, is an instruction set basically allowing fast floating point vector operations. See glossary item SIMD
STL	**S**tandard **T**emplate **L**ibrary. A C++ software library providing a vast amount of algorithms and data structures.
VISSLA™	Software written in Matlab, abbreviation of **VIS**ualisation tool for **S**imulation of **L**ight scattering and **A**berrations
VS	**V**isual **S**tudio (VS), developer program created by Microsoft.

References

[Acc] *AccelerEyes home page*. URL: https://www.accelereyes.com/.

[AU+10] Shams A.H. Al Umairy et al. "On the Use of Small 2D Convolutions on GPUs". Anglais. In: *A4MMC 2010 - 1st Workshop on Applications for Multi and Many Core Processors*. Ed. by Ana Lucia Varbanescu, Rob van Nieuwpoort, and Anca Molnos. Saint Malo, France, 2010. URL: http://hal.inria.fr/inria-00493873.

[Ber] Daniel J. Bernstein. *The art of FFT benchmarking*. Accessed May 2012. URL: http://cr.yp.to/djbfft/bench-notes.html.

[Bru78] Georg Bruun. "Z-Transform DFT filters and FFTs". In: *IEEE Acoustics, Speech and Signal Processing.* (1978), pp. 56–63.

[Cat] *Vision problems in the U.S. report, developed by the national eye institute and prevent blindness America*. 2002. URL: http://www.preventblindness.net/site/DocServer/VPUS_2008_update.pdf.

[CG00] Eleanore Chu and Alan George. *Inside the FFT Black box: Serial and parallel fast fourier transform algorithms*. Accessed 16 May, 2012. CRC, 2000. URL: http://dsp-book.narod.ru/FFTBB/0270_PDF_C23.pdf.

[Col] *Column major storage in Matlab*. Acessed Sept 29, 2012. URL: http://www.mathworks.se/help/matlab/matlab_external/matlab-data.html#f17318.

[CT65] James W. Cooley and John W. Tukey. "An algorithm for the machine calculation of complex Fourier series". In: *Mathematics of Computation* (1965).

[Cuda] *CUDA API Reference Manual*. 2012. URL: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_Toolkit_Reference_Manual.pdf.

[Cudb] *CUDA C Programming Guide*. NVIDIA Corporation. 2012. URL: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

[DKE10] Steffen Frey Daniel Kauker Harald Sanftmann and Thomas Ertl. *Memory Saving Discrete Fourier Transform on GPUs*. Tech. rep. University of Stuttgart, 2010. URL: http://www.vis.uni-stuttgart.de/~sanftmhd/papers/flexft.pdf.

[DL42] G. C. Danielson and C. Lanczos. *Some improvements in practical Fourier analysis and their application to X-ray scattering from liquids*. Tech. rep. J. Franklin Inst., 1942.

[Ffta] *Fastest Fourier Transform in the West*. URL: http://fftw.org/.

[Fftb] *FFT Accuracy Benchmark Methodology*. URL: http://www.fftw.org/accuracy/method.html.

[Fftc] *FFTw benchmarks*. Accessed May 2012. URL: http://www.fftw.org/speed/CoreDuo-3.0GHz-icc64/.

[FJ05] Matteo Frigo and Steven G. Johnson. "The Design and Implementation of FFTW3". In: *Proceedings of the IEEE* 93.2 (2005). Special issue on "Program Generation, Optimization, and Platform Adaptation", pp. 216–231.

[FJ12] Matteo Frigo and Steven G. Johnson. *FFTW User's Manual*. 2012. URL: http://www.fftw.org/fftw3.pdf.

[Fou07] Jean Baptiste Joseph Fourier. *Mmoire sur la propagation de la chaleur dans les corps solides*. Cambridge University Press, 1807, pp. 215–221. URL: http://gallica.bnf.fr/ark:/12148/bpt6k33707/f220n7.capture.

[Fri99] Matteo Frigo. *A Fast Fourier Transform Compiler*. Tech. rep. MIT Laboratory for Computer Science, 1999. URL: http://www.fftw.org/pldi99.pdf.

[Get09] Pascal Getreuer. 2009. URL: http://www.sal.ufl.edu/NewComers/matlab_optimization_2.pdf.

[GH11] Chris Gregg and Kim Hazelwood. *Where is the Data? Why you cannot debate CPU vs. GPU Performance without the answer*. Tech. rep. University of Virginia, 2011. URL: http://www.cs.virginia.edu/kim/docs/ispass11.pdf.

[Gov+08] Naga K. Govindaraju et al. *High performance discrete Fourier transforms on graphics processors*. Austin, Texas, 2008.

[Gpu] *gpuMat website*. Accessed May 2012. URL: http://gp-you.org/.

[GW06] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006. ISBN: 013168728X.

[Hea02] Michael T. Heath. *Scientific Computing An introductory survey*. Second. Elisabeth A. Jones, 2002.

[Hog74] J. A. Hogbom. "Aperture Synthesis with a Non-Regular Distribution of Interferometer Baselines". In: *A and AA* 15 (June 1974), p. 417.

[HVSB87] Michael T. Heideman Henrik V. Sorensen Douglas L. Jones and C. Sidney Burrus. "Real-valued fast Fourier transform algorithms". In: *IEEE Transactions on Acoustics, SPeech and Signal Processing* (1987), pp. 849–863.

[III84] W.T. Sullivan III, ed. *The early years of radio astronomy*. Press syndicate of the university of Cambridge, 1984. URL: http://books.google.se/books?id=v2SqL0zCrwcC&pg=PA172&redir_esc=y#v=onepage&q&f=false.

[Jac] *Jacket Homepage*. URL: https://www.accelereyes.com/.

[JF] Steven G. Johnson and Matteo Frigo. *Implementing FFTs in Practice*. Fetched May 1, 2012. URL: http://cnx.org/content/m16336/latest/.

[JS12] Emma Johansson and Daniel Samuelsson. *Visualization of lighting for a more secure infrastructure environment*. Tech. rep. Tekniska Högskolan i Jönköping, 2012.

[KCM11] Martin Knapp-Cordes and Bill McKeeman. *Improvements to tic and toc Functions for Measuring Absolute Elapsed Time Performance in Matlab*. Accessed May 12, 2012. 2011. URL: http://www.mathworks.com/tagteam/68600_91934v00_TicToc.pdf.

[KKH10] Neil G. Dickson Kamran Karimi and Firas Hamze. *A Performance Comparison of CUDA and OpenCL*. Tech. rep. D-Wave Systems Inc., 2010. URL: http://arxiv.org/ftp/arxiv/papers/1005/1005.2581.pdf.

[KS11] Pavel Karas and David Svoboda. "Convolution of large 3D images on GPU and its decomposition". In: *EURASIP Journal on Advances in Signal Processing* 2011 (1 2011). 10.1186/1687-6180-2011-120, pp. 1–12. ISSN: 1687-6180. URL: http://dx.doi.org/10.1186/1687-6180-2011-120.

[Leh] Alex Lehar. Accessed 1 July 2012. URL: http://www.picturesolve.com/.

[Liy08] Janaka Liyanage. Fetched March 18 2012. 2008. URL: http://www.cs.ucf.edu/~janaka/gpu/using_nvmex.htm.

[LZ01] Cheng Lizhi and Jiang Zengrong. "An efficient algorithm for cyclic convolution based on fast-polynomial and fast-W transforms". In: *Circuits, Systems, and Signal Processing* 20 (1 2001). 10.1007/BF01204923, pp. 77–88. ISSN: 0278-081X. URL: http://dx.doi.org/10.1007/BF01204923.

[MA03] Kenneth Moreland and Edward Angel. "The FFT on a GPU". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. HWWS '03. San Diego, California: Eurographics Association, 2003, pp. 112–119. ISBN: 1-58113-739-7. URL: http://dl.acm.org/citation.cfm?id=844174.844191.

[Mata] *Matlab documentation on FFTW*. URL: http://www.mathworks.se/help/matlab/ref/fft.html.

[Matb] *Matlab FFTW planner*. URL: http://www.mathworks.se/help/techdoc/ref/fftw.html.

[Matc] *Matlab news version 2012a*. URL: http://www.mathworks.se/support/solutions/en/data/1-HSZ26C/index.html?product=DM&solution=1-HSZ26C.

[Matd] *Matlab threading*. URL: http://www.mathworks.se/help/techdoc/ref/maxnumcompthreads.html.

[Mkl] *Math Kernel Library*. Accessed May 14, 2012. URL: http://software.intel.com/en-us/articles/intel-mkl/.

[MPV11] Franz Franchetti Markus Pschel and Yevgen Voronenko. *Spiral*. Tech. rep. Preprint, fetched June 2012. Carnegie Mellon University, 2011. URL: http://spiral.ece.cmu.edu:8080/pub-spiral/pubfile/paper_146.pdf.

[MSS00] Lawrence O'Gorman Michael Seul and Michael J. Sammon. "Practical Algorithms for Image Analysis". In: Cambridge University press, 2000. Chap. Appendix 1, p. 274.

[MTHB85] Don H. Johnson Michael T. Heideman and C. Sidney Burrus. "Gauss and the History of the fast Fourier transform". In: *Archive for History of Exact Sciences (Springer), 34(3)*. Vol. 1. Springer, 1985, pp. 265–277. URL: http://tinyurl.com/gausshistory.

[Mud09] Dheevatsa Mudigere. "Data Access optimized applications on the GPU using nVidia CUDA". Accessed May 22, 2012. MA thesis. Technische Universitt Mnchen, 2009. URL: http://www5.in.tum.de/pub/mudigere09.pdf.

[Nee] *Need help with QueryPerformanceCounter and Dual Processors*. Accessed May 14, 2012. URL: http://devmaster.net/forums/topic/4670-need-help-with-queryperformancecounter-and-dual-processors/.

[Net] *Netlib*. URL: http://www.netlib.org/liblist.html.

[NK06] Dong-Hoon Noh and JooSub Kim. "Method and apparatus for outputting audio data and musical score image". Patent US 2007/0012165 A1 (US). June 2006. URL: http://www.google.com/patents/US20070012165.

[Ns06] Carl Nordling and Jonny sterman. "Physics Handbook". In: Studentlitteratur, 2006. Chap. M-13, p. 436.

[Nuk06] Akira Nukada. *FFTSS: A High Performance Fast Fourier Transform Library*. Tech. rep. Department of Computer Science, University of Tokyo, 2006. URL: http://www.ssisc.org/fftss/ICASSP2006.pdf.

[Nuk11] Akira Nukada. *Nukada FFT library*. Web site. 2011. URL: http://matsu-www.is.titech.ac.jp/~nukada/nufft/.

[Nvi] *NVIDIA CUDA Programming Guide 4.2*. 2012. URL: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

[NVI12] NVIDIA. *CUFFT libray*. Fetched March 23, 2012. 2701 San Tomas Expressway, Santa Clara, CA 95050, 2012. URL: http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CUFFT_Library.pdf.

[Oct] *Octave webpage*. Accessed May 2012. URL: http://www.gnu.org/software/octave/.

[Oma] *O-Matrix home page*. URL: http://www.omatrix.com/.

[Ope] *OpenCL webpage*. Accessed May 2012. URL: http://www.khronos.org/opencl/.

[Par] *Parallel Computation Toolbox*. URL: http://www.mathworks.se/help/toolbox/distcomp/.

[Par12] Nathan Parrish. *A senior Project*. Tech. rep. Faculty of the Aerospace Engineering Department, California Polytechnic State Univ., 2012. URL: http://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=1084&context=aerosp.

[PF05] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005. Chap. 48. ISBN: 0321335597. URL: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter48.html.

[Pod07] Victor Podlozhnyuk. *Image Convolution with CUDA*. Tech. rep. nVidia Corporation, 2007. URL: http://tinyurl.com/nVidia-conv-separable.

[Pro] *Process Explorer home page*. URL: http://technet.microsoft.com/en-us/sysinternals/bb896653.aspx.

[Rel] *Relux home page*. URL: http://www.relux.biz/.

[RM09] Greg Ruetsch and Paulius Micikevicius. *Optimizing Matrix Transpose in CUDA*. Tech. rep. nVidia Corp., 2009. URL: http://www.cs.colostate.edu/~cs675/MatrixTranspose.pdf.

[SB99] Ivan W. Selesnick and C. Sidney Burrus. "Fast convolution and Filtering". In: *Digital Signal Processing handbook*. Ed. by Vijay K. Madisetti and Douglas B. Williams. CRC press, 1999. Chap. 8. URL: http://www.scribd.com/doc/30855287/Digital-Signal-Processing-Handbook.

[Sci] *Scilab webpage*. Accessed May 2012. URL: http://www.scilab.org/.

[Shu07] Loren Shure. *In-place Operations on Data*. Accessed May 13, 2012. 2007. URL: http://blogs.mathworks.com/loren/2007/03/22/in-place-operations-on-data/.

[Smi11] Julius O. Smith. *Spectral Audio Signal Processing*. W3K Publishing, 2011. URL: https://ccrma.stanford.edu/~jos/sasp/FFT_versus_Direct_Convolution.html.

[Smi97] Steven W. Smith. *The scientist and engineer's guide to digital signal processing*. San Diego, CA, USA: California Technical Publishing, 1997. ISBN: 0-9660176-3-3. URL: www.DSPguide.com.

[Spi] *Spiral Homepage*. URL: http://spiral.net/index.html.

[Sta] *Stackoverflow on slow CUDA initializations*. URL: http://stackoverflow.com/questions/10415204/how-to-create-a-cuda-context.

[Str03] Gilbert Strang. *Introduction to Linear Algebra*. Wellesley-Cambridge Press, 2003.

[Thr] *Thrust homepage*. URL: http://thrust.github.com/.

[TLM] Gallagher Pryor Torben Larsen and James Malcolm. "Jacket: GPU Powered MATLAB Acceleration". In: *in GPU Computing Gems, Jade Edition* 2 (), pp. 387–398.

[Too] *CUDA toolkit*. URL: http://gpgpu.org/wp/wp-content/uploads/2009/06/03-Toolkit.pdf.

[Tor12] Ben Tordoff. *gpuBench*. Acessed April, 2012. 2012. URL: http://www.mathworks.com/matlabcentral/fileexchange/34080-gpubench.

[Tt] Lszl Tth. *Experiences with real-valued FFT algorithms*. Tech. rep. Hungarian Academy of Sciences. URL: http://tinyurl.com/Real-valued-FFT.

[Und] *Undocumented Matlab*. Accessed May 12, 2012. URL: http://undocumentedmatlab.com/.

[Van06] Juul VanderSpek. *nvcc 2.0 documentation*. Tech. rep. NVIDIA Corporation, 2006-2008. URL: http://moss.csc.ncsu.edu/~mueller/cluster/nvidia/2.0/nvcc_2.0.pdf.

[Vec] *Vectura home page*. URL: http://vectura.se/en/.

[VFPR12] Daniel G. Aliaga Vitor F. Pamplona Manuel M. Oliviera and Ramesh Raskar. "CATRA: Interactive Measuring and Modeling of Cataracts". In: *Siggraph 2012* (2012). URL: http://tinyurl.com/Pamplona-tailoring.

[Vis11] Vishy. *A better way to time Jacket code*. Blog. Accessed, June 2012. 2011. URL: http://blog.accelereyes.com/blog/2011/03/30/better_way_to_time_jacket_code/.

[Vol] Volkov. *Volkov home page*. Accessed May 2012. URL: http://www.cs.berkeley.edu/~volkov/.

[Vra] *V-Ray home page*. URL: http://www.chaosgroup.com/en/2/index.html.

[VWLN10] Jatin Chhugani Michael Deisher Daehyun Kim Victor W Lee Changkyu Kim and Anthony D. Nguyen. "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU". In: *Proceedings of the 37th annual international symposium on Computer architecture*. ISCA '10. Saint-Malo, France: ACM, 2010, pp. 451–460. ISBN: 978-1-4503-0053-7. URL: http://doi.acm.org/10.1145/1815961.1816021.

[Wer03] Michael Werman. *Fast Convolution*. Tech. rep. Accessed May 13, 2012. School of Computer Science and Engineering, the hebrew university of Jerusalem, 2003. URL: http://www.cs.huji.ac.il/~werman/Papers/wscg03.pdf.

[WHP07] Willaim T. Vetterling Brian P. Flannery William H Press Saul A Teukolsky. *Numerical Recipes in C, The art of scientific computing*. 3rd ed. Cambridge, 2007. URL: http://www.nrbook.com/nr3/.

[Wow] *Words of wisdom*. Accessed Aug 2012. URL: http://www.nanophys.kth.se/nanophys/fftw-info/fftw_2.html#SEC13.

[XLW09] Xiaoyu Song Xiangyang Liu and Yuke Wang. "Performance Evaluation on FFT Software Implementation". In: *Proceedings of the Internation MultiConference of Engineers and Computer Scientists*. Vol. II. 2009. URL: http://tinyurl.com/FFT-evaluation.