Directional Decomposition of Images: Implementation Issues Including GPU Techniques

Jérôme Dubois

Master of Science in Computer Science
Submission date: February 2008
Supervisor: Anne Cathrine Elster, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

GPUs are becoming increasingly more suitable for general scientific computing.
For instance, NVIDIA® CUDA™ (Compute Unified Device Architecture) is a new programming interface that lets users program NVIDIA General Purpose GPUs (GPGPUs) in a C-like fashion for data parallel intensive computation.
Numerical libraries such as NVIDIA CUBLAS and CUFFT have also recently become available.

In this project, the student will look at a seismic application that uses a filter bank for the directional decomposition of images to remove noise from a ground image.
The goal is to investigate GPU technologies for this problem.

Assignment given: 03. September 2007
Supervisor: Anne Cathrine Elster, IDI

NTNU
Innovation and Creativity

Directional Decomposition of Images:
Implementation Issues Including GPU Techniques

Jérôme Stanislas Augustin Dubois

Master of Technology in Computer Science

Submission date:	February 2008
Advisor:	Anne C. Elster, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

– blank page –

Abstract

Directional decomposition of an image consists of separating it into several components, each containing *directional information* in some specific directions. It has many applications in digital image processing, such as image improvement or linear feature detection, and could be used on seismic data to help geophysicists finding faults. In this thesis, we look at a directional filter bank (DFB) introduced by Bamberger and Smith and how to implement it efficiently on CPU and GPU. *Graphics Processing Units* (GPUs) are becoming increasingly more suitable for general scientific computing, and applications with suitable properties run much quicker on a GPU than a CPU. For instance, NVIDIA® CUDA$^{\text{TM}}$ (Compute Unified Device Architecture) is a new programming interface that lets users program NVIDIA General Purpose GPUs (GPGPUs) in a C-like fashion for data parallel intensive computation.

We translate the DFB algorithm from a theoretical signal processing description to an algorithmic description from computer scientists'point of view, including a readable `C` implementation. Tools are developed to ease our DFB investigation, including a tailored library to manipulate images in suitable text-based and binary formats and for generating test images with suitable properties. Several implementations of 1D filter banks are also provided.

Finally, part of the Bamberger DFB is implemented efficiently using the CUDA environment for NVIDIA GPUs. We show that directional filter banks can efficiently be executed on GPUs and demonstrate that the CPU-GPU bandwidth affects performance considerably. Hence, care should be taken to do as many steps as possible on the GPU before returning results to the CPU.

Biographical Sketch

Jérôme Dubois is born in Lyon, France, in 1984. He started studying in summer 2002 at the *Cycle Préparatoire Polytechnique*, a two year university level course of the *Grenoble Institute of Technology* (INPG) in Grenoble, France, as a preparation to enter French engineering schools. He then entered in summer 2004 the *École Nationale Supérieure d'Informatique et de Mathématiques Appliquées de Grenoble* (ENSIMAG), school of computer science and applied mathematic of the INPG. Since the summer of 2006, he has been a Master student at the *Norwegian University of Science and Technology* (NTNU) in Trondheim, Norway.

This document is the Master thesis done at the *Department of Computer and Information Science* (IDI) of the NTNU. It will also be considered a part of the author's work toward the *diplôme d'ingénieur* from ENSIMAG.

The author of this thesis can be reached by email:

Jerome.Dubois@ensimag.imag.fr

Acknowledgements

Several people have helped me conduct this thesis.

First, thanks to Dr. Elster for being my main advisor and for her training program. I also thank her for her guidance and helping me set priorities. She also provided all the materials I needed for this thesis, including a computer having a high quality graphic card supporting CUDA.

Second, I would like to extend my gratitude to Tore Fevang, software engineer at Schlumberger Trondheim, for his useful help and for always being ready to work with me and answer my questions.

I would also like to thank Rune Jensen, Ingebrigt Hole, and Sigurd Saue, for their help. Rune helped me optimizing the CPU code. Ingebrigt answered my questions about CUDA and connected me with Sigurd when Tore was away. Sigurd helped me analysing the 1D and 2D signals output by the program I had written.

Thanks NVIDIA for allowing me to using the graphic of Figures 3.1 and for the interesting conversation on Tesla products at SC'07.

Finally, I would like to give collective thanks to everyone at Schlumberger Trondheim, for welcoming me and providing tools, including Tore, Wolgang Hochweller, Sigurd, Ingebrigt, Dan, Tore O,...

Contents

List of Figures

List of Tables

Glossary

GPU	Graphics Processing Unit
CPU	Central Processing Unit
FPU	Floating Point Unit
GPGPU	General Purpose Graphics Processing Unit
NVIDIA	Company making hardware and software for graphic, multimedia, and now general computation
CUDA	Compute Unified Device Architecture, by NVIDIA
Schlumberger	Schlumberger Limited is the world's largest[1] oilfield services corporation
Slb	Schlumberger
NTNU	Norwegian University of Science and Technology, Trondheim, Norway
ENSIMAG	École Nationale Supérieure d'Informatique et de Mathématiques Appliquées de Grenoble
FB	Filter Bank
QMF	Quadrature Miror Filter
DFB	Directional Filter Bank
FT	Fourier transform
DTFT	Discrete-time Fourier transform
DFT	Discrete Fourier transform
FFT	Fast Fourier transform
DWT	Discrete Wavelet transform
DCT	Discrete Cosine transform
GRIM	Grey Image (a specific image representation)
LPF	Low-pass filter
HPF	High-pass filter
1D, 2D, 3D	One-, Two-, or Three-dimensional

Chapter 1

Introduction

In this thesis, we take a close look at an algorithm for directional decomposition of images, and describe it and the underlying theories in detail so that it is made accessible to computer scientists without a strong signal processing background.

Directional decomposition of an image consists of separating it into several components, each containing *directional information* in some specific directions. Directional information represent the visual direction of the patterns of the image. For example, a digital image representing a path in a forest might have more directional information in the direction of the path and the vertical direction (the tree trunks) than other directions. Decomposing an image into directional components can be applied in most areas of image processing [1, 2]. It has for example successfully been used for feature extraction and pattern recognition[3].

Graphics Processing Units (GPUs) are pieces of integrated circuits originally designed to process graphic data. GPUs are in a period of fast evolution and they are now suitable for non graphic computation, a concept called *General Purpose GPU*. The interest in GPGPUs have raised all the more that new programming concepts and tools have been developed to ease GPU programming.

In this thesis, we will study a specific directional decomposition algorithm and investigate how to implement it as a traditional CPU program and on a GPU.

Section 1.1 describes the motivation for this project and its goals. We shall then give an outlook of the thesis in Section 1.2. The intended audience of this thesis are professionals in the large field of computer science. Mathematics are recommended to understand this thesis, and we will introduce the digital signal processing concepts needed.

1.1 Motivation and Problem Description

Geophysicist typically need to analyse ground images obtained by various exploration geophysics methods such as reflection seismology. They manipulate ground data in two or three dimensions through specialized interactive software such as *Petrel Geophysics*, and typically apply various transformation and filters on the data and want to visualize the result. Directional decomposition could be used to highlight faults in a ground image [4].

In the history of directional decomposition of images, Bamberger and Smith made a milestone in 1992 by introducing [1] a filter bank using a polyphase filter bank that achieves both directional decomposition and maximally decimation. This means that all the subbands together have the same size than the original signal.

When geophysicist analyse ground data, they do not want to wait too long for the result of an operation, though filtering takes time. The result should therefore be computed quickly. Three solutions hold:

- Run the DFB on the local PC, using the CPU.
  This is the easiest solution, but also probably the less efficient.

- Run the DFB on a remote powerful server, such as a cluster.
  This could give very good performance and is already used in practice for other image operations, at Schlumberger for example. Limitations come from the bandwidth between the client and the server. Additionally, the client need be connected to the server and cannot perform the job alone.

- Run the DFB on the local PC, using advanced hardware features, and GPU in particular. Using a GPU to filter would improve the wall-clock time of the operation, while freeing the CPU at the same time, which could therefore improve the overall performance of the system.

The original goal of this thesis was to parallelize an implementation of the Bamberger directional filter bank (*DFB*) and investigate GPU technologies for this algorithm. However, this became much more challenging than expected because we did not find an initial implementation. We hence ended up studying the algorithm from signal processing papers in order to implement a CPU (serial) version from scratch first. The focus of this thesis is therefore to investigate how to implement the Bamberger DFB both on CPU and GPU.

1.2 Outline

This thesis is structured as follows:

- **Chapter 2** Gives digital image processing background including a presentation of the Bamberger directional decomposition filter bank. In this chapter, we will first present the theoretical tools needed to understand the filter bank algorithm. We seek to give an intuitive understanding of what a directional filter bank does and how it works. The 2-band filter bank which we focus on is presented in detail, and a review of different methods to use it in a N-band filter bank is given.

- **Chapter 3** introduces general programming on GPU, including a presentation of GPUs. Emphasis is put on presenting `CUDA` technology for GPGPU programming.

- **Chapter 4** presents implementation issues of filters, and describes step by step how to implement the 2-band Bamberger DFB from a computer-scientists'point of view. The tools and programs developed during this thesis are then introduced. These include a toolkit library for image manipulation, some code in `octave`, `C` implementations of a 1D non-polyphase filter bank, a 1D polyphase filter bank, a 2-band Bamberger DFB, and eventually the `CUDA` implementation of horizontal and vertical 2D convolution used by the Bamberger DFB.

- **Chapter 5** presents the results of the programs developed and compares performance of the CPU and GPU versions.

- **Chapter 6** summarizes the contribution of this thesis and suggests future work.

In addition to the above, appendices include practical information related to this thesis:

- **Appendix A** presents the most important papers and books of the bibliography, including those which might be important for a possible future work.

- **Appendix B** gives an overview of the programs developed during the thesis. Important parts of the programs are listed, including the core of the DFB and GPU code.

Chapter 2

Image Processing

This chapter gives digital image processing background including a presentation of the Bamberger directional decomposition filter bank. In Section 2.1, we present the theoretical tools needed to understand a 1D polyphase filter bank algorithm. In Section 2.2, we seek to give an intuitive understanding of what a *directional* filter bank does and how it works. The 2-band filter bank which we focus on is presented in Section 2.3, and a review of different methods to use it in a N-band filter bank is given.

2.1 1D Filters

This section is an overview of 1D signal processing basics. Concepts needed to understand a polyphase filter bank are introduced.

2.1.1 Fourier Related Transforms and Convolution

The below subsection presents some mathematical background of 1D filters, including four transforms and the concept of convolution.

Fourier analysis is a pillar of signal processing. Many transforms derive from the *Fourier transform*. They transform a function into an other. Instead of functions, we can think of *signals*. If we note x the input signal, it is common to use the corresponding capitalized letter to represent its transform: X. It is a habit in signal processing to use the expressions *time domain* (or *space domain*) to refer to the domain of the original signal, and *frequency domain* (or *Fourier domain*) to refer to the domain of the transformed signal. All Fourier-related transforms used here are invertible. x and X are therefore just different ways to represent the same signal. These transforms do not lose or add any information, but reorganize it. Information organization matters because it permits to process it according to a semantic need, including compression, filter, and extraction.

These transforms are presented in many (digital) signal processing books. In addition to definitions and properties, Deller gives in his paper *Tom, Dick, and Mary Discover the DFT* [5] a deep understanding of the relationship between the FT, DTFT and DFT, three of the transforms presented below.

FT

If we note $x(t)$ a continuous signal with $t \in \mathbb{R}$, then its *Fourier transform* $X_{FT}(t)$ is:

$$X_{FT}(f) = \int_{-\infty}^{\infty} x(t)\, e^{-i2\pi ft}\, dt \qquad \text{with } f \in \mathbb{R}$$

Here and thereafter, i denotes the imaginary unit ($i^2 = -1$). The corresponding inverse Fourier transform is:

$$x(t) = \int_{-\infty}^{\infty} X_{FT}(f)\, e^{i2\pi ft}\, df$$

Figure 2.1 shows in the left column some basic signals, and in the right column their Fourier transform.

DTFT

The signals with which we work are discrete. Therefore, a suitable transform is used: the *discrete-time Fourier Transform* (DTFT)[5, 6, 7]. The DTFT is closely related to the continuous FT in that if $x[n]$ are regular spaced samples of the continuous signal $x(t)$, then the DTFT $X(\omega)$ of $x[n]$ is a scaled periodic replica of the Fourier transform $X_{FT}(f)$ of $x(t)$ [5][1]. The definition of the DTFT is[2]:

$$X(\omega) = \sum_{n=-\infty}^{\infty} x[n] e^{-i\omega n} \qquad \omega \in \mathbb{R}$$

Because we work with time-limited signal (i.e. null outside of a close subset), the summation can be done only for $n = 0, \ldots, N-1$. It is obvious from this formula that the DTFT is periodic with period 2π (because $e^{i(\omega+2\pi)} = e^{i\omega}$). Reasoning and evaluating $X(\omega)$ on one period is sufficient. We will use the period $[-\pi, \pi]$.

[1]From [5] equation (20): $X_{DTFT}(f) = \sum_{n=-\infty}^{\infty} X_{FT}(f - nf_s)$, with f_s the sampling frequency ($f_s = \frac{1}{T_s}$ and $x[n] = x(nT_s)$) and $\omega = 2\pi T_s f$.

[2][5] gives a slightly different definition which emphasize the relation to the continuous FT: $X(f) = \sum_{n=-\infty}^{\infty} x(nT_s) e^{-i2\pi f nT_s}$ with the notation of the previous footnote.

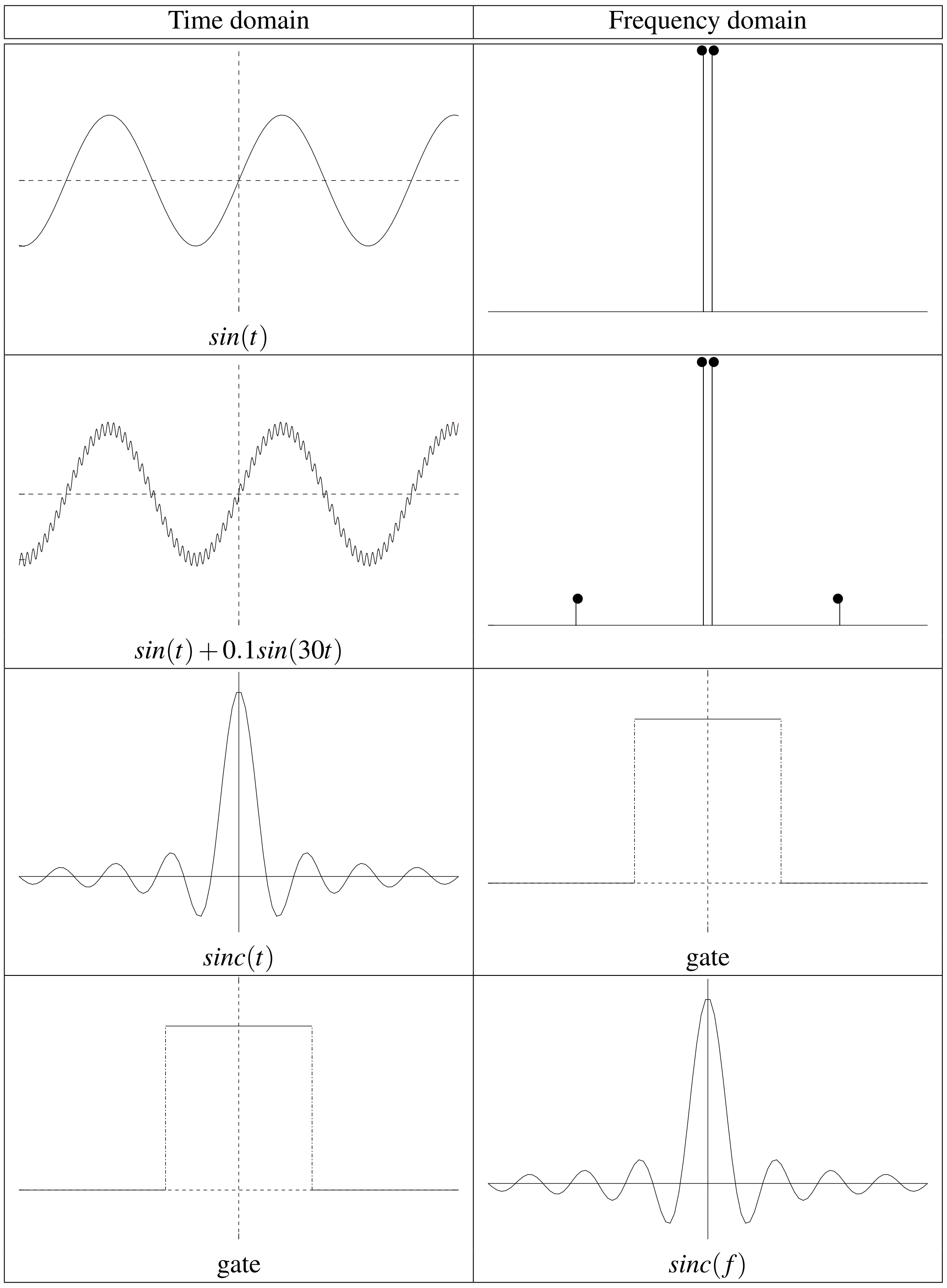


Figure 2.1: Example of Signals and their Fourier Transform

The DTFT is invertible and its inverse is:

$$x[n] = \frac{1}{2\pi}\int_{-\pi}^{\pi} X(\omega)\cdot e^{i\omega n}\, d\omega$$

From now, we will mainly work with the DTFT, and $X(\omega)$ represent the DTFT of $x[n]$. Like many other places in the literature, we will sometime abusively use the term *Fourier transform* or *FT* to refer to the *DTFT*.

DFT

The DTFT transforms a discrete signal into a continuous signal. If this permits to represent nice spectra which make reasoning easy, it is not directly possible for a computer to work with discrete signals. However, computing only N well chosen samples of the DTFT, N being the number of sample of the time domain signal, is enough to recover the original time-domain signal[5]. This samples are called the *discrete Fourier transform* (DFT). If we note $x[n]$ a discrete signal, its discrete Fourier transform $X_{DFT}[k]$ is:

$$X_{DFT}[k] = \sum_{n=0}^{N-1} x[n] e^{\frac{-2i\pi}{N}kn} \qquad k = 0, \ldots, N-1$$

It is invertible and the inverse is:

$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} X_{DFT}[k] e^{\frac{2\pi i}{N}kn} \qquad n = 0, \ldots, N-1$$

The DFT is no more than samples of the DTFT with $\omega = \frac{2\pi}{N}k$, the important result being that these samples are enough to recover the time-domain signal (see [5] and [8]).

FFT

It is easy to program a DFT function on a computer. A straight-forward implementation typically has a complexity of $O(N^2)$. However, less time-consuming algorithms exists. Much research has been done both to find faster algorithms and to program them efficiently. The most known DFT algorithm is called the *fast Fourier transform* (FFT) [9] and has a complexity of $N\log(N)$. The most efficient freely available implementations of the FFT are `FFTW` [10] and `djbfft` [11].

Convolution

The *convolution*, noted $u \star v$ of two discrete[3] signals u and v is a signal y such that $\forall n,\ y[n] = (x \star v)[n] = \sum_k u[n]v[n-k]$. It is commutative, associative, and distributive.

The *convolution theorem* states that the Fourier transform of a convolution product is the point-wise product of the transformed signal[4]: if $y = u \star v$ then $Y = U \cdot V$.

When one wants to compute a convolution comes the choice to compute it in the time domain, using a straightforward use of the convolution definition, or in the frequency domain, by transforming the signals, computing the pointwise product of the convolution theorem, and transforming back the result. We shall explore the trade-off between of these two methods later (in 2.1.2).

Z-transform

The *Z-transform* transform a signal x into X_Z according to:

$$\forall z \in \mathbb{C}, X_Z(z) = (\mathcal{Z}(x))(z) = \sum_n x[n]z^{-n}$$

It is revertible[5]. It is noteworthy that the DTFT is a specific case of the Z-transform[6], evaluated on the unit circle[6]. We will abusively note $X(z)$ the Z-transform and $X(\omega)$ the DTFT[7]. The Z-transform domain is also called frequency domain. Some properties of the Z-transform are presented in Table 2.1. The Z-transform is the discrete equivalent of the Laplace transform.

2.1.2 LTI Systems and Filters

LTI Systems

> *"To study multidimensional systems productively, it is necessary to restrict our investigations to certain classes of operators which have properties in common. Linear shift-invariant (LSI) discrete systems*

[3]Equivalent definition and results hold for continuous signal

[4]Using any of the transforms. If the frequency domain is continuous, then the normal product is used.

[5]From [12], the reverse Z-transform is $x[n] = \mathcal{Z}^{-1}(X(z)) = \frac{1}{2\pi i}\oint_C X(z)z^{n-1}dz$, where C is a counterclockwise closed path encircling the origin and entirely in the region of convergence (ROC).

[6]$X_Z(e^{i\omega}) = \sum_n x[n]e^{-i\omega n} = X_{DTFT}(\omega)$

[7]It is common in the literature to use the same notation for both transforms. So do [1]. A more formal way is to note $X(z)$ the Z-transform and $X(e^{i\omega})$ the DTFT.

Table 2.1: Z-transform Properties

	Time domain	Frequency domain (Z-transform)
notation	$x[n]$	$X(z) = X_Z(z)$
linearity	$a_1x_1[n] + a_2x_2[n]$	$a_1X_1(z) + a_2X_2(z)$
time shifting	$x[n-k]$	$z^{-k}X(z)$
time reversal	$x[-n]$	$X(z^{-1})$
convolution theorem	$x_1[n] \star x_2[n]$	$X_1(z) \cdot X_2(z)$

are the most frequently studied class of systems for processing discrete signals of any dimensionality. These systems are both easy to design and analyse, yet they are sufficiently powerful to solve many practical problems." – Dudgeon and Mersereau [13]

A *Linear Time-Invariant* (LTI) or *Linear Shift-Invariant* (LSI) system[8] S is a system whose output signal y is linked to the input signal x according to:

- *linearity*: if $\forall n$, $x[n] = ax_1[n] + bx_2[n]$, and $S(x_1) = y_1$ and $S(x_2) = y_2$ then $\forall k\ y[k] = ay_1[k] + by_2[k]$
- *time invariance*: $\forall n, T$, $S(x[n-T]) = y[n-T]$

A LTI system can be completely characterized by a function h called *impulse response*, which is the output of the system when the input is the Dirac function $\delta[n]$. The output of the system is obtained by convolution of the input signal x and the impulse response h: $y = x \star h$.

Using the convolution theorem (see 2.1.1) and noting H the Fourier transform of h, we get $Y = X \cdot H$. Therefore, H also characterises the system. It is called the *transfer function*. Similarly, it can be characterized by the Z-transform H_Z of h, and $Y_Z = X_Z \cdot H_Z$.

Digital filters

Lutovac et al. [7] define a *digital filter* as

"a discrete-time system that alters the spectral information contained in some discrete-time signal x producing a new discrete signal y."

[8]LTI is the general term, whereas LSI is the name of the concept while dealing with discrete-time signals[14].

Considering LTI systems, we can express the result by the linear difference equation:

$$y[n] = \sum_{k=1}^{N} b_k x[n-k] - \sum_{k=1}^{N} a_k y[n-k]$$

If $\exists k, a_k \neq 0$, the filter is said to have an infinite impulse response (IIR). If $\forall k, a_k = 0$, the filter has a finite impulse response (FIR). Because IIR filters are difficult to design and implement[4], we will use FIR filters.

Frequency Domain

It can be helpful to reason in the frequency domain about the effect of a filter. For example if one wants to attenuate some frequency component of the signal or amplify other frequencies. Audio devices often show the Frequency domain representation of their equalizer. The value of $H(\omega)$ shows by how much the filter multiplies the frequency ω of the signal. Finding a transfer function H with good properties is part of the design of filters [7]. For example, a *pass-band* filter keeps only some frequencies, which corresponds to a multiplication by 1, and stop all the others, which corresponds to a multiplication by 0.

Because the N DFT samples $X_{DFT}[k]$ fully represent the signal, such a filter can be implemented by:

1. computing the DFT of $x[n]$: $X_{DFT}[k] = X(\frac{2\pi}{N}k)$
2. multiplying every value by the transfer function: $Y_{DFT}[k] = X_{DFT}[k] \cdot H_{DFT}[k]$
3. computing the inverse DFT of $Y_{DFT}[k]$: $y[n]$

Using good implementations of complexity $N\log(N)$ of the DFT, the total filter has a complexity of $N\log(N)$.

Time Domain

Filtering in the time domain is pretty easy, since it consists in doing a convolution between the signal x and the impulse response h. If h has NB_COEF coefficients, convolving implies doing $N * NB\_COEF$ multiplication and $N * (NB\_COEF - 1)$ additions (derived from the convolution definition 2.1.1). The complexity is therefore in $O(N * NB\_COEF)$.

If filters are most of the time designed in the frequency domain, a choice has to be made whether to implement it in the time or frequency domain. Because in many applications, $NB\_COEF = N$ or NB_COEF is big, using the frequency domain method is common, and it is called *Fast convolution* [15, 16, 17]. However,

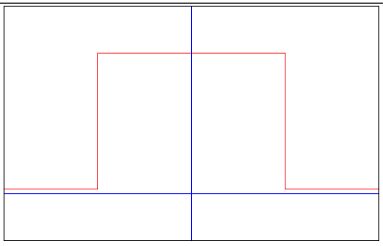
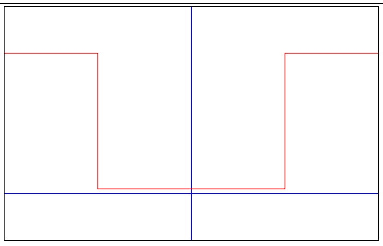
Low-pass filter $H_0(\omega)$	High-pass filter $H_1(\omega)$
	

Figure 2.2: Low-pass and High-pass Filter Transfer Functions

using the normal convolution algorithm gives better performances when the size of the filter is small. The choice is therefore case specific.

The targeted architecture should also be taken in consideration. In the time domain algorithm, the signal is used as a stream, whereas in the frequency domain, it can be computed block by block in parallel for example.

Note that this discussion will take place as well when we will study 2D signals, and the conclusions may differ.

High-Pass and Low-Pass filter pair

Two specific types of pass-band filters are widely used: *high-pass* filters stop the low frequencies of a signal, and *low-pass* filters stop the high frequencies. It is a habit to call h_0 the low-pass filters and h_1 the high-pass filters. The frequency domain of an ideal low-pass and high-pass filters are shown in Figure 2.2. Recall that the frequency domain is periodic. We show only one period of the filters. The limit frequency of their pass-band region is called *cut-off* frequency.

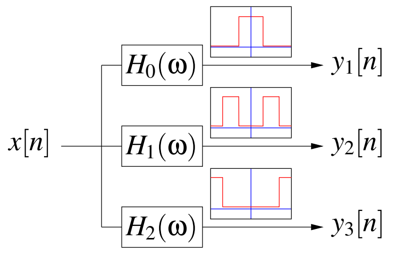
It is common to use a pair of low-pass and high-pass filters with the same cut-off frequency, so that they separate the signal into two complementary subbands.

The inverse FT of a gate is a cardinal sinus (sinc) function (See Figure 2.1 for an illustration), so the impulse response coefficients $h_0[n]$ are regularly spaced samples of a sinc.

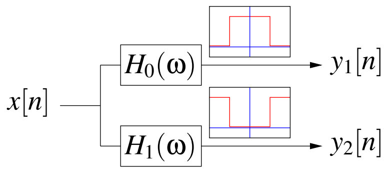
2.1.3 Filter Banks and Polyphase Filter Banks

Filter banks

A *filter bank* is a group of filters that separates the input signal x into components that contains subband frequencies. Figure 2.3 describes (a) a 3-channel filter bank and (b) a 2-channel filter bank. In the figures of this report, We shall represent



(a) 3-channel filter bank



(b) 2-channel filter bank

Figure 2.3: Basic 1D Filter Banks

the filters by the name of their transfer function (e.g.: $H(\omega)$) in a box, sometime showing a sketch of it beside the box.

We will now consider the two-channel filter bank of Figure 2.3 (b). It separates the input signal x into its low frequency subband y_0 in the first band and high frequency subband y_1 in the second band. Trying to recover x from its subbands y_i is called reconstruction. If we note $\hat{x}$ the resulting signal, the filter bank is said to be *perfect reconstruction* [2] if $\hat{x}[n] = x[n]$. The process of decomposing the signal into subbands is called *analysis* whereas the phase to reconstruct the signal is called *synthesis*. The filters of the synthesis part are noted $G_0(\omega)$ and $G_1(\omega)$. Daubechies wrote [18] in a tutorial on wavelets: *"The purpose of subband filtering is of course not to just decompose and reconstruct. The goal of the game is to do some compression or processing between the decomposition and reconstruction stages."* We will present some applications of a filter bank in the 2D case in 2.2.

Signals y_1 and y_2 have the same size N than x. Therefore, we need two times more space for the output of the filter than for the input. It can be shown that by removing half of the samples, we can still get perfect reconstruction [19], assuming that each filter passes half of the original spectrum. This step is called *downsampling* the signals, is done by keeping every other sample, and is noted by a a downward arrow followed by the downsampling factor (in our case: 2) on the Figures. The signals need to be *upsample* back to their original size in order to reconstruct the signal. The term *maximal decimation* is used when the set of subbands has the same amount of samples than x. For example, in a k-channel filter bank, each subbands contains $\frac{N}{k}$ samples. Figure 2.4 shows a complete filter bank including decimation and reconstruction.

There have been done much research on design of good and appropriate filters. In particular, we seek a filter pair that permits perfect reconstruction. *Quadrature mirror filter* (QMF) are often used when aliasing cancellation is considered [1]. Deller et al. [19, page 494] explain the design of a filter bank using QMF filters having the properties summarised in Table 2.2.

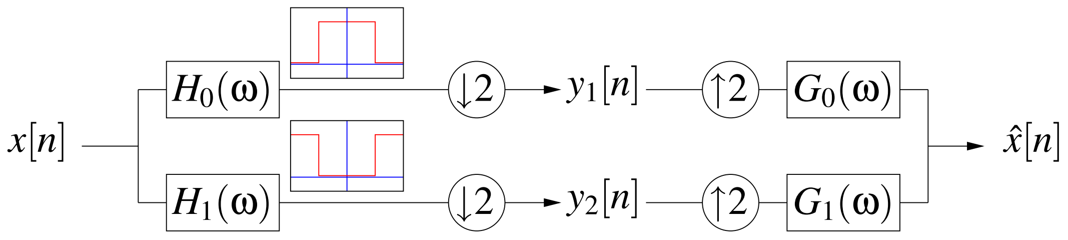


Figure 2.4: 2-channel Filter Bank with Reconstruction and Decimation

Table 2.2: Properties to Get Perfect Reconstruction

	Time domain	Frequency domain
Mirror	$h_1[n] = (-1)^n h_0[n]$	$H_1(\omega) = H_0(\omega - \pi)$
Quadrature		$\mid H_0(\omega)\mid^2 + \mid H_1(\omega)\mid^2 = 1$
	$g_0[n] = 2h_0[n]$	$G_0(\omega) = 2H_0(\omega)$
	$g_1[n] = -2h_1[n]$	$G_1(\omega) = -2H_1(\omega)$

Polyphase Filter Banks

In the filter bank describe in the previous section and represented in Figure 2.4, we compute the filters on signals of length N just before throwing half of the result in the downsampling phase. In order to save computation, the concept of *polyphase* filter bank has been introduced, where the signal is downsampled *before* being filtered, thus saving about half of the computation of the filters. However, special care need to be taken. First, if we downsample the signal and process the output on the two channels, it would mean that the sample thrown are really lost and not recoverable. The solution is to process every even sample in the first band and every odd sample in the second band. Second, we would like the subbands to be the same as previously in the non polyphase filter bank. For that, we need to combine the result of the two filters together. The polyphase filter bank used here is the one presented by Vaidyanathan in [20, 21] and summarized in [19]. This filter bank is shown on Figure 2.5. The sign $\oplus$ represent the addition of two signals and $\ominus$ the negation of one signal. Using the Z-transform notation and its properties summarized in Table 2.1, we note z^{-1} a time delay of one sample. Similarly, a downsample can be represented by z^2 and an upsample by $z^{\frac{1}{2}}$.

Deller et al. [19] present which filters to use in this filter bank:

$$p_0[m] = h_0[2m]$$
$$p_1[m] = h_0[2m+1]$$

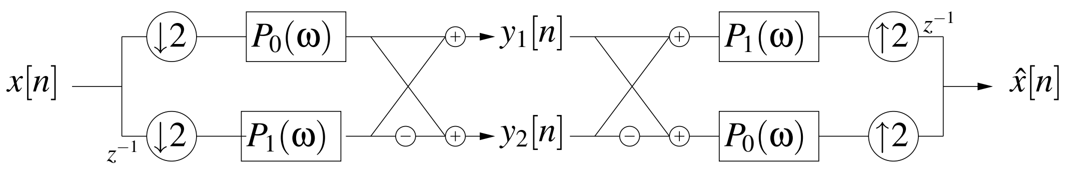


Figure 2.5: Polyphase 1D Filter Banks. This figure is a combination of Figure 8 in [21] and Figure 7.50 in [19]. We use the same subbands than [21], whereas they are inverted in the synthesis part of [19].

with $m = 0, \cdots, \frac{N}{2}$.

In this project, we have implemented both the 1D simple filter bank and the 1D polyphase filter bank in standard `C`.

2.2 Directional Representation of Images

This section explains what is directional information of an image and what is the aim of directional filters.

Directional decomposition of images is a process which consists in decomposing an image into several subbands, each representing directional information or *energy* along one direction. As in the 1D case, the inverse process can be applied on the subbands in the process of reconstruction.

A directional decomposition can be used in many fields in image processing [2, 1, 22], for example for common automatic target recognition, texture analysis, segmentation, and classification, image denoising and enhancement, linear feature or edge detection, computer vision, determining the direction of a wave or a plane. In particular, a directional decomposition of image can help removing ground roll or detecting faults in seismic data.

To help understanding what is directional decomposition of an image, let us consider the three images of Figure 2.6. Both images (a) and (b) have information in only one direction. If we combine these images taking for each pixel of the output image the average of the corresponding pixels on the input images, we get image (c). A perfect directional decomposition of image (c) along two dimensions would give two subbands which represent images (a) and (b).

2.2.1 Image coding

A raster graphic image, as opposed to vector graphic image, is a two dimensional array of pixels. Each pixel color is coded by a number (float or integer) in a given

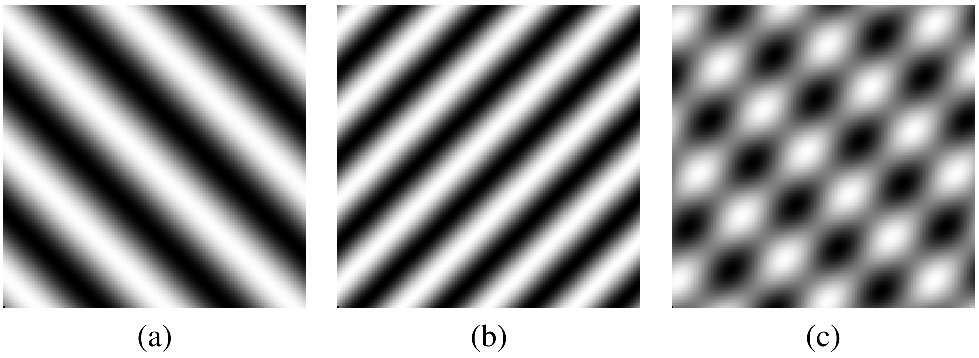


Figure 2.6: Simple Directional Decomposition Example. (a) and (b) have information in directions $\pi/4$ and $3\pi/4$ respectively (see Figure 2.8). (c) is an average of (a) and (b).

range.

In this thesis, we work with grey images that have 256 levels of grey. A pixel color is thus represented by an integer between 0 and 255. However, as we process these images, operations may return non integer pixels. In the example above (Section 2.2 Figure 2.6), taking the average of two pixels may return a real number. While processing images, pixels will be represented by floating numbers. They are converted to the output format at the end of the process. In the theoretical parts, we consider real numbers.

2.2.2 Frequency Domain and Direction Information

An image is viewed as a discrete two dimensional signal $x[n]$, with $n = \begin{pmatrix} n_1 \\ n_2 \end{pmatrix}$. The one-dimensional Discrete Fourier Transform has been generalized to the multidimensional case and many properties remain true. A short introduction to the processing of multidimensional digital signals can be found in [23], whereas [13] covers the subject in great details. Since images are discrete signals, they are processed with the Discrete Fourier Transform (DFT) instead of the standard Fourier transform. Deller explains in [5] the relationship between continuous and discrete Fourier transforms. If we note $n = \begin{pmatrix} n_1 \\ n_2 \end{pmatrix}$ and $\omega = \begin{pmatrix} \omega_1 \\ \omega_2 \end{pmatrix}$, the (2D discrete) Fourier transform $X[\omega]$ of a signal $x[n]$ is defined by :

$$X[\omega] = \sum_{n \in \Gamma} x[n] e^{-j\omega^T n}$$

with Γ the integer lattice of points.

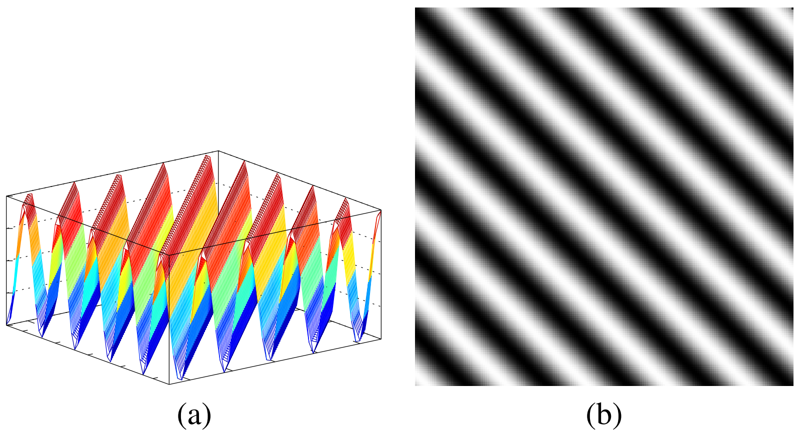


Figure 2.7: Representation of a 2D Space Signal (a) in 3D, (b) in 2D.

Again, as in the 1D case, the Fourier transform still fully represent the time domain signal. No information is lost, and recovering the time domain signal is possible using the inverse transform. It is just another way to represent the information.

Let us consider the signal $x[n] = sin(-3n_1 + 3n_2)$. We can represent it by a three dimension graph as in Figure 2.7 (a). However, because we work with images, we will thereafter represent such a signal by a 2D image in which the pixel grey levels represent the third dimension. Thus, Figure 2.7 (b) represents the same signal.

The Fourier transform $X[\omega] = X[\begin{pmatrix} \omega_1 \\ \omega_2 \end{pmatrix}]$ of a two dimensional signal $x[n] = x[\begin{pmatrix} n_1 \\ n_2 \end{pmatrix}]$ is also a two dimensional signal and can similarly be represented by a 3D graph or a 2D image. We will use only image representation. Note that Frequency domain images in the figures of this document are not exact. They have not been calculated (computing a DFT) but drawn.

As drawings say more than a long text, we show several 2D signals and their Frequency domain equivalent to illustrate some 2D Fourier transform properties. Figure 2.8 shows how the FT of a signal represent directional information. Figure 2.9 shows that low frequencies are near the center and higher frequencies are more far. Figure 2.10 shows that an image have different directional information according to its components. With Figure 2.11, we want to be closer to real word images by showing the FT of lines of different width. A comparison is done with the 1D case.

This section's aim was to give a feeling of how *locality* information in a Fourier transform image represent *directional* information of the corresponding

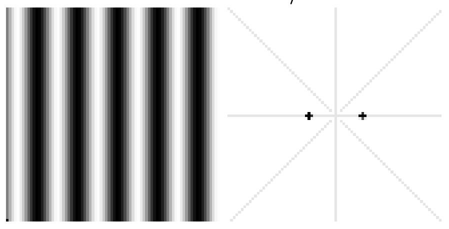
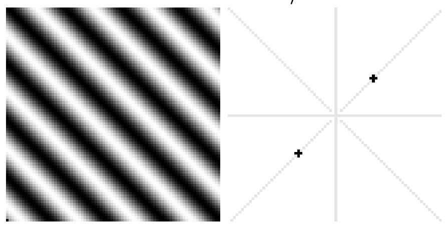
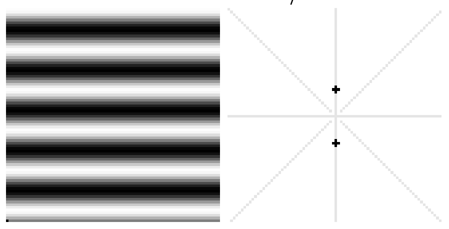
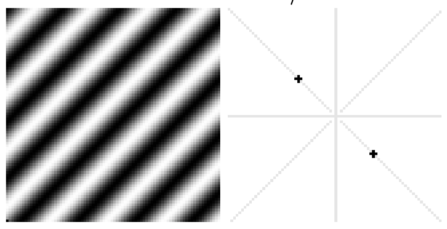
Direction 0π/4	Direction 1π/4
	
Direction 2π/4	**Direction 3π/4**
	

Figure 2.8: 2D Signals and their Fourier Transform: Directions

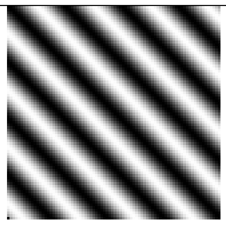
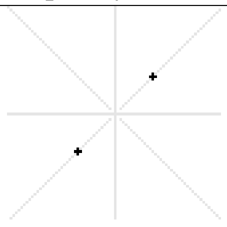
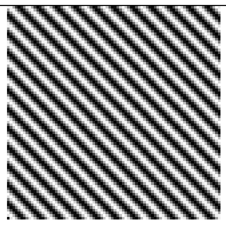
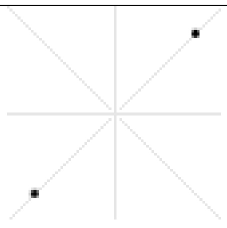
	Time domain	Frequency domain
Low frequency		
High frequency		

Figure 2.9: 2D Signals and their Fourier Transform: Frequencies

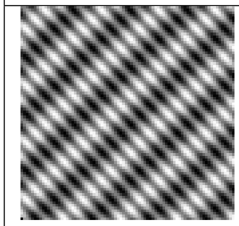
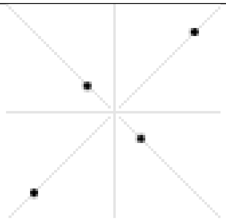
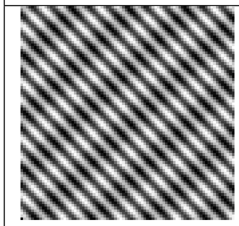
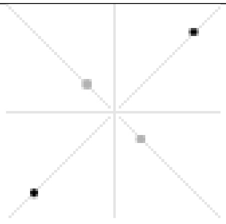
Time domain	Frequency domain
	
	

Figure 2.10: 2D Signals and their Fourier Transform: superposition of a sinus of direction $3\pi/4$ and frequency f and a sinus of direction $\pi/4$ and frequency $3f$. In the second image, the low frequency sinus is with coefficient 1 and the high frequency with coefficient 3. Both sinuses are with the same coefficient in the first image.

time domain image.

2.2.3 Directional Decomposition

Definition and Notation

Decomposition is the process of separating the input image into several components, each one containing a set of frequency subbands. These sets of frequency subbands can be represented in the frequency domain by partitioning the 2D spectrum. Figure 2.12 (a) shows such a decomposition into five components. Each of the component is obtained by applying a 2D filter to the image. The filter to separate the fourth component can be represented as in Figure 2.12 (b), where the pass-band region is dark and the stop-band region is white. If we call F_4 this filter, $F_4(\omega) = 1$ where the point ω is dark, and $F_4(\omega) = 0$ everywhere else. The decomposition described in Figure 2.12 (a) is done by a 5-band filter bank represented by Figure 2.13.

Spectrum Partitioning

The DFB introduced studied in this project has wedge shaped passband regions, as in Figure 2.14

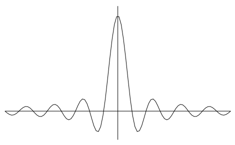
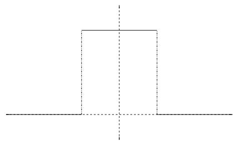
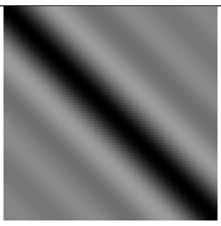
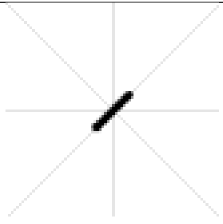
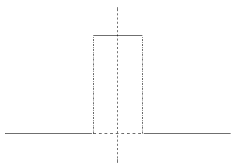
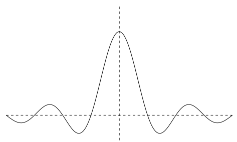
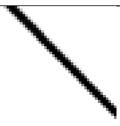
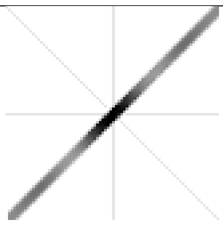
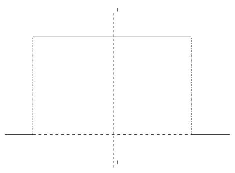
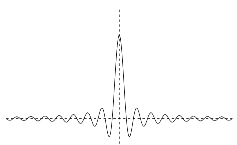
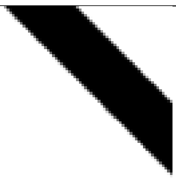
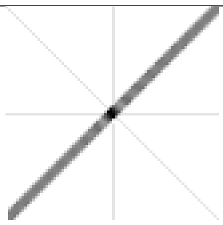
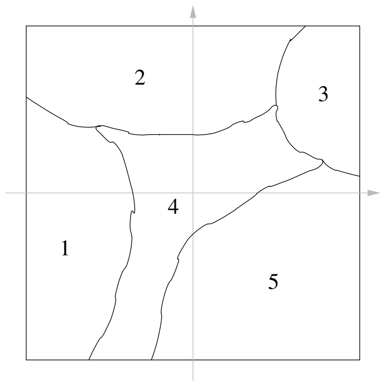
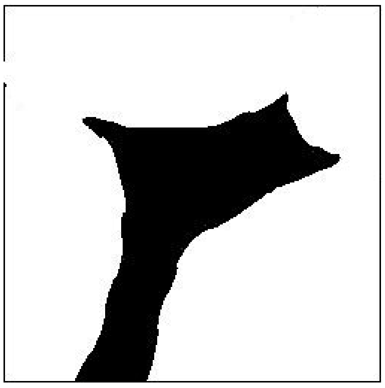
1D		2D	
Time	FT	Time	FT
 *sinc*(n)	 gate		
 gate	 *sinc*(ω)	 thin line	
 gate	 *sinc*(ω)	 wide line	

Figure 2.11: 2D Signals and their Fourier Transform: Line



(a)



(b)

Figure 2.12: 2D Spectrum Partitioning. (a) is a possible 5-band partition of an image FT, and (b) represent the filter $F_4(\omega)$ that extracts the fourth subband region.

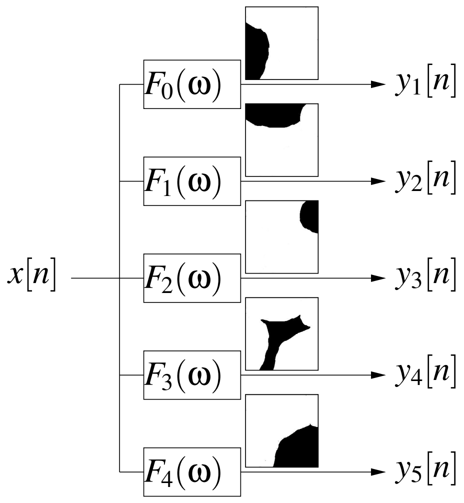


Figure 2.13: The 2D 5-band Filter Bank, corresponding to the partitioning of Figure 2.12

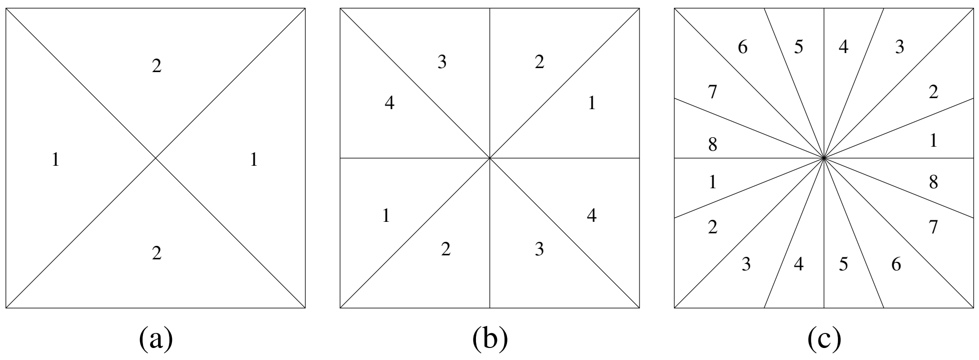


Figure 2.14: Conventional DFB Spectrum Partitioning for (a) two subbands, (b) four subbands, (c) height subbands

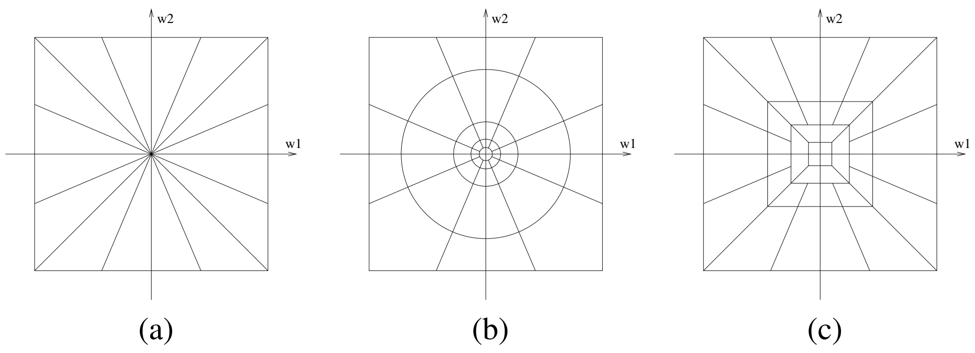


Figure 2.15: Different DFB Spectrum Partitioning: (a) Conventional (Bamberger and Smith), (b) Steerable pyramid (Simoncelli et al.), (c) Pyramidal (Do et al.). The idea for this figure was taken from Figure 2.6 in [2]

Though this partitioning is common, it has been criticised[2]. Most of the energy of an image is in the low frequencies. Therefore, the filters need to be very sharp at the center because a small perturbation could have a significant impact. Other directional partitioning have been proposed that takes this phenomenon into account. Figure 2.15 shows two other solutions. However, algorithms for these partitioning are less computationally efficient [4], and the conventional partitioning is often used in practice. A high pass filter can possibly be applied on the image before, in order to remove the very low frequencies too close to the center $(0,0)$ [2, page 28].

2.3 2D Filter Banks

This section extends to the 2D case the theory presented in Section 2.1 for the 1D case, and present the Bamberger polyphase filter bank.

2.3.1 Fourier Related Transforms, Convolution, and Resampling

Some mathematical background of 2D filters is presented.

DFT

As seen in 2.2.2, the 2D DFT is:

$$X(\omega) = \sum_{n \in \Gamma} x[n] e^{-j\omega^T n}$$

Convolution

The *convolution*, noted $u \star v$ of two 2D signals u and v is a signal y such that $\forall n = \begin{pmatrix} n_1 \\ n_2 \end{pmatrix} \in \mathbb{N}^2, y[\begin{pmatrix} n_1 \\ n_2 \end{pmatrix}] = (x \star v)[\begin{pmatrix} n_1 \\ n_2 \end{pmatrix}] = \sum_{k_1} \sum_{k_2} u[\begin{pmatrix} k_1 \\ k_2 \end{pmatrix}] v[\begin{pmatrix} n_1 - k_1 \\ n_2 - k_2 \end{pmatrix}]$. It is commutative, associative, and distributive.

As in the 1D case (see 2.1.1 page 8), The *convolution theorem* states that the Fourier transform of a convolution product is the multiplication of the Fourier transforms: if $y = u \star v$ then $Y = U \cdot V$. Thanks to this result, a convolution product can be computed in the time domain or frequency domain, as we shall discuss later.

Z-transform

The two-dimensional Z-transform X_Z of a signal x is defined as:

$$X_Z(z) = X_Z \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \sum_{n_1} \sum_{n_2} x \begin{bmatrix} n_1 \\ n_2 \end{bmatrix} z_1^{-n_1} z_2^{-n_2}$$

The two-dimensional Z-transform has many of the properties of its one-dimensional equivalent. Some of them are presented in Table 2.3. These properties are detailed in [13] .

Downsampling

Downsampling [1, 13, 2], sometime referred as *decimation* [23, 2] is an operation consisting in removing some of the values of a signal and rearranging the values kept. The resulting signal is smaller, but information has been lost. A simple downsampling would be to keep only values with even coordinates. The resulting image would be four times smaller (two times in each direction). The downsampled signal x_d of a 2D signal x is characterized by a 2×2 matrix μ and:

$$x_d[n] = x[\mu n]$$

An example of downsampling operation is in Figure 2.16 where the image is downsampled by $\mu = \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}$

Upsampling

Upsampling [1, 13, 2], sometime referred as *expansion* [2] (and related to *interpolation* [23]) is an operation consisting in expanding a signal by rearranging the values and adding zeroes. The resulting signal is bigger, but no information has

Table 2.3: 2D Z-transform Properties. We use both the vertical and horizontal notation, though it would be more rigorous to use the transpose $(n_1, n_2)^T = \begin{pmatrix} n_1 \\ n_2 \end{pmatrix}$

	Time domain	Frequency domain (Z-transform)
notation	$x[n] = x\begin{bmatrix} n_1 \\ n_2 \end{bmatrix} = x[n_1, n_2]$	$X(z) = X_Z(z) = X_Z\begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = X_Z(z_1, z_2)$
separable signals	$x[n] = v[n_1]w[n_2]$	$X(z) = V(z_1)W(z_2)$
linearity	$a_1x_1[n] + a_2x_2[n]$	$a_1X_1(z) + a_2X_2(z)$
linearity	$a_1u[n] + a_2v[n]$	$a_1U_Z(z) + a_2V_Z(z)$
time shifting	$x[n+k]$	$z_1^{k_1}z_2^{k_2}X(z)$
reflexion	$x[-n_1, n_2]$ $x[n_1, -n_2]$ $x[-n_1, -n_2]$	$X(z_1^{-1}, z_2)$ $X(z_1, z_2^{-1})$ $X(z_1^{-1}, z_2^{-1})$
modulation	$a^{n_1}b^{n_2}x[n]$	$X(a^{-1}z_1, b^{-1}z_2)$
convolution theorem	$u[n] \star v[n]$	$U(z) \cdot V(z)$

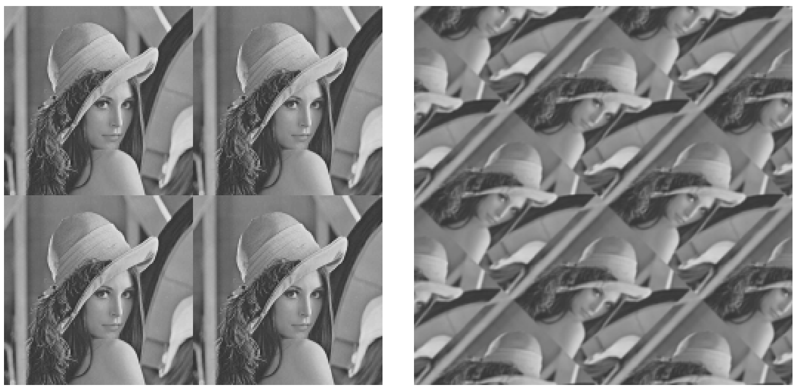

(a) (b)

Figure 2.16: Downsampling Operation: (a) Periodic *Lena* image (b) *Lena* is downsampled by $\begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}$

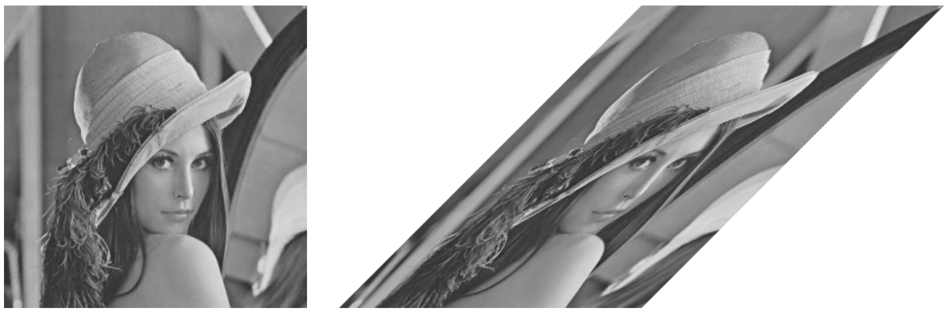


Figure 2.17: Resampling Operation: (a) *Lena* image (b) *Lena* is resampled by $R = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$. Here, $|R| = 1$.

been added. The upsampled signal x_u of a 2D signal x is characterized by a 2×2 matrix Λ and:

$$x_u[n] = \begin{cases} x[\Lambda^{-1}n], & \text{if } \Lambda^{-1}n \text{ is in the integer lattice of points} \\ 0 & otherwise. \end{cases}$$

A simple upsampling operation would be to double both dimensions of an image x and copying its pixels in x_u such as $x_u[n_1, n_2] = \begin{cases} x[\frac{n_1}{2}, \frac{n_2}{2}] & \text{if } n_1 \text{ and } n_2 \text{ are even,} \\ 0 & otherwise. \end{cases}$ The resulting image would have three times more zeroes than other values. Therefore, a downsampler followed by an upsampler lose information. A solution to get the upsampled image nice looking is to do *interpolation*. The Bamberger filter bank do not need interpolation because by combining the upsampled signals of all subbands, we get back all the information. However, upsampling and then downsampling a signal with the same matrix ($\Lambda = \mu$) returns the input signal[9].

Resampling

Resampling [2, 22] consists in rearranging the organisation of the values of a signal. It is actually no more than up or downsampling by an unimodular matrix R ($|R| = \pm 1$). An example of resampling operation is shown in Figure 2.17.

[9]because $x_d[n] = x_u[\mu n] = x[\Lambda^{-1}\mu n] = x[\mu^{-1}\mu n] = x[n]$. Note that we escape the zeroes because we know that $\Lambda^{-1}\mu n$ is in the integer lattice of points.

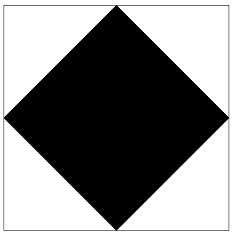
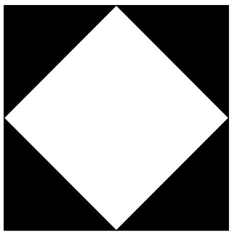
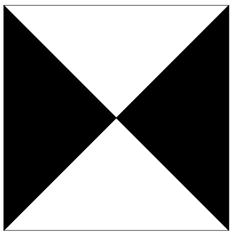
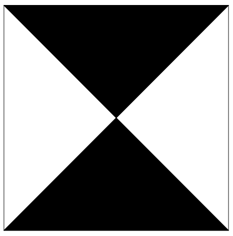
Diamond filter pair		Hour-glass filter	
			
(a)	(b)	(c)	(d)

Figure 2.18: Ideal Frequency Response for Diamond and Hour-glass Filters.

2.3.2 Introduction to the Bamberger Directional Filter Bank (DFB)

The Bamberger DFB can separate an image into a power of two of subbands, using the wedge shape passband regions shown by the spectrum partitioning of Figure 2.14. This can be achieved by cascading 2-band filter banks whose partitioning is the one of 2.14 (a). Figure 2.18 (c) and (d) shows the two hourglass filters used in this 2-band filter bank.

These two filters are not implemented directly. Instead, it is equivalent to shift the signal by π in the *horizontal* direction of the *frequency* domain and filter it with the diamond filters shown in Figure 2.18 (a) and (b). We shall see later how to design the diamond filter pair. The frequency shift is expressed by $(\omega_1, \omega_2) \longrightarrow (\omega_1 - \pi, \omega_2)$, which correspond to multiplying the signal by $e^{-i\pi n_1}$ in the time domain[10]. Note that $e^{-i\pi n_1} = (-1)^{n_1}$ so the frequency shift is as simple as multiplying by -1 every odd column of x. Figure 2.19 shows the building block of a N-band DFB. (a) and (b) are equivalent.

Cascading

The original filter bank introduced by Bamberger and Smith [1] needs another 2-band filter bank building block with different passband in order to cascade the filters. In [24], Park, Smith and Mersereau introduce a new cascading structure which uses only the 2-band filter bank with hourglass passband. In [3], the authors (including Smith) use a slightly modified version which again uses only the hourglass filter bank. In Figure 2.20, we show a complete process of an height-band

[10]If $X = \mathcal{DFT}(x)$ is the Fourier transform of x, then a frequency shift by (θ_1, θ_2) is expressed by $X(\omega_1 - \theta_1, \omega_2 - \theta_2) = \mathcal{DFT}(x(n_1, n_2)e^{i\theta_1 n_1 + i\theta_2 n_2})$. It correspond thus in the time domain by a multiplication of the signal by $e^{i\theta_1 n_1 + i\theta_2 n_2}$

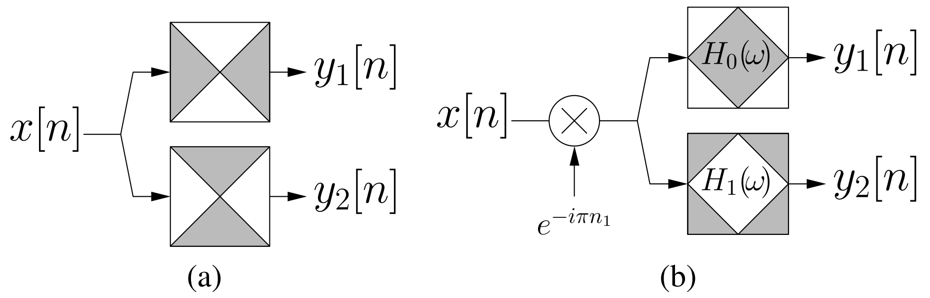


Figure 2.19: Analysis Part of a 2-Band DFB Using (a) Hour-glass and (b) Diamond Filters. (b) needs a frequency shift to be equivalent to (a).

DFB. In order to use the same 2-band filter bank for every stage, a resampling operation must be done between stages to reorganize information. The resampling operations are represented by a box containing the name of the resampling matrix used. The resampling matrices R_i used are shown in Table 2.4. The downsampling matrix used is $Q_1 = \mu = \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}$.

Table 2.4: Resampling Matrices. Taken from [24, 22].

$$R_1 = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \quad R_2 = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \quad R_3 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \quad R_4 = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}$$

Diamond Filter bank

The diamond Filters are not applied directly but on the downsampled images. As shown in Figure 2.16, the downsampling matrix considered also rotates the images counterclockwise by 45 degrees ($\pi/4$ radians). Because of this rotation, the filters to apply now have a checkerboard passband, as shown in Figure 2.21. This two filters are *separable*. This mean that we can apply first an horizontal filter and then a vertical filter. Separable filters have less computation complexity than non separable filters. We use the notation $H(\omega_1)$ and $H(\omega_2)$ to designate a filter on the rows and on the columns, respectively.

Polyphase Filter Bank

The Bamberger DFB use a polyphase structure similar to the 1D polyphase filterbank described in 2.1.3 and summarized in Figure 2.5. The signals are downsam-

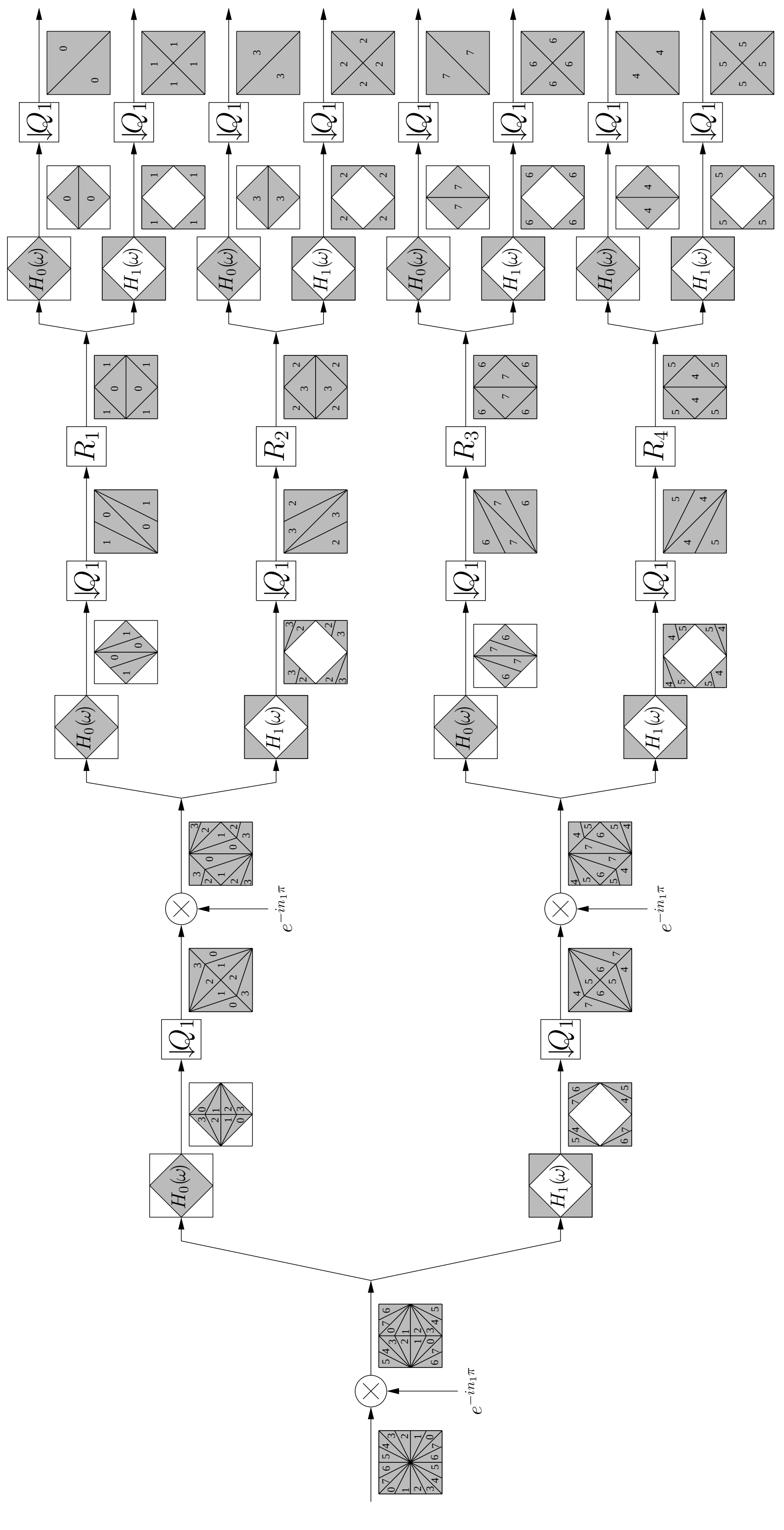


Figure 2.20: (a) First Stage, (b) Second Stage, and (c) Third Stage of an Eight-band DFB. Adapted from blocks in [3, 22] and an other tree structure in [2, page 26]. The resampling matrices are shown in Table 2.4. Q_1 is the quicunx downsampling matrix used in the filter bank.

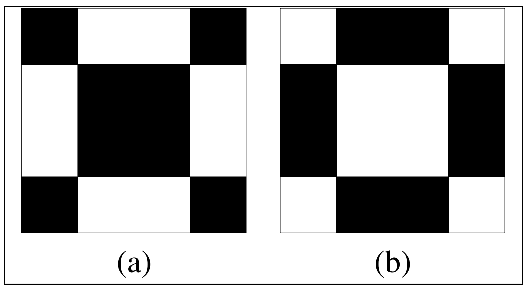


Figure 2.21: Ideal Frequency Response for Checkerboard Filters.

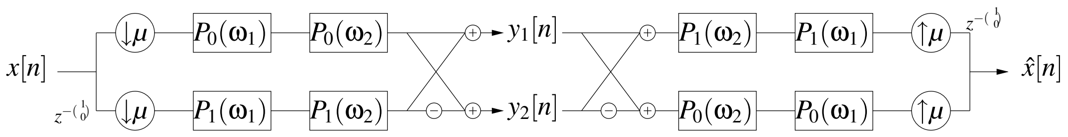


Figure 2.22: 2D Polyphase DFB

pled first, then filtered, and then combined to produce the two subbands, as shown on the analysis part of Figure 2.22. In the 1D case, we took the even samples on the first band and the odd samples on the second band. In the 2D case, a delay of one pixel is applied on the second band, which is represented by $z^{-(1,0)^T}$. The analysis and synthesis parts of the 2-band Bamberger filter bank is shown on Figure 2.22. The filters P_0 and P_1 are the 1D filters presented with the 1D polyphase filter bank in 2.1.3. Bamberger explains in [1, section IV] why using the filters P_0 and P_1 along with the downsampling and combine operations produces the expected subbands. It can be noticed that unlike in the 1D case, there is no straight forward non-polyphase filter bank equivalent. Such a filter bank would need to implement the diamond filters directly.

The polyphase filter bank described here has been implemented in standard `C` and partly in `CUDA`. Implementation issues of the 2-band Bamberger filter bank presented here are detailed in 4.2.

Chapter 3

GPGPU

GPGPU is a short-hand notation for *General-purpose computing on graphics processing units*. The idea is to use the computation power of the GPU to execute non-graphical computation. It can significantly improve the computation time of a program, including removing CPU load. What makes using GPUs for general computing possible and worthwhile are their performance, price, availability, bandwidth, memory, and architecture.

We will first have a look at a typical GPU architecture, then see how to program them.

3.1 GPU

The GPU, or *Graphics Processing Unit* is a chip whose aim is to help the CPU handle graphic-related computations. It is often on a computer graphic card and has its own memory and a direct access to the video memory where the computer screen frame buffer is.

GPUs used to be highly specialized processors. They have evolved to implement 2D primitives in hardware in the 1980's and 3D in the 1990's. They also offered more and more functions, as OpenGL and DirectX evolved, and became quicker. Currently, GPUs are performant for many applications including graphics rendering, 3D imaging and visualization, games, vector graphic viewers, Adobe Postscript, PDF, and Flash, Video and audio coding and manipulation, numerical simulation, virtual reality, and even less obvious applications such as stock options pricing determination.

3.1.1 GPU Architecture

Internally, a GPU differs from a traditional processor in that it contains many Floating Point Units (FPU), fewer logic, and very small caches. The architecture is designed to allows many threads to run at the same time in a SIMD fashion (Single Instruction, Multiple Data), for example one per pixel of the screen. GPUs are also designed to have a good bandwidth between their memory and their computing units.

Two GPU manufacturers dominates the market: NVIDIA and AMD (through their purchase of ATI). GPU evolution, which has long been driven by computer gaming, has now evolved to focus on other markets as well. Until recently, there were two types of processors in a GPU, specialised for vertex and fragment processing respectively. These architectures are adapted to the *graphic pipeline*, that is the different computational steps needed to compute the color of a pixel on the screen or a frame buffer. Larsen gives in [25] a very good introduction to GPU history and non-unified GPU architecture. NVIDIA introduced the concept of unified architecture in November 2006 with its GeForce 8 series. They only have one type of processor, sometime referred to as a *unified shader* or a stream processor. The GeForce 8800 has 128 processors, which is four times more than previous generation GPUs from NVIDIA [25]. The unified architecture improves performance both by giving the same features available for fragment and vertex shaders, and by allowing a better utilization of the cores. Additionally, it comes along with a new programming model as we shall see in next section.

3.1.2 GPU Programming

GPUs are traditionally difficult to program, and that is changing. Before, the way to program GPUs was to use a graphic API whether the program was intended for graphic purpose or not. People used a shading language (Cg, HLSL, OpenGL Shading Language) along with OpenGL or DirectX. Tricks to do general-computing on non-unified architecture are given in [26]. We also recommend *A Survey of General-Purpose Computation on Graphics Hardware* [27] from Owens et al., published in 2005.

High level APIs or programming languages designed specifically for general-purpose computing have been developed:

Microsoft Research Accelerator Project [28] provides a high-level data-parallel library to simplify GPU programming.

Brook [29] is a stream programming language extended from C and designed for general programming on GPU. BrookGPU is a compiler and runtime implementation of Brook. This project is lead by Stanford University.

Sh [30] is a library to program GPUs with hardware abstraction. The project was lead by University of Waterloo and is no longer maintained. Sh is RapidMind predecessor.

RapidMind [31] is a platform that makes easy to program in a multithreaded environment. They claim to have good performance over many hardware components: modern NVIDIA GPUs (over CUDA) and ATI graphic cards, IBM CellBE, AMD and Intel CPUs. RapidMind's two strengths are to ease multithreading programming and provide hardware portability, at the cost of being dependant of their platform. The programming model uses three basic types: Values (floats, integers), Arrays (1D, 2D, 3D), and Programs (Kernels, working on arrays). It uses the data-parallel model, which they call *SPMD*[1] *stream programming model.*

Larrabee [32] is a project of processor by Intel. Michael Feldman, HPCwire editor, described [33] Larrabee processors as:

> *"a manycore (i.e., more than 8 cores) device [...] based on a subset of the IA instruction set with some extra GPU-like instructions thrown in."*

Larrabee processors may be released in 2009.

Close to Metal [34], also known as *CTM*, is a hardware interface for stream processing developed formerly by ATI and now by AMD.

Cell Broadband Engine [35], or *CellBE*, is a processor from IBM which include a PowerPC core called PPE (Power Processor Element)and height RISC cores called SPE (Synergistic Processing Elements). Cell has shown[36] good performances for scientific applications. It is regarded as powerful but difficult to program.

CUDA [37] (Compute Unified Device Architecture) is a new programming interface that lets users program NVIDIA GPGPUs in a C-like fashion for data parallel intensive computation. The programmer has access to on-chip memory and run a thousands of kernel into threads. CUDA will be introduced in more details in the next section.

Some of these technologies are moving from GPU to CPU, and others takes the inverse path. NVIDIA, AMD, Intel and IBM each come with a complete *hardware + software* platform, whereas BrookGPU, Sh and Rapidmind are hardware independent. Among these technologies, some are still immature (MRAP,

[1]Single Program, Multiple Data

Larrabee, CTM) or outdated (Sh). Two pure software solutions and two complete solutions remain. CUDA is used in this project because it is easy to program and because NVIDIA cards are the most commons.

3.2 CUDA

CUDA is NVIDIA's answer to the GPGPU trend. It is a software framework running on GeForce 8 series and later NVIDIA hardware. The aim is to ease programming GPGPUs and open new markets. Because some clients are interested in the performance of GPUs for other purpose than printing images on the screen, NVIDIA has released a set of cards called *Tesla* which are virtually graphic cards with powerful GPUs, but without screen plugs. They are designed only for scientific computing. CUDA is also usable on GeForce 8 graphic cards and QuadroFX 4600/5600 graphic cards. CUDA is a young technology and will probably gain maturity over the time.

GPUs traditional task, graphic rendering, is a highly parallel, compute-intensive task. Since the control and cache parts take about half the surface of a CPU, they can be greatly reduced on a GPU, therefore leaving place for computation cores. These cores are actually FPUs (Floating Point Unit), so they are small, and there can be many on the same chip. A GPU has many (128) small FPUs, and very few logic and cache, whereas a CPU has only a few (4) cores, a huge cache and control part. Finally, GPUs have ten times more computational power and ten times more memory bandwidth than typical CPUs [38, page 15].

CUDA capable devices, such as the *NVIDIA GeForce 8800 GTX* used in this project have cores that are grouped by eight to form *multiprocessors*. A GeForce 8800 GTX has 16 multiprocessors, so $8 * 16 = 128$ cores. The 8 cores of a Multiprocessors share 2^{13} 32-bit-registers (average of 1KB/core), 16KB of *shared memory*, and two 8KB *caches* for constants and textures. A multiprocessor has one *Instruction Unit* and works in a SIMD[2] manner. All multiprocessors share the *Device Memory*. Figure 3.1, taken from the CUDA programming guide [39], shows a CUDA device (Graphic card or Tesla) architecture.

The CUDA programming guide [39] is a good document to learn programming using CUDA. Examples of programs are provided with the CUDA SDK.

[2]Single Instruction, Multiple Data

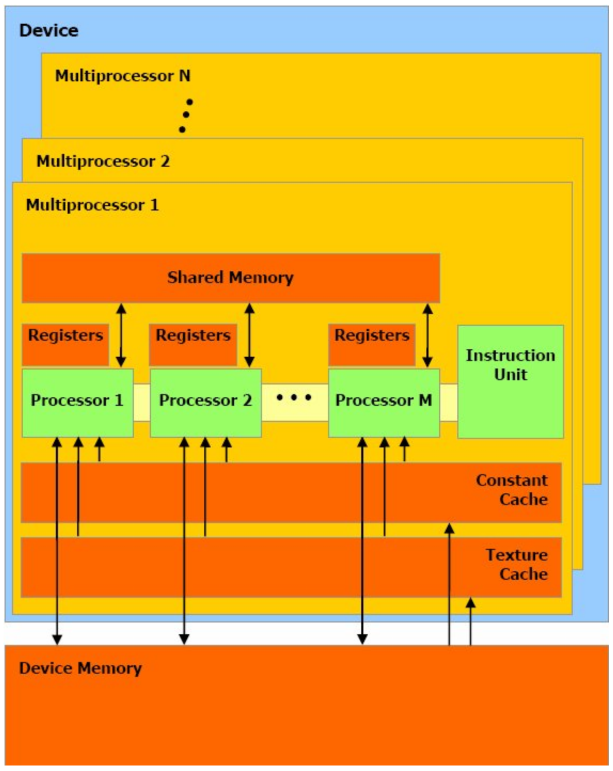


Figure 3.1: CUDA 1.x Capable Device Architecture. This Figure comes from the CUDA programming guide [39] and is used with permission. For the GeForce 8800 GTX, $N = 16$ and $M = 8$.

Chapter 4

Implementation Issues

This chapter presents implementation issues of filters, and describes step by step how to implement the 2-band Bamberger DFB from a computer-scientist point of view. The tools and programs developed during this thesis are then introduced.

4.1 Filter Implementation Issues

4.1.1 Using Periodic Extention

Let us consider a 1D signal. When convolving [13] it, the output signal is wider than the input signal, by $FILTER\_SIZE - 1$. This is rather inconvenient for several reasons. First, it takes more disk place to store the result and more memory to process the signal. The performance might suffer, especially in a tree structure when the signal is getting bigger and bigger at every stage. Additionally, it makes programming harder. In the 2D case, the same problem occurs. In addition to the 1D problems, maximal decimation ease visual subband analysis and some application like image classification [24]. The solution is to consider the periodic extension of the signal, because convolving a periodic signal gives a periodic signal with the same period. If we consider only one period, the size remain constant.

4.1.2 Filter Coefficients

In this thesis, we have used the filter coefficients of Tables 4.1 and 4.2. They come from [19] and represent a half band low pass filter. We use them for $h_0[n]$ and compute the other filter coefficients from h_0, as described in Chapter 2.

Table 4.1: QMF Filter Bank Coefficients of Length 8: copy from Table 5.5 in [19]

$.93871500e-02$
$-.70651830e-01$
$.69428270e-01$
$.48998080e+00$
$.48998080e+00$
$.69428270e-01$
$-.70651830e-01$
$.93871500e-02$

Table 4.2: QMF Filter Bank Coefficients of Length 16: copy from Table 5.5 in [19]

$.10501670e-02$
$-.50545260e-02$
$-.25897560e-02$
$.27641400e-01$
$-.96663760e-02$
$-.90392230e-01$
$.97798170e-01$
$.48102840e+00$
$.48102840e+00$
$.97798170e-01$
$-.90392230e-01$
$-.96663760e-02$
$.27641400e-01$
$-.25897560e-02$
$-.50545260e-02$
$.10501670e-02$

4.2 Complete Bamberger DFB Algorithm

This section present step by step the path taken by a 2D signal in the Bamberger filter bank. We will follow the path a matrix takes within the DFB.

The image is represented by a matrix *matA*1. For this example, let us use the following matrix:

$$matA1: \begin{array}{c|c|c|c|c|}
\cline{2-5}
3 & \mathrm{M} & \mathrm{N} & \mathrm{O} & \mathrm{P} \\ \cline{2-5}
2 & \mathrm{I} & \mathrm{J} & \mathrm{K} & \mathrm{L} \\ \cline{2-5}
1 & \mathrm{E} & \mathrm{F} & \mathrm{G} & \mathrm{H} \\ \cline{2-5}
0 & \mathrm{A} & \mathrm{B} & \mathrm{C} & \mathrm{D} \\ \cline{2-5}
\multicolumn{1}{c}{} & \multicolumn{1}{c}{0} & \multicolumn{1}{c}{1} & \multicolumn{1}{c}{2} & \multicolumn{1}{c}{3}
\end{array}$$

We do a frequency shift of π. Since

$$x(n_1,n_2)e^{j\theta_1 n_1 + j\theta_2 n_2} \leftrightarrow X(\omega_1-\theta_1,\omega_2-\theta_2)$$

the frequency shift corresponds to multiplying each value in the matrix by $e^{jn_1\pi} = (-1)^{n_1}$.

Our matrix thus becomes:

$$matA2 = \begin{pmatrix} M & -N & O & -P \\ I & -J & K & -L \\ E & -F & G & -H \\ A & -B & C & -D \end{pmatrix}$$

In order to downsample, we consider the periodic extension of the image:

$$matA2b = \begin{array}{cc|cccc|ccc}
G & -H & E & -F & G & -H & E & -F & G \\
C & -D & A & -B & C & -D & A & -B & C \\ \hline
O & -P & M & -N & O & -P & M & -N & O \\
K & -L & I & -J & K & -L & I & -J & K \\
G & -H & E & -F & G & -H & E & -F & G \\
C & -D & A & -B & C & -D & A & -B & C \\ \hline
O & -P & M & -N & O & -P & M & -N & O \\
K & -L & I & -J & K & -L & I & -J & K
\end{array}$$

matA2b is periodic with periodicity matrix [23, 13]: $\begin{pmatrix} 4 & 0 \\ 0 & 4 \end{pmatrix}$.

4.2.1 Downsampling

We use downsampling matrix $\mu = \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}$ whose inverse is $\mu^{-1} = \frac{1}{2}\begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}$.

After downsampling, *mat*3.1 and *mat*3.2 are periodic. Different periodicity matrices are possible, and we have the choice of which fundamental period to keep for filtering. The fundamental period should be rectangular. In the example, we can chose a 2×4 or a 4×2 fundamental period. We keep only one period because of implementation limitation, but in theory, the filter is done on the infinite plan. Therefore, we will take 4×2 period for horizontal filtering and the 2×4 period for vertical filtering. This is possible because the filters do not change the periodicity.

The 4×2 fundamental period corresponds to the following periodicity matrix: $\begin{pmatrix} 4 & 2 \\ 0 & 2 \end{pmatrix}$.

The downsampling operation appears to be the only step that limits the algorithm to square images. Downsampling non-square images works fine, but their is not a possible rectangular fundamental period in both directions[1].

4.2.1.1 LPF

We downsample $matA2$ with μ: $matA3.1(n) = matA2(\mu n)$.

The result is:

$matA3.1$:

C	$-H$	I	$-N$	C	$-H$	I	$-N$	C	$-H$
$-P$	A	$-F$	K	$-P$	A	$-F$	K	$-P$	A
I	$-N$	C	$-H$	I	$-N$	C	$-H$	I	$-N$
$-F$	K	$-P$	A	$-F$	K	$-P$	A	$-F$	K
C	$-H$	I	$-N$	C	$-H$	I	$-N$	C	$-H$
$-P$	A	$-F$	K	$-P$	A	$-F$	K	$-P$	A

$matA4.1$:

$-H$	I	$-N$	C
A	$-F$	K	$-P$

4.2.1.2 HPF

For the high pass filter, we apply the z^{k_1} transformation, which k_1 being a coset vector of μ. If we take the first coset vector $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$, we get:

$matA3.2$:

$-D$	E	$-J$	O	$-D$	E	$-J$	O	$-D$	E
M	$-B$	G	$-L$	M	$-B$	G	$-L$	M	$-G$
$-J$	O	$-D$	E	$-J$	O	$-D$	E	$-J$	O
G	$-L$	M	$-B$	G	$-L$	M	$-B$	G	$-L$
$-D$	E	$-J$	O	$-D$	E	$-J$	O	$-D$	E
M	$-B$	G	$-L$	M	$-B$	G	$-L$	M	$-G$

[1]If the input image has size $W \times H$, then its periodic extension has periodicity matrix $P = \begin{pmatrix} W & 0 \\ 0 & H \end{pmatrix}$. Then the downsampled signals have periodicity matrix $\mu P = \begin{pmatrix} W & -W \\ H & H \end{pmatrix}$.

$$matA4.2 : \boxed{\begin{array}{rrrr} E & -J & O & -D \\ -B & G & -L & M \end{array}}$$

4.2.2 Filtering

When filtering, we apply the filter on one period of the downsampled images. Because we take periodicity into account, the output has the same periodicity as the input.

The output of horizontal filtering would hence give two matrices of the form:

$$matA5.1 : \boxed{\begin{array}{rrrr} m & n & o & p \\ i & j & k & l \end{array}} \text{ and } matA5.2 : \boxed{\begin{array}{rrrr} e & f & g & h \\ a & b & c & d \end{array}}$$

Since both of these matrices have periodicity matrix $\begin{pmatrix} 4 & 2 \\ 0 & 2 \end{pmatrix}$, we can also use other fundamental period (see 4.2.1). For vertical filtering, we use a vertical fundamental period and will apply the filter on:

$$matA6.1 : \boxed{\begin{array}{rr} o & p \\ k & l \\ m & n \\ i & j \end{array}} \text{ and } matA6.2 : \boxed{\begin{array}{rr} g & h \\ c & d \\ e & f \\ a & b \end{array}}$$

And get the result :

$$matA7.1 : \boxed{\begin{array}{rr} oo & pp \\ kk & ll \\ mm & nn \\ ii & jj \end{array}} \text{ and } matA7.2 : \boxed{\begin{array}{rr} gg & hh \\ cc & dd \\ ee & ff \\ aa & bb \end{array}}$$

4.2.3 Combining

The last step is to combine them so that they represent directional information: $matA8.1 = matA7.1 + matA7.2$ and $matA8.1 = matA7.1 - matA7.2$.

Finally, we get two matrices which represent each information in one of the two hourglass region:

$$matA8.1 : \boxed{\begin{array}{rr} OO & PP \\ KK & LL \\ MM & NN \\ II & JJ \end{array}} \text{ and } matA8.2 : \boxed{\begin{array}{rr} GG & HH \\ CC & DD \\ EE & FF \\ AA & BB \end{array}}$$

4.3 Tools

We developed several tools specifically for this project. Among these is an image manipulation library called `GRIM`. This application can be used by including the header file `grim.h` and linking to the static library `libgrim.a`. The name *GRIM* comes from the fact that it manipulates *GRey IMages*. We have also developed a program called `grim2x` that converts image files between different formats. This section describe these tools.

4.3.1 Image Representation

The images manipulated are grey images. A pixel is represented by a number between 0 (black) and 255 (white).

In a program

The simplest way to represent a 256 grey level image in a `C` program is to create a `C` structure as in Figure 4.1, and store in it the dimensions of the image and an array of pixel. Because we need more precision than 256 levels when processing the images, we will use a similar structure using an array of float. These two structures are called *grim* and *fgrim* and defined in Figure 4.1.

```
typedef struct {
  int   w;      /* width  of image */
  int   h;      /* height of image */
  bit8 *pix;   /* array of pixels */
} grim;

typedef struct {
  int   w;      /* width  of image */
  int   h;      /* height of image */
  float *pix; /* array of pixels */
} fgrim;
```

Figure 4.1: `grim` and `fgrim`: Two Structures to Represent an Image. `bit8` is defined as a `char`. It could be a `uint8_t` (defined in `stdint.h` in `C99`)

As a file

We also need file formats to store and manipulate images.

We have created our own format, called *grim*, to store and load images. This format is a direct extension of the `grim` structure of Figure 4.1.The width and then height of the image are stored as 32-bit unsigned integers, using little endian representation. Starting just after, at the ninth byte of the file, the pixel array is dumped in the file. Each pixel is represented by an unsigned 8-bit integer and they are stored line by line. Files have a *.grim* extension.

Because we would like to exchange images with the rest of the world, we also use the *BMP* (bitmap) format [40], also known as *DIB* (Device Independant Bitmap). There are actually several different BMP file formats, and they contain information in their header to determine which representation they use. We use the *Windows V3* version to be able to read files created by the `GIMP` software. We use the *.bmp* extension for BMP files. A BMP file begins with a 14-bytes file header, then a DIB header containing information about the image and the way it is represented. The DIB header is 40-bytes long in the Windows V3 format. In it, we indicate in a field that we use a 256 color representation. After the DIB header comes a palette, where the 256 colors are defined in the RGB format. In the palette, we indicate that color *i* in the pixel array represent the color RVB(*i*,*i*,*i*). Then, starting at byte 1078 ($14+40+256*4$) is the bitmap data. Each pixel is represented by a 8-bit unsigned integer. The total length of a file is then $(1078 + width * height)$ bytes. As this format is pretty easy, the reader can refer to our implementation (see file `grim.c`) to understand it, and use an hexadecimal editor to analyse images.

For debugging purpose, it is nice to print the numbers of an image. We print on a first line the width, a space, and the height of the image. Then, we print every image line on a new line, and pixel numbers are separated by spaces. When printed in a text file, we give it a *.grtxt* extension.

We have used the `octave` software for numerical computing (see Section 4.4). To exchange images with octave, we have used a specific text format. An example is given in Table 4.3.

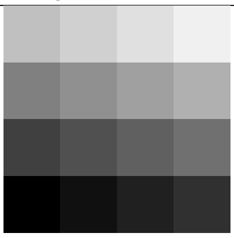
Note that in an image, lines are from bottom to top (i.e.: pixel $(0,0)$ is at the lower left corner) whereas in memory and text representations, lines are from top to bottom (i.e.: pixel $(0,0)$ is at the upper left corner).

Transform Between Formats

The `GRIM` library provides a set of `C` functions to transform image formats. Note that there exists many graphic libraries available that allow to manipulate images, and in particular BMP images. Because we need only a few functions, we have found simpler to write our own small library. This also allows us to use multiple platforms without installing large libraries. Among graphic libraries,

Table 4.3: Image Representation

image (BMP)	text (file.grtxt)	file.mat (octave)

	`4 4`
	`  0  16  32  48`
	` 64  80  96 112`
	`128 144 160 176`
	`192 208 224 240`
		`# Created by hand`
		`# name: by16_4x4`
		`# type: matrix`
		`# rows: 4`
		`# columns: 4`
		`  0  16  32  48`
		` 64  80  96 112`
		`128 144 160 176`
		`192 208 224 240`

we could have use `GTK+` (glade, Glib, Gazpacho) [41], `ImageMagick`[2] [42] or `GraphicsMagick` [43][3], `imlib` [44], `DevIL` [45]. Most of them are released under the *GNU Lesser General Public License* (LGPL) [46], which allows to link to the licensed library without restricting the program license[4].

Using the `GRIM` library, internal representation of images can be converted with the functions:

```
int  grim\_to\_fgrim(grim *in, fgrim *fout);
int fgrim\_to\_grim(fgrim *fin, grim *out);
```

Input output in different formats are done by the functions:

```
int  grim_write(grim *img, FILE *fpout);
int fgrim_write(fgrim *fimg, FILE *fpout);
int  grim_print(grim *img);                     // to stdout
int fgrim_print(fgrim *fimg);                   // to stdout
int  grim_write_bmp(grim *img, char* bmp);      // (&img, "img.bmp");
int  grim_read_bmp(char* bmp, grim *img);       // ("img.bmp", &img);
int  grim_write_grim(grim *imgin, char* fileout);// (&img, "img.grim");
int  grim_read_grim(char* filein, grim *imgout); // ("img.grim", &img);
int  grim_text_read(char* filein, grim *imgout); // ("img.grtxt",&img);
```

[2]`ImageMagick` is a software suite rather than a library, and uses a GPL compatible licence. MagickWand and MagickCore are tools and libraries to convert, compose, and edit images from C programs.

[3]`GraphicsMagick` is a fork of `ImageMagick`.

[4]However, changes of the library itself must be made public.

The two following functions draw the curve corresponding to one row of the input signal. It is typically used to visualise in 2D a 1D signal. The horizontal axis is the column index in the row (n) and the vertical axis is the value of the pixel ($x[n]$).

```
int  grim_line_to_2d(grim  *fin, int line, grim *out_2d);
int fgrim_line_to_2d(fgrim *fin, int line, grim *out_2d);
```

The `grim2x` program permits to transform images between different file formats. It uses the `GRIM` library.

```
 * Description:
    Convert grey image in different formats :
     * GRIM  image.grim
     * BMP   image.bmp
     * text  image.grtxt
 * Format:
    grim2x [-h] [-o <image_out>] <conv> <image_in>
     -h :        print help and exit
     <conv>      the conversion to make: one of "grim2bmp", "bmp2grim",
                 "grim2grtxt", "grtext2grim", "grim_print", "bmp_print"
     <image_in>  is the name of input image to convert
     <image_out> is the name of output converted image.
                 Default is out.grim, out.grtxt, out.bmp, or stdout
```

4.3.2 Image Manipulation

The following tools are functions of the `GRIM` library.

Basic Manipulation

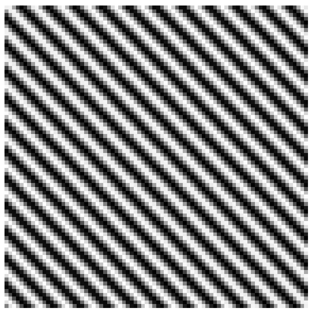
```
int  grim_create(grim *img, int width, int height);
int  grim_free(grim *img);
int  grim_copy(grim *in, grim *out);
int fgrim_create(fgrim *fimg, int width, int height);
int fgrim_free(fgrim *fimg);
int fgrim_copy(fgrim *in, fgrim *out);
```

Image Comparison, Sizing, and Sinus

```
int  grim_diff(grim *im1, grim *im2);       // difference in %
float fgrim_diff(fgrim *fim1, fgrim *fim2); // difference in %
```

```
grim g;
int w = 80;
int h = w;
grim_create(&g, w, h);
grim_set_sinus(&g, WHITE, 1*PI/4.0, 5);
grim_write_bmp(&g, "sinus_image.bmp");
grim_free(&g);
```

(a)



(b)

Figure 4.2: Example of Use of `fgrim_set_sinus`: (a) code and (b) result

```
int  grim_cmp(grim *im1, grim *im2);        // return 1 iff equals
int  grim_scale(int fact, grim *in, grim *out); // scale by fact
int  grim_set_color(grim *img, bit8 color); // set all pixels to color
int  grim_set_sinus(grim *img, bit8 color, float direction,int period);
int fgrim_set_sinus(fgrim *fimg,bit8 color,float direction,int period);
int  grim_resize(grim *in, grim *out, int new_w, int new_h); //crop/pad
```

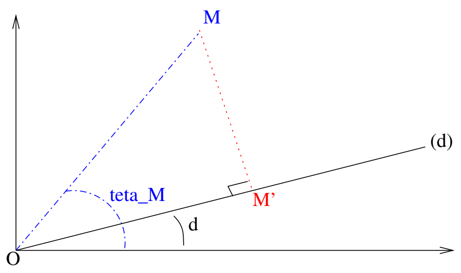
Figure 4.3 explains how the function `fgrim_set_sinus` works and Figure 4.2 shows how to use it.

4.4 Using `octave`

`Octave` has been used to produce processed images, as a proof of concepts, and to plot graphics (such as the 3D sinus of Figure 2.7 (a) page 16 and the 2D sinc functions of Figure 2.8).

`Octave`, also known as `GNU Octave`, is a numerical computing software. It is free[5] and open-source, since it uses the GPL Licence. `Octave` has a scripting language highly compatible with `Matlab` and offer similar services. It can be used with `gnuplot` (and many other plotting software) to produce 2D or 3D graphics in many formats including the `Xfig` format. It has many possibilities like libraries,

[5]`Octave` is free both as in free speech (freedom) and as in free beer (price). See the Free Software Foundation (FSF) definition of free software: `http://www.gnu.org/philosophy/free-sw.html`



```
d   = [angle] direction
O   = [point] origin: (0,0)
(d) = [line ] line through O with direction d
M   = [point] (i,j)
M'  = [point] orthogonal projection of M on (d)
teta_M  = [angle] angle of point M in polar coordinates = atan(j/i)
M_norm  = [distance] distance between O and M = sqrt( i^2 + j^2 )
Mp_norm = [distance] distance between O and M'= M_norm * cos(teta_M-d)
Color of point M = 255/2 * ( sin(Mp\_norm * 2 PI / period)+1 )
```

Figure 4.3: Explanation of `fgrim_set_sinus`

debuggers, etc... most of the time through extensions (sometime compatible with `Matlab`).

We have used a text format to exchange images outside the software. The method is presented in Figure 4.4. Inside `Octave`, an image is represented by a matrix.

We have chosen to use `Octave` because it is easy to use, and its scripting language[6] gives, for instance, the resulting numbers of a convolution without the need for compilation and printing tools. We have used it as a proof of concepts and to adjust the filter offsets. Used with small matrices, the computation steps can be verified by hand or the numbers can be tracked across the operations, for example to see how a downsampling operation reorganize the matrix and if an upsampling operation is the exact inverse of a downsampling. If big matrices are used (we provide some in *.mat* files), it is convenient to disable the output of the intermediate results by ending the function calls by a `;` (semicolon).

Each function is in a file with a *.m* extension. They must be loaded (e.g.: `source('downsample.m');`) before to be used (e.g.: `[matA4b1,matA4b2] = downsample(matA2);`). The files `dfb.m`, `rev_dfb` and `main1.m` load and use several functions. The complete filter bank is not implemented in octave because we have stopped using it when we have understood all the concepts. The only

[6]`octave` is both the name of a software and of the scripting language it uses.

```
shell$ octave
octave:1> a = [1 2;3 4]
a =

   1   2
   3   4

octave:2> save -text a.mat a
octave:3> exit
shell$ cat a.mat
# Created by Octave 2.9.12, Sun Jan 06 14:27:28 2008
   CET <jerome@woobo>
# name: a
# type: matrix
# rows: 2
# columns: 2
 1 2
 3 4
shell$ octave
octave:1> load('a.mat');
octave:2> a
a =

   1   2
   3   4

octave:3>
```

Figure 4.4: A Method to Export and Import an Image with `Octave`

steps missing is using a different offset for filtering the second time and using proper filter coefficients.

However, `Octave` is slow, like its competitors. This is because it uses an interpreted language, uses general tools rather than specialized, do not take advantage of specific hardware (The version we have used has been compiled for any x486 processor). `Octave` passes function arguments by copy. When the argument is a matrix, it is easy to understand that it is not the right tool to use when seeking performance. Some compilers exist, like there exist Matlab or Java compilers that produce assembly (or C) code, but it is not the primary aim, and `Octave` remains a very high level language that cannot compete with other technologies. If it served us during this project, we moved on and now use C and CUDA.

4.5 Using `C`

This section describes the structure of our `C` implementation of the three filter banks developed during the thesis: 1D non-polyphase ands polyphase FB and the Bamberger DFB.

The code has been designed to be portable. It has been tested on Linux Ubuntu 7.10 32 bits, Linux Ubuntu 7.10 64 bits, and Windows XP using `Cygwin`. Early versions of the `GRIM` library and the 1D non polyphase filter have been tested on Windows XP using `Visual C++`.

Figure 4.5 summarizes the algorithms implemented. The name of the signals throughout the process are indicated. For example, in Figure 4.5 (a), the filter $H_0(w)$ transform the signal *E1* into *E2b1*. Signals of the first and second band of the filter-banks have the suffix *b1* and *b2* respectively. In the code, the signal have the prefix *s* and are of the type `fgrim` describe previously (see Figure 4.1). For example, signal *C5b1* is declared by `fgrim sC5b1;`. Filters operate in place on the signals. Signals *C3b1* and *C4b1* are therefore represented by the same variable, called `sC3a4b1` (*a* for *and*). In (a) and (b), signals have a width of 1 (they are 1D signals). In (c), signals are square.

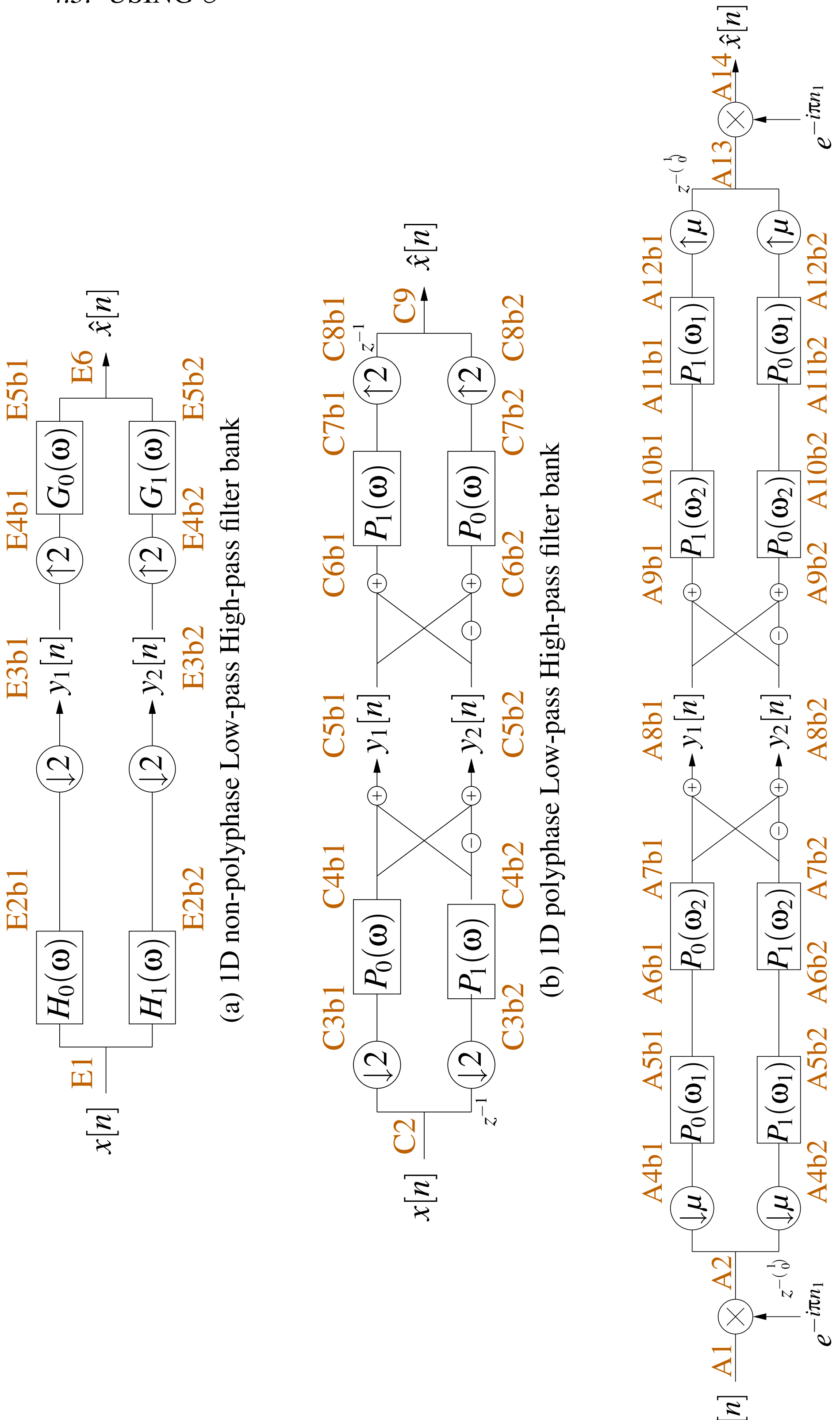
The three algorithm implemented and described in Figure Figure 4.5 (a), (b), and (c) are:

(a) 1D non-polyphase Low-pass High-pass filter bank – Implemented in `1d_filter.c`. If the macro `NODOWNUP` is defined during the compilation, the downsampling and upsampling operation are skipped. This filter bank has been used to ensure the validity of its polyphase version.

(b) 1D polyphase Low-pass High-pass filter bank – Implemented in `1d_polyphase_filter.c`. By achieving perfect reconstruction and giving exactly the same intermediate results than its non-polyphase version, this filter bank permits to validate the concepts of implementation issues related to QMF filters (see 2.1.3) and polyphase FB. In particular, it validates the coefficients for $P_0(\omega)$ and $P_1(\omega)$.

(c) 2D 2-band polyphase hourglass filter bank (Bamberger DFB) – Implemented in `2d_polyphase_filter.c` and called by the `main()` function in `dfb.c`. The difference between the signals *A5b1* and *A6b1* is that the later is a vertical tile of the periodic extension generated by the former. In other word, it is reorganized from horizontal to vertical, as described previously in 4.2.1. Appendix B describes the program structure and lists some important parts of the code.

Most functions used by the three programs are implemented in `filter.c`. Emphasis have been put on the clarity of the code rather than performance. The



(a) 1D non-polyphase Low-pass High-pass filter bank

(b) 1D polyphase Low-pass High-pass filter bank

(c) 2D 2-band polyphase hourglass filter bank (Bamberger DFB)

Figure 4.5: Filter Bank Implementation: Algorithm and Name of Intermediate Signals. These three schemes represent the same algorithms than Figures 2.4, 2.5 and 2.22.

code is therefore rather slow. For example, we duplicate signals before filtering whereas the filters work in place. Two function have been optimized because we want to compare them against their GPU counterpart: `filter_periodic_horizontal` and `filter_periodic_vertical`.

4.6 Using CUDA

As convolution is a common task, it has already been studied. In particular, NVIDIA provide an example of separable convolution in the CUDA SDK 1.1 `separableConvolution` sample. The sample, unlike our application, performs non-periodic convolution and is a standalone application that creates random data before processing them. We have used parts of this code in our CUDA implementation, and most of the concepts they describe in the documentation [47].

Parts of our program is listed in Subsections B.2 and B.3

Chapter 5

Results

In this chapter, some signals and their process through the different filter banks are shown and discussed in Section 5.1 and 5.2 . Performance analysis is then presented in Section 5.3.

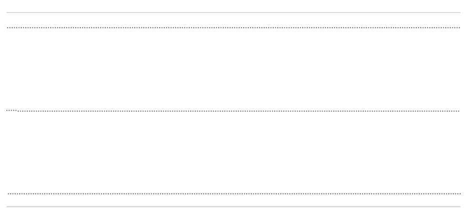
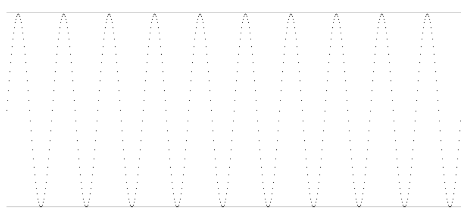
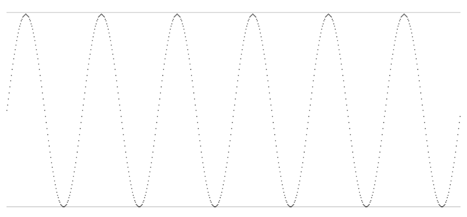
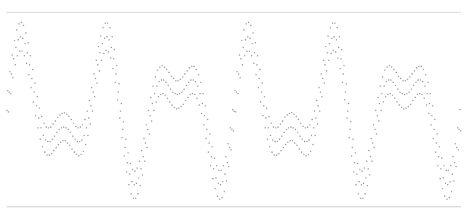
5.1 1D signal analysis

From the *Nyquist-Shannon sampling theorem* [48], the maximum frequency that can be represented in a signal has a period of two pixels. A half-band filter would therefore cut a frequency corresponding to a period of 4 pixels. We have applied the 1D 2-band low-pass high-pass filter banks on signals composed of sinuses, to be able to analyze better the result. The differences around the period of 4 pixels was observed. An example of signals processed by the filter bank, including intermediate ones, are described below. We used the function `fgrim_line_to_2d` of the `GRIM` library to draw these signals.

Table 5.1 presents the input signal E1 processed. The intermediate signals of the 1D non polyphase filter bank are shown in Table 5.2. The name of the signal are the same as in Figure 4.5. First, the input E1 and the output E6 are exactly the same, using either the `fgrim_diff` function or the unix `diff` utility on the BMP files produced. Note that the later method accepts small errors because of the rounding. The filter coefficients have the properties needed to get perfect reconstruction. Second, it is visible from the signals (e.g. E4bx) that band 1 keeps the low frequencies and band 2 the low frequencies. Table 5.3 shows the output signal (E6) of the filter bank when one of the two subbands E3bx is set to zero. The filter bank has nearly separated the high frequency sinus from the two low frequency sinuses.

Using the polyphase filter bank, we also get perfect reconstruction, and the subbands C5b1 and C5b2 are exactly the same than E3b1 and E3b2 respectively.

Table 5.1: 1D Filter Bank Example: Input Signal. Note that s_1 appears like 3 lines, but is actually a sinusoid whose period is so small that only 3 pixels per period is printed.

 s_1: period of 3 pixels	 s_3: period of 60 pixels
 s_5: period of 100 pixels	 E1: $0.5s_1 + 1s_3 + 1.5s_5$

Setting one band to zero produces the same results as in the non polyphase case. From this analysis, we are confident that the 1D filter banks work correctly.

5.2 2D signal analysis

We present the results of the 2-band DFB implemented on an image of size 256^2. The application is called by `./dfb MA256.grim`. The intermediate signals of are shown in Table 5.4. The name of the signal are the same as in Figure 4.5. The CPU and GPU versions of the DFB always give exactly the same results. Therefore, this section applies to both cases.

The filter bank gives perfect reconstruction. Images A1 and A14 are the same, as well as A2 and A13. However, the subbands images do not seem to contain directional information. Table 5.5 (a) and (b) shows the output signal of the filter bank when one of the two subbands A8bx is set to zero. Again, we do not really see directional information. If we apply the same image on the filter bank without doing the frequency shift before and after, we get the two subbands (c) and (d). These two images show high frequencies in (d) and low frequencies in (c) as they

Table 5.2: 1D Filter Bank Example: Intermediate Signals

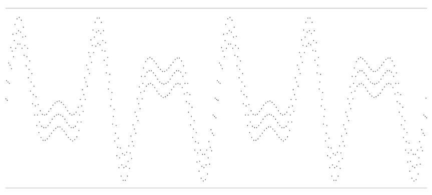
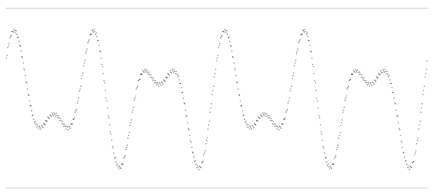
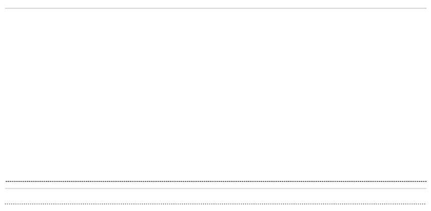
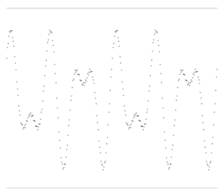
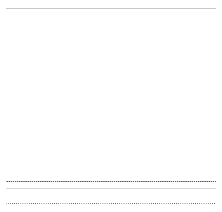
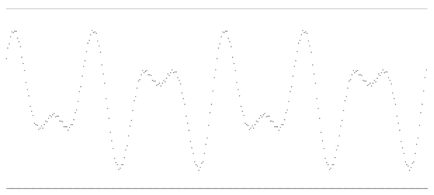
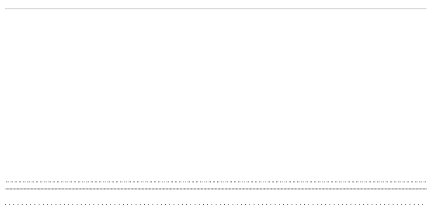
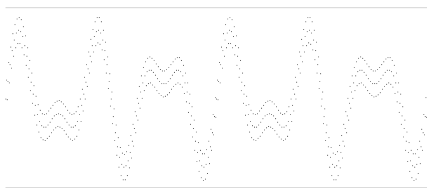
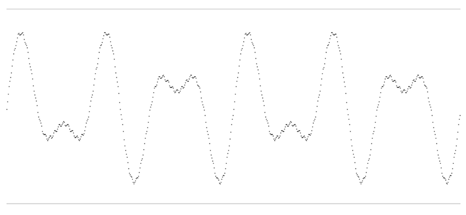
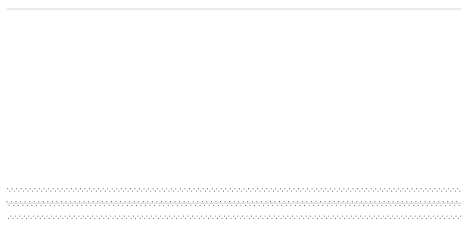
 E1	
 E2b1	 E2b2
 E3b1	 E3b2
 E4b1	 E4b2
 E6	

Table 5.3: 1D Filter Bank Example: Reconstructed Subbands.

	
E6 reconstructed from E3b1 only	E6 reconstructed from E3b2 only

should do. However, we cannot conclude whether they really correspond to the diamond shape filters as they should. If they do, then the problem is probably the frequency shift. This problem does not affect the computation load of the filters, so the performance analysis in next section is valid.

5.3 Performance

5.3.1 Test Machine and Methodology

The machine used for the test is described in Table 5.6. The CPU cores can run at a frequency of 3.20 GHz, but usually run at 2.8 GHz to save power, and switch to a higher frequency when needed. This makes measurements very unstable. We have therefore forced the cores to run at their top speed (3.20 GHz) using the command:

```
cpufreq-selector -g performance      # set core 0 frequency to max
cpufreq-selector -g performance -c 1 # set core 1 frequency to max
```

The code have been compiled with version 4.1.3 of `gcc` and `g++`. Version 4.2 has been tested too but did not give us any better results. The code have been compiled with the flag `-O3` for optimization, `-march=pentium4` for architecture specific optimisation, `-mfpmath=sse` for using SSE2 instruction set. The graphic card used for the computation was also running the X server. Shutting X or having another graphic card for the screen might give better performance.

The graphic card is a GeForce 8800 GTX. Its characteristics are in Table 5.7. The theoretical bandwidth between the CPU and the GPU is 4 GB/s. However, we have measured a bandwidth of only 980MB/s. We were expecting about two times more from the web. All the more, the bandwidth is unstable, and not better

Table 5.4: Bamberger DFB Example: Intermediate Signals

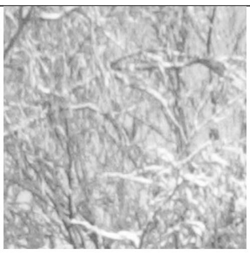
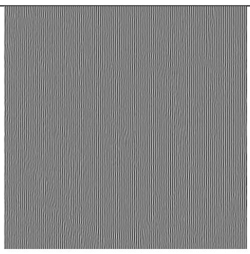
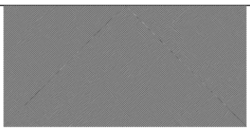
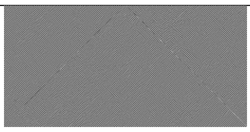
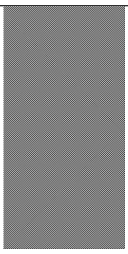
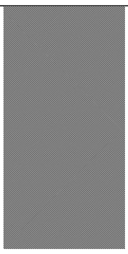
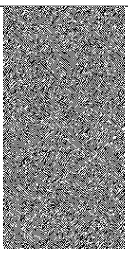
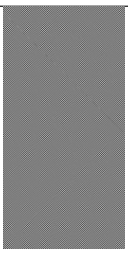
 A1	 A2
 A4b1	 A4b2
A5b1	A5b2
A6b1	A6b2
 A7b1	 A7b2
 A8b1	 A8b2

Table 5.5: Bamberger DFB Example: Reconstructed Subbands. In (c) and (d), the filter bank do not include the frequency shifts before and after. (b) and (c) are smooth.

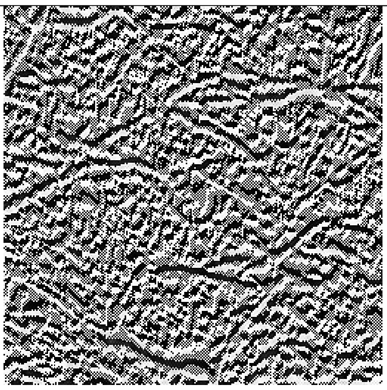
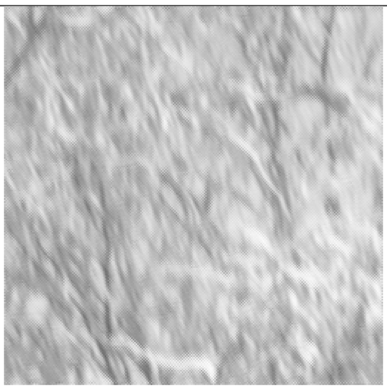
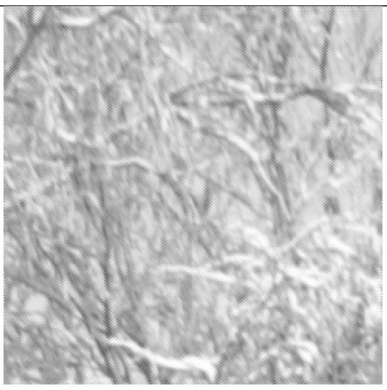
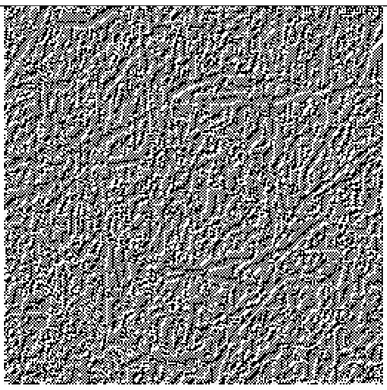
band 1	band 2
 (a)	 (b)
 (c)	 (d)

Table 5.6: Test Platform Description

Operating System	Linux Ubuntu 7.10
Kernel	2.6.22-14 i686
Processors	Dual core Pentium 4, 3.20 GHz
Cache size	2048 KB
RAM	1011 MB
Compiler for C	gcc 4.1.3 prerelease (also tried 4.2)
Compiler for C++	g++ 4.1.3 prerelease (also tried 4.2)
Compiler for CUDA	nvcc, release 1.1 (V0.2.1221)
Graphic card	NVIDIA GeForce 8800 GTX
Graphic card drivers	169.09 (January 2008)
CUDA Toolkit	1.1 (December 2007)

for page-locked memory. The same problems have been reported [49] by several people on the NVIDIA forum. No reasons have been found. It may come from the operating system settings or from the mother board.

The tests have been run 10 times each. The highest and lowest result have been removed to compute the mean. Care have been taken to keep a low standard deviation. The tested application is `dfb`, whose `main` function is in the file `dfb.c`. The macro `noGPU`, `TIMER`, and `noSAVE_IMAGES` have been set in `2d_polyphase_filter.c` to compile for the CPU, and `GPU`, `TIMER`, and `noSAVE_IMAGES` to compile for the GPU. The filter size used by the convolution is 8. That is, h_0 and h_1 are 16 numbers long, but we convolve with the polyphase filters p_0 and p_1, which are two time smaller.

As can be seen from the code in Figure 5.1, only the horizontal and vertical filters have been timed. Indeed, the rest is only implemented on the CPU.

For timing, we have used the timer `clock()` from `time.h` and `gettimeofday()` from `sys/time.h`. The former appeared to have a resolution of 10 ms, which was too low for our comparisons. The later gives result with a detail of 0.001 ms. We have also used the x86 assembly instruction `RDTSC` which returns the number of ticks of the processor. For the results below, the timer `gettimeofday()` have been used.

5.3.2 CPU Version

Table 5.8 present the result of the tests for convolution on CPU, and Figures 5.2 (a) and (b) shows the corresponding curves. The horizontal axis of the graphics is the *side* of the images used for the tests. It is the square root of the number

Table 5.7: Graphic Card Description: NVIDIA GeForce 8800 GTX

Global memory	768 MB (GDDR3)
Multiprocessors	16
Processors	128 = 16×8
Bandwidth CPU-GPU	4 GB/s in theory
Memory bandwidth	86.4 GB/s in theory
Clock	575 MHz
CUDA compute capability	1.0
Shared memory	16 KB per multiprocessor
Transistors	681 millions (480 mm^2 die surface, 90 nm)
Power	185 Watts
Bus	Two connectors PCI Express (x16)

```
TIMER_BEGIN;
  MACRO_filter_periodic_horizontal(&sA4a5b1, &p0_coef, offset1);
  MACRO_filter_periodic_horizontal(&sA4a5b2, &p1_coef, offset1);
TIMER_END("horizontal")
SAVE(&sA4a5b1, "sA05b1.bmp"); SAVE(&sA4a5b2, "sA05b2.bmp");
  reorganize_h2v(&sA4a5b1, &sA6a7b1);
  reorganize_h2v(&sA4a5b2, &sA6a7b2);
SAVE(&sA6a7b1, "sA06b1.bmp"); SAVE(&sA6a7b2, "sA06b2.bmp");
TIMER_BEGIN;
  MACRO_filter_periodic_vertical(&sA6a7b1, &p0_coef, offset1);
  MACRO_filter_periodic_vertical(&sA6a7b2, &p1_coef, offset1);
TIMER_END("vertical␣␣")
```

Figure 5.1: Extract of the Analysis Part of the 2D Polyphase Filter, in `2d_polyphase_filter.c`. The functions within the timer run on the CPU or GPU depending whether the `GPU` macro is defined or not.

Table 5.8: CPU Results: Wall-clock Time in ms for Horizontal and Vertical Filters on Different Image Size.

Horizontal convolution						
Image side (pixel)	256	512	1024	1920	2048	7680$^{(*)}$
Maximum time	2.35	9.08	25.21	78.16	84.33	4542.89
Minimum time	1.35	5.09	20.57	72.14	83.45	4542.89
Average time	1.99	6.14	21.93	73.8	83.81	4542.89

Vertical convolution						
Image side (pixel)	256	512	1024	1920	2048	7680$^{(*)}$
Maximum time	2.78	12.13	145.88	133.26	550.77	8536.42
Minimum time	1.55	7.4	133.28	123.27	531.63	8536.42
Average time	2.16	9.69	137.01	127.14	544.72	8536.42

$^{(*)}$ The CPU tests have been run only once on the 7680×7680 image, because the whole program (though not the timed functions) is too slow for that size.

of pixels of the image. The first remark is that the results are consistent in that the different measurements of the same test are very similar. We will therefore consider only the average time.

Let us first consider the horizontal convolution. Convolving an image has a complexity of N^2 (with a fixed size filter), N being the side of the image. Without being evident, at least the curve (a) do not mismatch the complexity, especially. For an image of 1920^2, a naive algorithm of three for loops takes 440ms. Our optimized version now takes 75ms. A similar speedup has been observed for other matrix sizes.

The results of vertical convolution are more surprising. If we ignore the 1920^2 image, the curve has the same shape than for horizontal convolution but about 5 times slower. It is not surprising that vertical convolution is slower because it has a bad memory access pattern. Indeed, it reads the values by column, so it gets a cache miss at least for every new value read, whereas the horizontal convolution access memory sequentially and nearly always hit the cache. Recall that convolution is $y[n] = \sum_k x[n+k]h[k]$, with h the filter (of size 8 in the tests). The memory address read increased by 1 height times and decrease by 7 one time, and increase again. Let us now consider the value for 1920. It is definitely not an accident because it has been reproduced many times. Here also, the explanation could be the cache. We do not know the number of lines of the cache, but we suppose that the other images used have a width which is a multiple of the number of cache lines. Then, every value of a column would want to be in the same line

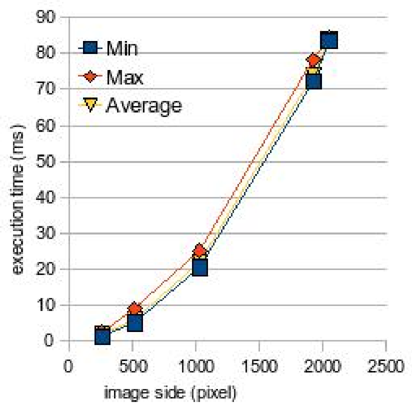
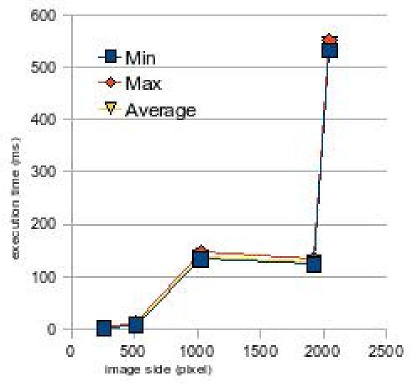
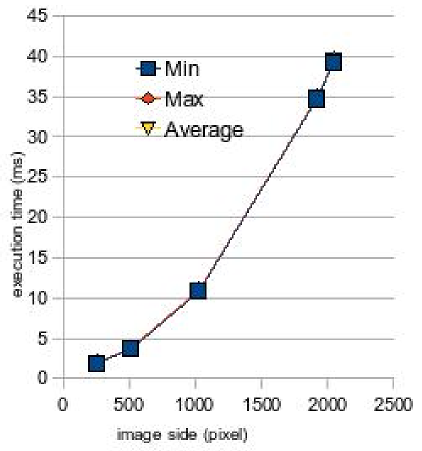
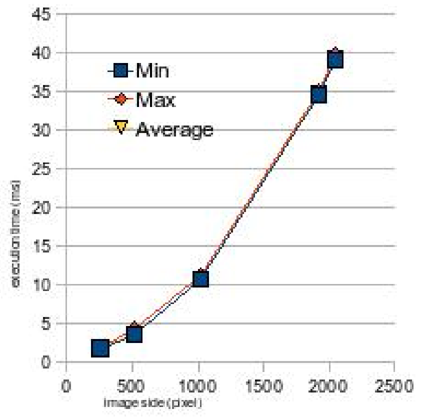
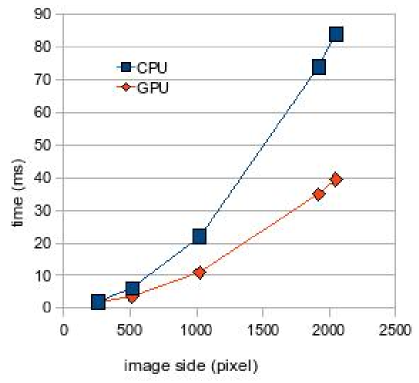
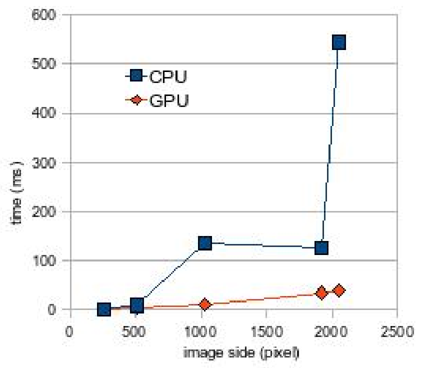
Horizontal	Vertical
CPU	
 (a)	 (b)
GPU	
 (c)	 (d)
Both CPU and GPU	
 (e)	 (f)

Figure 5.2: Performance Comparison

Table 5.9: GPU Results: Wall-clock Time in ms for Horizontal and Vertical Filters on Different Image Size.

Horizontal convolution						
Image side (pixel)	256	512	1024	1920	2048	7680
Maximum time	2.04	3.81	11.05	35.04	39.52	1066.2
Minimum time	1.91	3.74	10.87	34.76	39.29	601.01
Average time	1.93	3.77	10.93	34.88	39.38	874.76

Vertical convolution						
Image side (pixel)	256	512	1024	1920	2048	7680
Maximum time	1.9	4.2	11.25	34.99	39.72	537.3
Minimum time	1.77	3.59	10.73	34.57	39.13	534.06
Average time	1.8	3.63	10.8	34.77	39.23	534.86

of the cache, and we would always miss the cache. For the 1920^2 matrix however, the distance in memory between two values of a column is not a multiple of the line number, so the values are shifted in the cache and arrive on different lines. Therefore, the 7 previous values are still there (on other lines) and we have only one miss per pixel. We did not have time to investigate further cache issues, nor have we used a cache profiler such as `Cachegrind` in this thesis. Some tricks could be used to optimize the code further. A first idea is to allocate a new place in memory of the size of the image and copy the input image while transposing it. Then the reads would virtually always hit the cache.

5.3.3 GPU Version

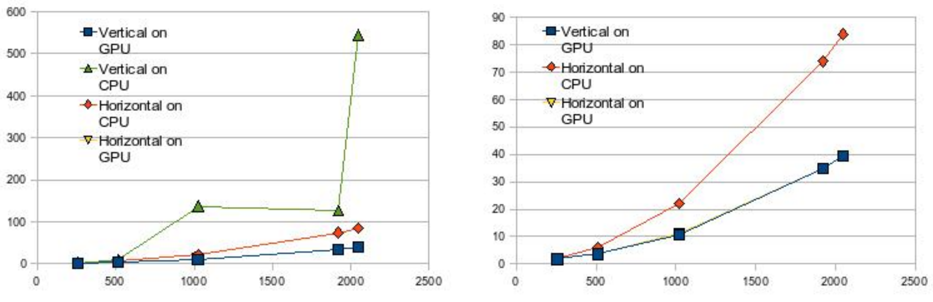
Table 5.9 present the result of the tests for convolution on GPU, and Figures 5.2 (c) and (d) shows the corresponding curves. Same as before, the different measurements of the same test are very similar so we consider only the average times. The curves are consistent as well with the N^2 complexity.

There is no specific cache issue here because the image is first accessed sequentially to send it to the device, then in parallel by block of width 16 from the global memory as described in section 4.6.

It is worth noticing that the horizontal and vertical convolution takes the same time. On the graphics of Figure 5.3, it is not possible to distinguish these two curves.

We have run several tests on the 1920^2 image commenting different parts of the code to find the bottlenecks. We have found one: the bandwidth. Of 35 ms,

Figure 5.3: Performance Comparison: CPU vs. GPU. In (b), the CPU vertical convolution curve is not shown. In (a) and (b), the two GPU curves for horizontal and vertical convolution overlap.



(a) (b)

about 5 ms are used to allocate device memory, 15 ms to transfer the input image, and 15 ms to load the result back. The computation itself takes less than 1 ms (measured with the CUDA toolkit timer). 1920^2 floats represents 14 MiB, which would mean that the bandwidth is about $\frac{1920^2*4}{15*10^{-3}}/10^6 = 983MB/s$. The theoretical bandwidth is 4 GB/s. Running the bandwidth test of the CUDA SDK gives similar results, both for paged-blocked and non paged blocked host memory, as discussed previously. We were surprised that the kernel needs less than 1 ms to run, even on quite large images. It result from this result that emphasis should be put more on the code surrounding the kernel. We had not notice it early, as we have developed the application mainly using a simulator on a machine without CUDA graphic card. To summarize this paragraph, the bottleneck is the bandwidth.

5.3.4 GPU-CPU Comparisons

Figure 5.3 (a) shows the curves for CPU and GPU, horizontal and vertical convolution. On (b), the vertical curve has been removed, which permits a better comparison between CPU and GPU. The speedup observed for the GPU is about 2. This is rather low and should be qualified. This result represent a worse case because (a) the C implementation against which it is compared is rather fast (see the description in Section 4.5 and the results in 5.3.3), (b) the bandwidth of the test platform is abnormally slow, and (c) only a small part of the algorithm has been implemented on the GPU so the arithmetic intensity is not very high.

(a) The speedup for the vertical convolution is much higher than for horizontal, as can be seen in Table 5.10.

(c) The key to get a good overall speedup for the application is to keep the

Table 5.10: Speedup of the Filter Steps for GPU vs. CPU for Different Image Size

Image side (pixel)	256	512	1024	1920	2048	7680
Horizontal speedup	1.03	1.63	2.01	2.12	2.13	5.19
Vertical speedup	1.2	2.67	12.69	3.66	13.89	15.96

images on the GPU between computation steps. The other steps than filtering (downsampling, period reorganization, combine, upsampling) should also be programmed on the GPU, not only to increase their execution time, but also to avoid transferring data between the CPU and the GPU. Allocating two times the size of the image should be enough, and the buffers can be used alternately as input or output of each steps. The most important step to program on the GPU is the reorganization. The reason is that it is between the two steps that benefit the most of the GPU: filters. It could be possible to avoid the reorganisation steps by changing the reading part of the vertical analysis filters and the writing part of the vertical synthesis filters. In most applications, only the analysis or the synthesis part is considered and the subbands need to be transferred. However, for a simple known operation on one of the subbands, it could be done on the GPU too. If this operation is set one of the subbands to zero, then half of the first combine steps can be skipped as well as half or the whole of the second combine step (which would add or subtract zero), depending on the band set to zero.

Chapter 6

Conclusions and Future Work

In this chapter, we summarize our research findings and look at potential future work.

6.1 Conclusion

In this thesis, we looked at the Bamberger directional filter bank (DFB) algorithm and how to implement it efficiently on CPU and GPU. We have translated the algorithm from a theoretical signal processing description to a practical computer scientist language, including a readable `C` implementation. Tools have been developed to ease DFB investigation, including an independent library to manipulate images in different formats and to generate test images with suitable properties, as well as different implementations of 1D filter banks. The most compute intensive steps of the Bamberger DFB were implemented efficiently with CUDA to run on GPU. It was shown that the bandwidth is the bottleneck. We observed for different size of images a speedup of 2 *in the worse case*, that is comparing horizontal convolution with an optimized CPU version. The speedup for vertical convolution was much better and would still be better with a more optimized CPU version. Our platform had a bandwidth about two times slower than bandwidths reported on the Internet. It is likely that running the program on a platform without this problem gives a much better speedup, since for medium size images, the transfer represents more than 80% of the time spent by the functions tested. Our results indicate that the overall performance of the DFB could significantly benefit from being implemented on the GPU. Given that the stages of the DFB may all benefit from the GPU, we predict an overall performance of 5-10, depending on the system bandwidth, for a full DFB implementation. Directional filter banks can efficiently been executed on GPUs.

6.2 Future Work

As a computer science student with limited background in signal processing, the journey of developing these implementations from scratch was a lot of work, but a very educational one. Hopefully, future students and others interested in optimization of filter bank implementations will benefit from this journey.

Unfortunately, the time allocated for this thesis was limited, and the lack of an initial implementation of the Bamberger DFB moved back the starting point of the thesis. In this section, we provide some pointers to ideas that could be explored further.

- On a digital processing side, a deeper analyze of intermediate results should be done to determine if they are what is expected or not, and why. Looking at the pre-processing and post-processing frequency shifts might be an issue.

- Again on a digital processing side, some work need to be done to see how to use this filter bank in a real-world application. There is probably a trade-off between desirable visual results and computation time. The choice of the Bamberger filter-bank was made to satisfy the need of efficiency. This choice could be reconsidered or some improvements to the algorithm as described in this thesis could be made. It has for example been suggested[1, 2] to apply a low pass filter on the image first to get rid of the border aliasing near the center of the frequency plan.

- Only chosen steps of the DFB have been implemented on the GPU, and we predict that the overall program would benefit from GPU implementation. Such a task would need to be made if this project was to be continued. Some other GPU technologies could be investigated, for example to be vendor independent, so that the user is not tied use a graphic card from NVIDIA.

- Many personal computers do not have an advanced graphic card, and doing a performant CPU implementation of the DFB would also make sense for these cases. Using SSE instructions could be a starting point for optimization, but the key bottleneck to analyze is undoubtedly the use of the cache. Because multi-core CPUs are now common, investigating a multi-threaded version of the algorithm would also be worthwhile, for example using `pthread`.

Bibliography

[1] R. H. Bamberger and M. J. T. Smith, “A filter bank for the directional decomposition of images: theory and design,” *Signal Processing, IEEE Transactions on*, 1992.

[2] T. T. Nguyen, *The multiresolution directional filter banks*. PhD thesis, University of Texas at Arlington, August 2006.

[3] Chul-Hyun Park, Joon-Jae Lee, M. J. T. Smith, Sang-il Park, and Kil-Houm Park, “Directional filter bank-based fingerprint feature extraction and matching,” *Circuits and Systems for Video Technology, IEEE Transactions on*, 2004.

[4] Tore Fevang, engineer at Schlumberger Trondheim. Personal conversation, 2007-2008.

[5] J. Deller, Jr., “Tom, Dick, and Mary discover the DFT,” *Signal Processing Magazine, IEEE*, 1994.

[6] Unknown, “Introducing the z-transform.” typeset with LaTeX by Stuart Longland, available at Available on: `http://dev.gentoo.org/~redhatter/misc/z-transform.pdf`, accessed on 15 February 2008.

[7] M. D.Lutovac, D. V. Tosic, and B. L. Evans, *Filter design for signal processing, using `MATLAB`® and `Mathematica`®* . Prentice-Hall, 2001.

[8] Wikipedia. Available on: `http://en.wikipedia.org/wiki/Discrete_Fourier_transform#Spectral_analysis`, accessed on 2 February 2008.

[9] Wikipedia. Available on: `http://en.wikipedia.org/wiki/Fast_Fourier_transform`, accessed on 15 February 2008.

[10] M. Frigo and S. G. Johnson, “FFTW: Fastest fourier transform in the west.” Available on: `http://www.fftw.org/`, accessed on 15 February 2008.

[11] D. J. Bernstein, "djbfft." Available on: `http://cr.yp.to/djbfft.html`, accessed on 15 February 2008.

[12] Wikipedia. Available on: `http://en.wikipedia.org/wiki/Z-transform`, accessed on 2 February 2008.

[13] D. E. Dudgeon and R. M. Mersereau, *Multidimensional Digital Signal Processing*. Prentice-Hall, 1984.

[14] Wikipedia. Available on: `http://en.wikipedia.org/wiki/LTI_system_theory`, accessed on 6 February 2008.

[15] NVIDIA, *FFT-based 2D convolution*, 1.1 ed., June 2007.

[16] Wikipedia. Available on: `http://en.wikipedia.org/w/index.php?title=Convolution&oldid=187459852#Fast_convolution`, accessed on 15 February 2008. This page is the version of the 28 January 2008 of the page `http://en.wikipedia.org/wiki/Convolution#Fast_convolution_algorithms`, also accessed on 15 February 2008.

[17] Wikipedia. Two Wikipedia pages that gives hints on using FFT for computing convolution: `http://en.wikipedia.org/wiki/Circular_convolution` and `http://en.wikipedia.org/wiki/A_derivation_of_the_discrete_Fourier_transform`, both accessed on 15 February 2008.

[18] I. Daubechies, "Ten lectures on wavelets," *Society for Industrial and Applied Mathematics*, 1992.

[19] J. R. Deller, J. G. Proakis, and J. H. L. Hansen, *Discrete-time processing of speech signals*. Prentice-Hall, 1993.

[20] P. Vaidyanathan, "Multirate digital filters, filter banks, polyphase networks, and applications: a tutorial," *Proceedings of the IEEE*, Jan 1990.

[21] P. Vaidyanathan, "Quadrature mirror filter banks, m-band extensions and perfect-reconstruction techniques," *ASSP Magazine, IEEE [see also IEEE Signal Processing Magazine]*, Jul 1987.

[22] Sang-Il Park, M. J. T. Smith, and R. M. Mersereau, "Improved structures of maximally decimated directional filter Banks for spatial image analysis," *Image Processing, IEEE Transactions on*, 2004.

[23] R. Mersereau and T. Speake, “The processing of periodically sampled multidimensional signals,” *Acoustics, Speech, and Signal Processing [see also IEEE Transactions on Signal Processing], IEEE Transactions on*, 1983.

[24] Sang-Il Park, M. J. T. Smith, and R. M. Mersereau, “A new directional filter bank for image analysis and classification,” *Acoustics, Speech, and Signal Processing, 1999. ICASSP ’99. Proceedings., 1999 IEEE International Conference on*, 1999.

[25] L. C. Larsen, “Utilizing gpus on cluster computers,” project report, Norwegian University of Science and Technology, 2006.

[26] M. Harris, “Mapping computational concepts to gpus,” in *GPU Gems 2: Programming Techniques for High-performance Graphics and General-purpose Computation* (Addition-Wesley, ed.), pp. 493–508, 2005.

[27] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, “A survey of general-purpose computation on graphics hardware,” in *Eurographics 2005, State of the Art Reports*, pp. 21–51, Aug. 2005.

[28] Microsoft, “Microsoft Research Accelerator Project.” Available on: `http://research.microsoft.com/research/downloads/Details/25e1bea3-142e-4694-bde5-f0d44f9d8709/Details.aspx?CategoryID`, accessed on 15 February 2008.

[29] “Brook GPU home page.” Available on: `http://graphics.stanford.edu/projects/brookgpu/`, accessed on 15 February 2008.

[30] “Sh library.” Available on: `http://libsh.org/`, accessed on 15 February 2008.

[31] RapidMind, “RapidMind.” Available on: `http://www.rapidmind.net/`, accessed on 15 February 2008.

[32] Wikipedia. Available on: `http://en.wikipedia.org/wiki/Larrabee_(GPU)`, accessed on 15 February 2008.

[33] HPC Wire, “GPU Computing Gets Ready for Act II.” Available on: `http://www.hpcwire.com/hpc/1856011.html`, accessed on 15 February 2008.

[34] ATI and AMD, “Close To Metal guide.” Available on: `http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf`, accessed on 15 February 2008.

[35] IBM, "Cell Broadband Engine." Available on: http://www-128.ibm.com/developerworks/power/cell/, accessed on 15 February 2008.

[36] Sam Williams, "The Potential of the Cell Processor for Scientific Computing." Available on: http://www.cs.berkeley.edu/~samw/research/talks/lbl06.pdf, accessed on 15 February 2008.

[37] NVIDIA, "CUDA home page." Available on: http://developer.nvidia.com/object/cuda.html, accessed on 15 February 2008.

[38] David Kirk and Wen-mei W. Hwu, "Programming Massively Parallel Processors." Available on: http://courses.ece.uiuc.edu/ece498/al1/lectures/lecture1%20intro%20fall%202007.ppt, accessed on 15 February 2008.

[39] NVIDIA, *NVIDIA CUDA Programming Guide*, 1.0 ed., June 2007.

[40] "Description of the BMP file format." http://www.martinreddy.net/gfx/2d/BMP.txt17 February 2008.

[41] "GTK+." Available on: http://www.gtk.org/, accessed on 15 February 2008.

[42] "ImageMagick." Available on: http://www.imagemagick.org, accessed on 15 February 2008.

[43] "GraphicsMagick." Available on: http://www.graphicsmagick.org/, accessed on 15 February 2008.

[44] "imlib." Available on: http://freshmeat.net/projects/imlib/, accessed on 15 February 2008.

[45] DevIL. Available on: http://openil.sourceforge.net/, accessed on 15 February 2008.

[46] Free Software Fundation, "GNU Lesser General Public License." Available on: http://www.fsf.org/licensing/licenses/lgpl.html, accessed on 15 February 2008.

[47] NVIDIA, *Image Convolution with CUDA*, 1.1 ed., June 2007.

[48] A. V. Oppenheim, A. S. Willsky, and S. H. Nawab, *Signals & Systems*. Prentice Hall, 2nd ed., 1997.

[49] NVIDIA Forums, "Low CPU-GPU bandwidth on Linux." Available on: http://forums.nvidia.com/index.php?showtopic=53894, accessed on 14 February 2008.

[50] M. J. T. Smith and A. Docef, *A Study Guide for Digital Image Processing*. Scientific Plublishers, early realease ed., 1997.

[51] J. G. Rosiles and M. J. T. Smith, "A low complexity overcomplete directional image pyramid," *ICIP, IEEE*, 2003.

[52] P. S. Hong, L. M. Kaplan, and M. J. T. Smith, "Hyperspectral image segmentation using filter banks for texture augmentation," *IEEE*, 2004.

[53] "gpgpu.org." Available on: `http://www.gpgpu.org`, accessed on 15 February 2008.

Appendix A

Annotated Bibliography

This appendix presents the most important papers and book of the bibliography, including those which might be important for a deaper study.

Signal Processing and Digital Image Filters

An excellent presentation of the discrete Fourier transform (DFT) and its relationship with the continuous (FT) and continuous-time (CTFT) Fourier transforms is made by Deller in its paper *Tom, Dick, and Mary discover the DFT* [5, 14 pages].

A crach introduction from 1983 of periodic image processing is given by Mersereau and Speak in *The processing of periodically sampled multidimensional signals* [23, 4 pages].

Reading Chapters 1, 2, 3 and 8 of *Filter design for signal processing* [7] is probably enough to give a deep view of digital filter design.

However, we are more interrested in filter implementation than design, and interest ourselve to 2D signals. *Multidimensional Digital Signal Processing* [13] is a very good book from Dudgeon and Mersereau. It begins with a presentation of multidimensional signals and systems and discrete Fourier analysis (including many transforms), and then present in detail the design and (serial) implementation of 2D FIR and IIR filters.

In order to understand the concepts of quadrature mirror filters (QMF) and polyphase filters, *Discrete-time processing of speech signals* [19] from Deller and al. have been used, in particular the Appendix 7.A.

Finally, *A Study Guide for Digital Image Processing* [50] from Smith and Docef looks a promising book, though only the QMF filter bank coefficients of Table 5.5 have been used in this thesis.

Directional Filter Bank

In 1992, Bamberger and Smith published the article *A filter bank for the directional decomposition of images: theory and design* [1] presenting a novel algorithm for directional decomposition of images. This algorithm has many properties that made it known: efficient computation, maximally decimation, perfect reconstruction, separable filtering, tree structure.

In 1999, Sang-il Park, Smith and Mersereau improved this algorithm in *A new directional filter bank for image analysis and classification* [24]. They add resampling matrices to get visualizable directional subbands. The additional cost of the resampling matrices can be reduced by combining all resampling matrices together at the end of each bank-bands. They also gives many hints and design parameters to implement such a filter bank, provided the simple 2-band diamond filter bank.

The Bamberger filter-bank has been appreciated by the scientific community, as acknowledged by a paper from Rosiles and Smith in 2003: *A low complexity overcomplete directional image pyramid*[51]. They also present in this paper an undecimated DFB based on the Bamberger maximally decimated DFB. Using undecimated subbands permit them to design a filter with additional properties, such as shift invariance and an easy structure.

Hyperspectral image segmentation using filter banks for texture augmentation [52, 2004] uses an octave-band DFB for appending texture information to hyperspectral images in order to ease good classification.

Directional filter bank-based fingerprint feature extraction and matching [3, early 2004] uses the Bamberger DFB to extract feature vector from fingerprint images and compare them against a set of other fingerprint feature vector to find a possible match. This article presents in Figure 4 the complete path from the root to a leaf of the decomposition tree for an eight-band decomposition. We have extended this tree in Figure 2.20 page 27 of this report.

One of the latest publications on the subject is the PHD thesis of Truong Nguyen in 2006: [2] *The multiresolution directional filter banks*. In this document, Nguyen gives a global view of the DFB field. He presents the properties a DFB can have and discuss them, and present a novel algorithm and new results. Chapter 2 is a review of the fields. In particular, it presents the Bamberger DFB as the father of most others, and summarize the three critics that can be made to it.

GPU, GPGPU, CUDA

Many presentation slides can be found on the Internet that introduce with GPUs and the concept of GPGPU. However, the website *gpgpu.org* [53], and in particular the Tutorial section, is probably one of the best ressources to start learning practical GPGPU programming technologies. In *A Survey of General-Purpose Computation on Graphics Hardware* [27], Owens et al. give an excellent overview of GPGPU issues as of 2005. It presents the hardware mechanisms, programming concepts and current technologies, and applications. Each language and technology (OpenGL, Cg...) has its own acknowledged didactic programming book of choice. For CUDA, it is currently the programming guide [39]. The SDK samples are worth looking at, and some of them come along with a description guide in the `doc/` directory.

Appendix B

Description of the dfb program

The code of the project is organized into several directories. The GRIM library and grim2x application are in grim/, the octave code is in octave/, and the 1D and 2D filters are in dfb/. The 1D non-polyphase filter bank is in 1d_filter.c and the polyphase version in 1d_polyphase_filter.c. The Bamberger DFB is implemented in the function polyphase_filter_2d defined in 2d_polyphase_filter.c and in Section B.1 and the main program is in dfb.c. The command line to process the DFB on an image is dfb [-h] [-o <output_image>] <input_image>. In 2d_polyphase_filter.c, the macros GPU, TIMER and SAVE_IMAGES control at compilation time wether to compile the CPU or GPU version, wether to time the executions of the filters, and wether to input intermediate signals as BMP images. All intermediate images are generated in the directory images/. The length of the QMF half-band low pass filter h_0 to use is defined in filter.h. Remind that the length of the polyphase filters p_0 and p_1 are two times smaller. The filter banks use the following functions defined in filter.c:

```
void init_coef(fgrim *filter_coef);
void free_coef(fgrim *filter_coef);
int pair_filter_coef(fgrim *lpf, fgrim *hpf);

int frequency_shift_PI(fgrim *fimg);
int downsample_quincunx_q2(fgrim *in,fgrim *outb1,fgrim *outb2);
int upsample_quincunx_q2(fgrim *out, fgrim *inb1, fgrim *inb2);
int filter_periodic_horizontal(fgrim *fg, fgrim *coef, int
    offset);
int filter_periodic_vertical(fgrim *fg, fgrim *coef,int offset);
int reorganize_h2v(fgrim *fin, fgrim *fout);
int reorganize_v2h(fgrim *fin, fgrim *fout);
int polyphase_combine(const fgrim *ib1, const fgrim *ib2, fgrim
    *ob1, fgrim *ob2);
int amplify(fgrim *signal, float gain);
```

The host parts of the CUDA filters are in dfb_cuda.cu. The CUDA code to run on the device are in dfb_kernel.cu. In dfb_cuda_gold.cpp, it is still another version of a C implementation of the DFB.

The host and GPU code of the GPU version of the horizontal periodic filter are listed in Sections B.2 and B.3 respectively. We have used some code from the NVIDIA CUDA SDK 1.1 sample convolutionSeparable. The license authorizes us to use it and deny all liability. The license also states that we must include it in the code and the documentation of our program. Since this appendix is the documentation, we include NVIDIA disclaimer in Figure B.1.

B.1 DFB code

Below is the code in 2d_polyphase_filter.c, whose function polyphase_filter_2d is the core of the DFB.

```
/*****************************************************************************
 * Author: Jerome (J'er^ome) Dubois
 * Email : Jerome.Dubois@ensimag.imag.fr, jerome@stud.ntnu.no
 * Created: 2008-01-15
 * File  : 2d_polyphase_filter.c
 * Copyright 2008 Jerome Dubois
 *
 * Goal: Compute a 2D polyphase filter. Signals are 2D signals.
 ****************************************************************************/

#include <stdlib.h>  // exit, EXIT_SUCCESS
#include <stdio.h>   // printf, FILE. Already included in grim.h
 #include <assert.h> // assert
#include <time.h>    // clock()
#include "filters.h"
#include "stopwatch.h" // Timer stuff: code around gettimeofday() of sys/time.h

// Function declaration
extern int polyphase_filter_2d(fgrim *in, fgrim *out, fgrim *h0_coef); // here
// In a_my_convolutionSeparable.cu:
extern int filter_periodic_horizontalGPU(fgrim *fg, fgrim *coef, int offset);
extern int filter_periodic_verticalGPU(  fgrim *fg, fgrim *coef, int offset);

#define IMAGES "../images/"   // path to the images (to read and create)
#define noGPU
#define TIMER
#define SAVE_IMAGES

#define DEFINED
#undef  UNDEFINED

#ifdef GPU
#include <cuda_runtime.h> // for cudaMallocHost
#endif

/*#########################################################################*/
/*#                        various MACROS                                 #*/
/*#########################################################################*/

```

Figure B.1: We have used some code from the NVIDIA CUDA SDK 1.1 sample `convolutionSeparable`. The license authorizes us to use it and deny all liability. The license also states that we must include it in the code and the documentation of our program. Since this appendix is the documentation, we include here NVIDIA disclaimer.

```
#ifdef SAVE_IMAGES
#define SAVE(sig,file) fgrim_write_bmp((sig),(IMAGES file))
#else
#define SAVE(sig,file)
#endif

// Timer 1: clock() of time.h
#define TIMER_1_DECLARE clock_t start, stop; float time;
#define TIMER_1_BEGIN   start = clock();
#define TIMER_1_END(direction)                                  \
   stop = clock();                                              \
   time = ((float)stop - start) / CLOCKS_PER_SEC * 1000;        \
   printf( direction "␣filtering␣of␣both␣bands:␣Timer␣1:␣%f␣ms\n", time);
#define TIMER_1_DELETE

// Timer 2: gettimeofday() of sys/time.h through stopwatch.h
#define TIMER_2_DECLARE float stopwatch_time;
#define TIMER_2_BEGIN   stopwatch_reset(); stopwatch_start();
#define TIMER_2_END(direction)                                  \
    stopwatch_stop();                                           \
    stopwatch_time = stopwatch_getTime();                       \
    printf( direction "␣filtering␣of␣both␣bands:␣Timer␣2:␣%6.4f␣ms\n", \
            stopwatch_time );
#define TIMER_2_DELETE

#ifdef TIMER
#define TIMER_DECLARE        TIMER_1_DECLARE        TIMER_2_DECLARE
#define TIMER_BEGIN          TIMER_1_BEGIN          TIMER_2_BEGIN  // 2 has prio
#define TIMER_END(direction) TIMER_2_END(direction) TIMER_1_END(direction) // id
#define TIMER_DELETE         TIMER_1_DELETE         TIMER_2_DELETE
#else
#define TIMER_DECLARE
#define TIMER_BEGIN
#define TIMER_END(direction)
#define TIMER_DELETE
#endif

#ifdef GPU
#define MACRO_filter_periodic_horizontal(signal,coef,offset) \
              filter_periodic_horizontalGPU((signal), (coef), (offset));
#define MACRO_filter_periodic_vertical(  signal,coef,offset) \
              filter_periodic_verticalGPU(  (signal), (coef), (offset));
#else
#define MACRO_filter_periodic_horizontal(signal,coef,offset) \
              filter_periodic_horizontal((signal), (coef), (offset));
#define MACRO_filter_periodic_vertical(  signal,coef,offset) \
              filter_periodic_vertical(  (signal), (coef), (offset));
#endif

/*##########################################################################*/
/*#                        various functions                               #*/
/*##########################################################################*/
#ifdef GPU
int cudaMalloc_fgrim_create(fgrim *fimg, int width, int height);
int cudaMalloc_fgrim_free(fgrim *fimg);

int cudaMalloc_fgrim_create(fgrim *fimg, int width, int height){
  fimg->w = width;
  fimg->h = height;
  cudaMallocHost( (void**) &fimg->pix, width * height * sizeof(float) );
  return EXIT_SUCCESS;
}
```

```
int cudaMalloc_fgrim_free(fgrim *fimg){
  fimg->w = 0;
  fimg->h = 0;
  cudaFreeHost(fimg->pix);
  return EXIT_SUCCESS;
}
#endif // #ifdef GPU

/*###########################################################################*/
/*#                       2d_polyphase_filter                               #*/
/*###########################################################################*/
int polyphase_filter_2d(fgrim *in, fgrim *out, fgrim *h0_coef){
  int i, offset1, offset2;
  int w, w2; // width of the signals
  int fs, fs2; // size of the filters
  fgrim p0_coef, p1_coef;   // polyphase filter coefficients
  TIMER_DECLARE;

    /* preparation */
    w  = in->w;
    fs = h0_coef->w;
    assert(0 ==  w%2);
    assert(0 == fs%2);
    assert(w == in->h);
    w2  = w /2;
    fs2 = fs/2;

    // coefficient preparation
    fgrim_create(&p0_coef, fs2, 1);
    fgrim_create(&p1_coef, fs2, 1);
    for (i=0; i<fs2; i++){
      p0_coef.pix[i] = h0_coef->pix[2*i  ];
      p1_coef.pix[i] = h0_coef->pix[2*i+1];
    }
    offset1 = 0; // assert( abs( offset1)<w ) ?
    offset2 = fs2-offset1-1; // assert( abs( offset1)<w ) ?

    // bank preparation
    fgrim *sA1, sA2, *sA13a14; // legend: sA13a14 = signal A13 and A14
    fgrim sA4a5b1, sA6a7b1, sA8b1, sA9a10b1, sA11a12b1; // band 1
    fgrim sA4a5b2, sA6a7b2, sA8b2, sA9a10b2, sA11a12b2; // band 2

    sA1 = in;  // pointer copy
    fgrim_create(&sA2, w, w);
    //fgrim_create(&sA4a5b1  , w , w2); fgrim_create(&sA4a5b2  , w , w2);
    fgrim_create(&sA8b1    , w2, w ); fgrim_create(&sA8b2    , w2, w );
#ifdef GPU // We allocate page-locked memory to improve the bandwidth
    //cudaError_t err; err = cudaMalloc...
    //printf( cudaGetErrorString( err ) ); printf("\n");
    cudaMalloc_fgrim_create(&sA4a5b1  , w , w2);
    cudaMalloc_fgrim_create(&sA4a5b2  , w , w2);
    cudaMalloc_fgrim_create(&sA6a7b1  , w2, w );
    cudaMalloc_fgrim_create(&sA6a7b2  , w2, w );
    //cudaMalloc_fgrim_create(&sA9a10b1 , w2, w );
    //cudaMalloc_fgrim_create(&sA9a10b2 , w2, w );
    //cudaMalloc_fgrim_create(&sA11a12b1, w , w2);
    //cudaMalloc_fgrim_create(&sA11a12b2, w , w2);
    fgrim_create(&sA9a10b1 , w2, w ); fgrim_create(&sA9a10b2 , w2, w );
    fgrim_create(&sA11a12b1, w , w2); fgrim_create(&sA11a12b2, w , w2);
#else
    fgrim_create(&sA4a5b1  , w , w2); fgrim_create(&sA4a5b2  , w , w2);
    fgrim_create(&sA6a7b1  , w2, w ); fgrim_create(&sA6a7b2  , w2, w );
    fgrim_create(&sA9a10b1 , w2, w ); fgrim_create(&sA9a10b2 , w2, w );
    fgrim_create(&sA11a12b1, w , w2); fgrim_create(&sA11a12b2, w , w2);
#endif
    sA13a14 = out; // pointer copy
    fgrim_create(sA13a14, w, w);


    /* analysis */
  SAVE(sA1, "sA01.bmp");
    fgrim_copy(sA1, &sA2); // Not optimal, but we don't want to modify sA1
    frequency_shift_PI(&sA2);
  SAVE(&sA2, "sA02.bmp");
    downsample_quincunx_q2(&sA2, &sA4a5b1, &sA4a5b2);
  SAVE(&sA4a5b1, "sA04b1.bmp"); SAVE(&sA4a5b2, "sA04b2.bmp");
    TIMER_BEGIN;
      MACRO_filter_periodic_horizontal(&sA4a5b1, &p0_coef, offset1);
      MACRO_filter_periodic_horizontal(&sA4a5b2, &p1_coef, offset1);
    TIMER_END("horizontal")
  SAVE(&sA4a5b1, "sA05b1.bmp"); SAVE(&sA4a5b2, "sA05b2.bmp");
    reorganize_h2v(&sA4a5b1, &sA6a7b1);
    reorganize_h2v(&sA4a5b2, &sA6a7b2);
  SAVE(&sA6a7b1, "sA06b1.bmp"); SAVE(&sA6a7b2, "sA06b2.bmp");
    TIMER_BEGIN;
      MACRO_filter_periodic_vertical(&sA6a7b1, &p0_coef, offset1);
      MACRO_filter_periodic_vertical(&sA6a7b2, &p1_coef, offset1);
    TIMER_END("vertical␣␣")
  SAVE(&sA6a7b1, "sA07b1.bmp"); SAVE(&sA6a7b2, "sA07b2.bmp");
    polyphase_combine(&sA6a7b1, &sA6a7b2, &sA8b1, &sA8b2);
  SAVE(&sA8b1, "sA08b1.bmp"); SAVE(&sA8b2, "sA08b2.bmp");


    /* processing */
    //fgrim_set_color(&sA8b1, BLACK);
    //fgrim_set_color(&sA8b2, BLACK);
    //SAVE(&sA8b1, "sA08b1bis.bmp"); SAVE(&sA8b2, "sA08b2bis.bmp");


    /* synthesis */
    polyphase_combine(&sA8b1, &sA8b2, &sA9a10b1, &sA9a10b2);
  SAVE(&sA9a10b1, "sA09b1.bmp"); SAVE(&sA9a10b2, "sA09b2.bmp");
    filter_periodic_vertical(&sA9a10b1, &p1_coef, offset2);
    filter_periodic_vertical(&sA9a10b2, &p0_coef, offset2);
    //filter_periodic_verticalGPU(&sA9a10b1, &p1_coef, offset2); // no offset
    //filter_periodic_verticalGPU(&sA9a10b2, &p0_coef, offset2); // no offset
  SAVE(&sA9a10b1, "sA10b1.bmp"); SAVE(&sA9a10b2, "sA10b2.bmp");
    reorganize_v2h(&sA9a10b1, &sA11a12b1);
    reorganize_v2h(&sA9a10b2, &sA11a12b2);
    amplify(&sA11a12b1, 2);
    amplify(&sA11a12b2, 2);
  SAVE(&sA11a12b1, "sA11b1.bmp"); SAVE(&sA11a12b2, "sA11b2.bmp");
    filter_periodic_horizontal(&sA11a12b1, &p1_coef, offset2);
    filter_periodic_horizontal(&sA11a12b2, &p0_coef, offset2);
    //filter_periodic_horizontalGPU(&sA11a12b1, &p1_coef, offset2);
    //filter_periodic_horizontalGPU(&sA11a12b2, &p0_coef, offset2);
    amplify(&sA11a12b1, 4);
    amplify(&sA11a12b2, 4);
  SAVE(&sA11a12b1, "sA12b1.bmp"); SAVE(&sA11a12b2, "sA12b2.bmp");
    upsample_quincunx_q2(sA13a14, &sA11a12b1, &sA11a12b2);
  SAVE(sA13a14, "sA13.bmp");
    frequency_shift_PI(sA13a14);
  SAVE(sA13a14, "sA14.bmp");
  SAVE(sA13a14, "sA00.bmp");
  float diff = fgrim_diff(sA1, sA13a14);
  printf("diff=%f␣percent\n", diff );


    /* ending */
    sA1  = NULL;
    fgrim_free(&sA2);
    //fgrim_free(&sA4a5b1  ); fgrim_free(&sA4a5b2  );
    fgrim_free(&sA8b1    ); fgrim_free(&sA8b2    );
#ifdef GPU
    cudaMalloc_fgrim_free(&sA4a5b1  ); cudaMalloc_fgrim_free(&sA4a5b2  );
    cudaMalloc_fgrim_free(&sA6a7b1  ); cudaMalloc_fgrim_free(&sA6a7b2  );
    //cudaMalloc_fgrim_free(&sA9a10b1 ); cudaMalloc_fgrim_free(&sA9a10b2 );
    //cudaMalloc_fgrim_free(&sA11a12b1); cudaMalloc_fgrim_free(&sA11a12b2);
    fgrim_free(&sA9a10b1 ); fgrim_free(&sA9a10b2 );
    fgrim_free(&sA11a12b1); fgrim_free(&sA11a12b2);
#else
    fgrim_free(&sA4a5b1  ); fgrim_free(&sA4a5b2  );
    fgrim_free(&sA6a7b1  ); fgrim_free(&sA6a7b2  );
    fgrim_free(&sA9a10b1 ); fgrim_free(&sA9a10b2 );
    fgrim_free(&sA11a12b1); fgrim_free(&sA11a12b2);
#endif
    sA13a14 = NULL;
    fgrim_free(&p0_coef); fgrim_free(&p1_coef);
    TIMER_DELETE;

  return EXIT_SUCCESS;
}
```

B.2 GPU Horizontal Periodic Filter: host CUDA code

Below is the code of the function `filter_periodic_horizontalGPU` in `dfb_cuda.cu`.

```
int filter_periodic_horizontalGPU(fgrim *fg, fgrim *coef, int offset)
{

 float *h_Kernel, *h_Data; // host kernel and i/o data
 float *d_DataI, *d_DataO; // device global memory: i/o data
 TIMER_DECLARE;

 int data_w = fg->w;
 int data_h = fg->h;
 int data_size = data_w * data_h * sizeof(float);

 /* Preparing data */
 h_Data = fg->pix;
 cudaMalloc( (void **)&d_DataI, data_size);
 cudaMalloc( (void **)&d_DataO, data_size);
 // ~5 ms for both malloc and free for 2 calls

 /* Preparing filter coefficients */
 h_Kernel = coef->pix;

 /* Copy data and kernel to device */
 cudaMemcpyToSymbol(d_Kernel, h_Kernel, KERNEL_SIZE);
 cudaMemcpy(d_DataI, h_Data, data_size, cudaMemcpyHostToDevice); // 2 calls: ~15
     ms

 /* Prepare execution */
 dim3 blockGridRows(iDivUp(data_w, ROW_TILE_W), data_h);
 dim3 threadBlockRows(KERNEL_RADIUS_ALIGNED + ROW_TILE_W + KERNEL_RADIUS);

 /* GPU convolution */

 cudaThreadSynchronize() ;
 TIMER_BEGIN; // takes less than 1 ms
 convolutionRowGPU<<<blockGridRows, threadBlockRows>>>(
      d_DataO, // result
      d_DataI, // data
      data_w,
      data_h
 );
 cudaThreadSynchronize() ;
 TIMER_END("horizontal");

 /* Reading back GPU results */
 cudaMemcpy(h_Data, d_DataO, data_size, cudaMemcpyDeviceToHost);

 /* Shutting down */
 cudaFree(d_DataI);
 cudaFree(d_DataO);
 TIMER_DELETE;

 return(EXIT_SUCCESS);
}
```

B.3 GPU Horizontal Periodic Filter: device CUDA code

Below is the code of the function `convolutionRowGPU` in `dfb_cuda_kernel.cu`:

```
__global__ void convolutionRowGPU(
    float *d_Result,
    float *d_Data,
    int dataW,
    int dataH
){
 //Data cache
 __shared__ float data[KERNEL_RADIUS + ROW_TILE_W + KERNEL_RADIUS];
  //Current tile and apron limits, relative to row start
 const int         tileStart = IMUL(blockIdx.x, ROW_TILE_W);
 const int           tileEnd = tileStart + ROW_TILE_W - 1;
 const int        apronStart = tileStart - KERNEL_RADIUS;
 const int          apronEnd = tileEnd   + KERNEL_RADIUS;

 //Clamp tile and apron limits by image borders
 const int    tileEndClamped = min(tileEnd, dataW - 1);
 const int apronStartClamped = max(apronStart, 0);
 const int   apronEndClamped = min(apronEnd, dataW - 1);

 //Row start index in d_Data[]
 const int          rowStart = IMUL(blockIdx.y, dataW);

 //Aligned apron start. Assuming dataW and ROW_TILE_W are multiples of half-warp
      size, rowStart + apronStartAligned is  also a multiple of half-warp size,
      thus having proper alignment for coalesced d_Data[] read.
  const int apronStartAligned= tileStart - KERNEL_RADIUS_ALIGNED;

  int loadPos = apronStartAligned + threadIdx.x;
  //Set the entire data cache contents. Load global memory values, if indices are
      within the image borders, or initialize with values of the other side of
     the matrix for periodicity
  if(loadPos >= apronStart){
      const int smemPos = loadPos - apronStart;
      if ((loadPos < apronStartClamped) || (loadPos > apronEndClamped))
          loadPos = (loadPos + dataW) % dataW ; // periodic rows
      data[smemPos] = d_Data[rowStart + loadPos];
  }

  //Ensure the completness of the loading stage, because results, emitted by each
      thread depend on the data, loaded by another threads
  __syncthreads();
  const int writePos = tileStart + threadIdx.x;
  //Assuming dataW and ROW_TILE_W are multiples of half-warp size, rowStart +
     tileStart is also a multiple of half-warp size, thus having proper
     alignment for coalesced d_Result[] write.
  if(writePos <= tileEndClamped){
      const int smemPos = writePos - apronStart;
      float sum = 0;
      for(int k = 0; k <= KERNEL_RADIUS; k++) // k<=2*KERNEL_RADIUS
          sum += data[smemPos + k] * d_Kernel[k];
      d_Result[rowStart + writePos] = sum;
  }
}
```