NTNU
Innovation and Creativity

Analysis of fibre cross sections

Developing methods for image processing and visualisation utilising the GPU

Jørgen Bergquist
Helge Titlestad

Master of Science in Computer Science
Submission date: June 2006
Supervisor: Richard E. Blake, IDI
Co-supervisor: Per Nygård, PFI

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

Problem Description

Fibrous materials are an important part of our daily life. Applications span from textiles, baby's napkin, paper and cartons to more high performance applications as advanced composites within aerospace. All these materials are optimized for a specific use, and in general they can be decomposed into a bulk and a surface structure. To cover this large area of applications it is obvious that the fibre material, size, form and fibre network differ a lot depending on what function the product is supposed to fulfil. Detailed structural characterization of the fibres is important to understand the materials performance. One established method is to align fibres, embed them into epoxy, polish the cross sections and depict them with scanning electron microscopy (SEM). The following analysis of the cross sections is a manually done and time consuming job, and today improved description of the fibre cross-sectional details is often required.

There exist a need for improving the analysis both regarding methods for editing and analysing the images, gather the methods in a user friendly tool and develop effective graphical viewing on the screen. This master thesis will therefore be focused on the following tasks:

* Improve and eventually develop new methods for cross-sectional analysis of fibres.
* Join the methods into a user-friendly toolbox.
* Develop effective graphical viewing by using resources on the graphics card.

Assignment given: 01. February 2006
Supervisor: Richard E. Blake, IDI

Preface

This master thesis is the result of our diploma work, where we developed an application for cross-section image analysis of paper pulp fibres utilising the GPU.

Our goal is to develop methods for analysis of cross-section images utilising the GPU on well known algorithms which are parallelable, reducing the workload on the CPU and achieving faster interaction with the user, as well as faster computation of the algorithms. GPUs have been around for some years, but only lately achieved programmability. This has opened up for a new genre of general purpose computing on the GPU which we want to take advantage of.

The work has been carried out at *The Department of Computer and Information Science*, at the *Norwegian University of Science and Technology (NTNU)*, and in cooperation with *The Paper and Fibre Research Institute (PFI)*. The work was done by Jørgen S. Bergquist and Helge D. Titlestad[1], with professor Richard E. Blake at NTNU as thesis supervisor and with guidance from Per Nygård at PFI.

We want to thank Richard E. Blake and Per Nygård for their helpful suggestions and comments. Furthermore, thanks to Espen Brill for good discussions regarding algorithms, and our co-habitants for being patient and supportive. Data sets and work space have been provided by PFI.

This document is written using LaTeX. All code is formated and coloured by `listings`.

Trondheim, Juni 2006

Jørgen S. Bergquist Helge D. Titlestad

[1] [bergquis – titlesta]@pvv.ntnu.no

Abstract

In this master thesis we explore the possibility of taking advantage of todays generation of Graphical Processing Units, GPUs, performance when analysing digital images of fibre cross-sections. We implemented common algorithms such as the median filter, the SUSAN smoothing filter and various mathematical morphology operations using the high-level shader language OpenGL Shader Language, GLSL.

Modern graphics processing units, GPUs, have developed into high-performance processors with programmable vertex and pixel shaders. With these new opportunities a new subfield of research has emerge , dubbed GPGPU for General Purpose computing on the GPU. GPUs are now used in areas as oil exploration, processing of sound effects, neural networks, cryptography and image processing. As the GPUs functionality and performance are still increasing, more programmers are fascinated by their computational power.

To understand the performance of paper materials a detailed characterisation of the fibre cross-sections is necessary. Using scanning electron microscopy, SEM, fibres embedded in epoxy are depicted. These images have to be analysed and quantified.

When measured against equivalent image processing operations run on the CPU, we have found our GPU solution to perform about equally well. The operations run much faster on the GPU, but due to overhead of binding FBOs, initialising shader programs and transfering data between the CPU and the GPU, the end result is similar on the GPU and CPU implementations. We have deliberatly worked with commodity hardware to see what one can accomplish by just replacing the graphics card in the engineer's PCs. Using newer hardware the results would tilt heavily towards the GPU implementations.

We have concluded that making a paper fibre cross-section analysis program based on GPU image processing with commodity hardware is indeed feasible, and would give some benefits to the user interactivity. However, it is also harder to implement because the field is still young, with immature compilers and debugging tools and few solid libraries.

Contents

Glossary

AGP	Accelerated Graphics Port. An aging standard for connecting the graphics card to the PC.
API	Application Program Interface.
ARB	Architecture Review Board.
ATI	Graphics card manufacturer. Makes the Radeon series.
Branching	Splitting program flow into different branches.
CPU	Central Processing Unit - the processor on the main board.
FBO	Frame Buffer Object - an off-screen frame buffer.
Fragment	A per-pixel data structure created by the rasterization of graphics primitives.
Frame Buffer	region of graphics memory OpenGL render to, usually a window or the entire screen.
GLSL	OpenGL Shading Language.
GPGPU	General Purpose Computing on GPU.
GPU	Graphics Processing Unit - the processor on the graphics card.
GUI	Graphical User Interface.
HiST	Høgskolen i Sør-Trøndelag.
Lumen	Interior void of hollow paper fibres.
LUT	Look-Up Table.
MIMD	Multiple Instruction, Multiple Data.
NVIDIA	Graphics card manufacturer. Makes the GeForce series.
PBuffer	Pixel Buffer - an off-screen buffer
PCI-E	PCI-Express. An up-to-date standard for connecting expansion cards, including the graphics card, to the PC.
PFI	Papir- og fiberinstituttet.
Pixel	Picture Element - smallest discrete component of an image.
Pulping	The process of making pulp (No. papirmasse).
Qt	A GUI library from Trolltech.
Shader	A program that can transform the shape of objects, vertex shader, or the appearance of pixels, pixel shader.
SEM	Scanning Electron Microscope.
SIMD	Single Instruction, Multiple Data.
SUSAN	Smallest Univalue Segment Assimilating Nucleus.
UI	User Interface.

1 Introduction

1.1 Motivation

The birth of what we call digital image processing today can be traced back to the 1960s, when the first powerful enough computers to carry out meaningful image processing tasks appeared. Computer techniques for improving images have been used since the Jet Propulsion Laboratory transmitted pictures from the moon in 1964. The images from the Ranger 7 spacecraft were processed by a computer to correct various types of image distortions. In parallel with space applications, digital image processing techniques began in the late 1960s and early 1970s to be used in medical imaging, remote Earth resources observation and astronomy. From the 1960s and until now, the field of image processing has grown vigorously and processed images are now being used in a broad range of applications. With the fast computers and signal processors available in the 2000s, digital imaging has become the most common form of image processing. It is generally used not only because it is the most versatile method, but also the cheapest.

The main application of Graphics Processing Unit, GPU, has been in the entertainment business and computer games. Lately, the performance and functionality of GPUs have increased with the use of compilers for high-level programming languages, and other applications have been discovered. These take advantage of the parallelism and vector processing capabilities of the GPU.

In October 2002, with the introduction of the ATI Radeon 9700, pixel and vertex shaders could implement looping and lengthy float point math and quickly became as flexible as CPUs. Today parallel GPUs are used in a subfield of research, dubbed GPGPU for General Purpose Computing on GPU, in areas as diverse as oil exploration, processing of sound effects, neural networks, cryptography and image processing.

Modern GPUs feature some programmability in the form of 3D shaders, and should not be confused with general software programmability. GPUs operate as SIMD[1] or sometimes MIMD[2] parallel processors, and operations on rectangular arrays of coloured pixels are a prime application for these. Many array algorithms can be adapted to GPUs for extremely high throughput.

The GPGPU subfield is still in its early years and will expand a lot in the coming years along the development of future GPUs. Even though this subfield is somewhat immature we already can recognise its potential. We want to explore this new field in our diploma work for the master thesis.

Fibrous materials are an important part of our daily life. Applications span from textiles, baby nappies, paper and cartons to higher performance applications like advanced composites within aerospace. All these materials are optimised for a specific use, and in general they can be decomposed into a bulk and a surface structure. To cover this large area of applications it is obvious that the fibre material, size, form and fibre structure differ a lot depending on what

[1] Single Instruction, Multiple Data
[2] Multiple Instruction, Multiple Data

type of function the product is expected to fulfill.

To understand the performance of paper materials a detailed characterisation of the fibre cross-sections is necessary. The established method is to align the fibres, embed them into epoxy, polish the cross-sections and depict them with scanning electron microscopy, SEM. These images have to be analysed and quantified, and this is currently done manually which is a time-consuming process.

These images are noisy and have to be filtered, preferably by removing noise without loss of data. Then you have to segment the image and separate the fibres from each other. Finally, the person doing the analysis has to correct broken fibres and choose which fibres are going to be quantified or not. This last part is a very time-consuming task and is very difficult to do automatically. Hence an intuitive and efficient User Interface, UI, is very important. The workload in this final stage depends on how good the algorithms in the two previous stages are. When all this has been done, the selected fibres are quantified and summed up in a report.

1.2 Outline

In chapter 2 we describe previous work done at PFI and give some background information about GPUs and their pipeline. In chapter 3 we describe the work we have carried out, why we did it, and how it might me useful. Chapter 4 sums up our results, leading to chapter 5 where we hand out some suggestions for future work. Chapter 6 provides a short conclusion summarising the work and results of our diploma master thesis.

2 Background

In this chapter we present previous work at PFI, general information about image processing with the most common filters as well as some information about the GPU. We take a quick look at the GPU's pipeline and how to take advantage of it with regards to image processing.

2.1 Previous work at PFI

In 1998 Mads Brevik wrote several command line applications for preprocessing, segmenting and quantification. These programs are used by engineers at PFI, and require a lot of manual editing of the images. They have the advantage of being well tested so they are well suited for conformance testing.

In 2003-04 a group of students from HiST further developed Brevik's work into a library (pfi-lib) and built a GUI application (pfi-image) on top of it [Bergquist et al., 2004]. The project suffered from poor testing, hard to understand GUI code and less than perfect documentation.

Gary Chinga and Olav Solheim at PFI are currently developing some plug-ins for ImageJ for pre-processing, segmenting and quantification. So far only basic routines have been implemented.

2.2 Problem relevance for PFI

PFI was established as a self-owned foundation in 1923. Since then the institute has been among the important contributors to research and knowledge-building within the pulp and paper industry. PFI is a commercial organisation and an important goal is to enhance the competitiveness of its clients. A major part of their research is directly related to the needs of large industrial concerns, but many small and medium-sized businesses are also using their lab facilities for quality testing and product development.

Most research work at PFI is planned in close contact with their customers and is adapted to the customers needs. PFI's main task is to carry out long-term research of a more fundamental nature. The research programmes account for 75% of their activities, the remainder is contract work and consulting services.

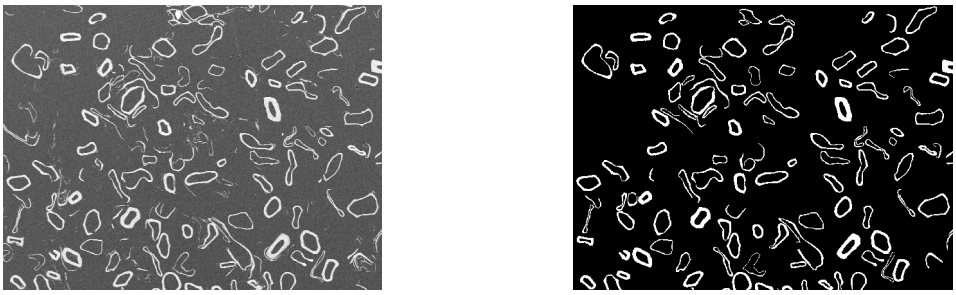
PFI is organised in two research groups; *Paper and novel materials* and *Fibre and pulp*. Each group has two core activities; *Paper* and *Novel materials* in one group and *Mechanical pulping and fibre characterisation* and *Adsorption and hygiene* in the other.

The core activity most related to our work is Mechanical pulping and fibre characterisation which involves fundamental knowledge as to how raw materials and production processes affect both the development of fibre properties and energy consumption.

One subsection is fibre cross-section analysis, where wet fibres are parallel aligned, freeze-dried and embedded in epoxy. Then images of fibre cross-sections are taken with a scanning electron microscope. The images are manually edited prior to calculation, where artefacts are removed,

damaged fibres are fixed and fibres which have grown together are separated.

Figure 2.1: (a) original, (b) edited



(*a*) (*b*)

Each fibre in the edited images are then measured in the following way:

- Cell wall thickness and its variation around the fibre circumference.
- Fibre wall area.
- Fibre cross-sectional area.
- Fibre and lumen perimeter.
- Fibre and lumen shape (collapse index).
- Cross-sectional compactness (z-value).

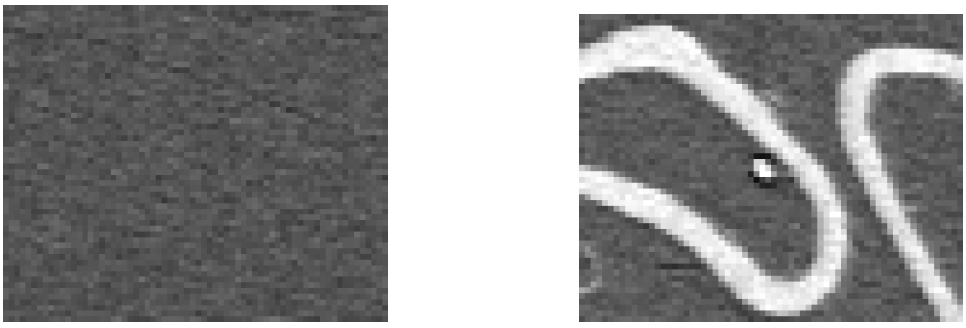
Results are given as average values and distributions for the studied fibre population. In addition the proportion of axially split fibres is reported. These results are used in comparison of fibre development for different fibre types, different species and various pulping processes, evaluation of screening and fractionation systems and processes and studies of relationships between fibre characteristics and paper properties.

2.3 Image processing

SEM images have several types of noise and artefacts from the image acquisition. Salt-and-pepper noise is scattered over the entire image, air bubbles may be trapped in the sample and make distortions. Finally the image usually has a "pillow effect" because of the method used for acquisition.

The objective of the preprocessing step is to separate all objects in the image from the background, and remove all noise.

Figure 2.2: (a) Salt and pepper noise. (b) An air bubble lies close to the fibre wall.



a **b**

Ideally the background would have a uniform grey-level throughout the image. However, as figure 2.2(a) shows, the background is far from uniform. Using threshold segmentation on an image with no preprocessing would find false positives, i.e. foreground objects which should have been labelled as background. We would also get false negatives, usually background labelled holes in fibre walls which should be labelled as foreground.

Figure 2.2(b) shows another artefact of the sample preparation process: Air bubbles can get trapped in the sample and distort the image, resulting in a black spot. Sometimes grinding dust get into these air bubbles resulting in a black ring around a bright white spot. Finding the air bubbles is not hard because the black ring is darker than the background grey-level and the white spot is usually whiter than the fibres. However, this information is lost after binarization, so if we want to address the air bubbles, we will have to do it in the preprocessing step.

2.3.1 Median filter

The median filter is a well-known non-linear filter for noise reduction. It is described in detail in [Gonzalez and Woods, 2002]. While most implementations use a square kernel of neighbourhood pixels (3x3, 5x5, etc.), our implementation can handle other kernels (e.g. a round kernel with radius 2 and total of 21 pixels) equally well.

Most published methods for doing median filters on GPUs use a multi-pass algorithm, either with a separable median filter or by subdividing the colour space. The main problem is sorting when applying a median filter on the GPU. Sorting inherently requires a lot of conditionals (i.e. if-tests) which is not what the GPU was designed for. Furthermore, looping over an array with a dynamic index is not supported by the GPU, so the compiler will have to unroll the sorting loops. When there is a large kernel to loop over we quickly approach the maximum number of instructions. By using a separable median filter we could first find the median of each row in the kernel, and finally find the median of those median values. Such an approach will support a much larger mask, but will lose some information because the output value is not always the true median of the kernel. A different approach is described in [Viola et al., 2003] which requires $log_2(256) = 8$ passes on our 256 grey levels, using a binary search to find the correct median.

2.3.2 SUSAN smoothing filter

The SUSAN (Smallest Univalue Segment Assimilating Nucleus) algorithms are a set of algorithms for smoothing, edge detection and corner detection based on the SUSAN principle (described in [Smith and Brady, 1995]). Several earlier applications ([Hagen and Holen, 2004], [Henden and Bache-Wiig, 2005]) at PFI have used SUSAN smoothing on fibre material even on 3D material.

The SUSAN smoothing algorithm tries to preserve edges in the image while smoothing the surfaces. This is an important feature for us because it keeps the fibres walls intact and prohibits fibres from growing into each other. If we compare it to the median filter, it might preserve a little too much information. Small objects which would be removed by the median filter will often survive the SUSAN filter. We are of the opinion that it is better to have too much information than too little, so we suggest using a minimal 4-neighbour median filter first and then applying SUSAN in several iterations afterwards. Small remaining objects can easily be removed in post-processing so they should not be a big problem.

2.3.3 Mathematical morphology

Mathematical morphology uses a mathematical interpretation of grey-level to do low-level image operations. In our case the images have either 256 (grey-scale) or 2 (binary) grey-levels. With mathematical morphology we interpret the grey-levels as a topographical surface where each grey-level is a subset of the layer below. For binary images we only have two levels: Foreground pixels and the entire image.

Many morphological operations can be expressed as a series of the two most basic operations: Dilation and erosion. Dilation of an image A with a structure element B is the union of all coordinates where B can be translated to and intersect with A. Similarly, erosion is the union of all coordinates where B can be translated to and is completely contained in A. Although most operations can be expressed as the combination of erosion, dilation and common set operators, they will usually be coded a bit more compact to avoid using unnecessary passes and storage space.

2.3.4 Thresholding

In a grey-scale image composed of light objects on dark background, one way of extracting the objects from the background is to select a threshold that separates the two. The simplest of all thresholding techniques is to partition the image histogram by using a single global threshold. Segmentation is then accomplished by scanning each pixel and label them as object or background, depending on whether the grey level of the pixel is greater or less than the threshold value. The success of such a method depends entirely on how well the histogram can be partitioned.

One way of obtaining the threshold value T automatically:

1. Select an initial estimate for T, often average grey level of the image.
2. Segment the image using T. This will produce two groups of pixels: G_1 consisting of $> T$ pixels and G_2 consisting of $\leq T$.
3. Compute the average grey level values μ_1 and μ_2 for the pixels in regions G_1 and G_2.
4. Compute the new threshold value: $T = \frac{1}{2}(\mu_1 + \mu_2)$.
5. Repeat step 2-4 until the difference in T in successive iterations is smaller than a predefined parameter T_0.

2.4 The Graphics Pipeline on a GPU

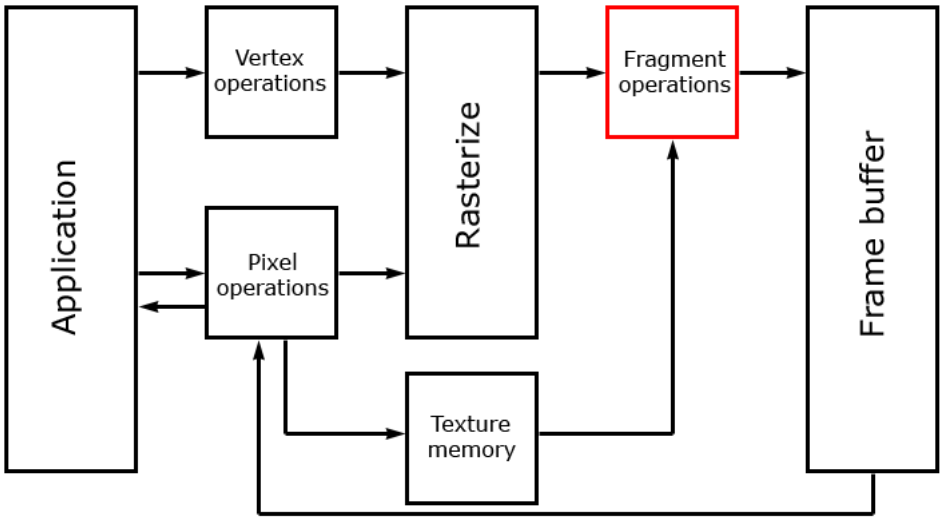


Figure 2.3: Graphics pipeline

Figure 2.3 depicts a very simplified view of the graphics pipeline on a GPU. Modern GPUs let us program the Vertex operations and Fragment operations with specialised programs, called

shaders. In our project we will only work with fragment shaders, as the geometry, vertices, are only simple rectangles. Next, see a more detailed description of the operation pictured in figure 2.3

2.4.1 Vertex operations

Geometry data is passed from the application to the GPU as 3D points, vertices, in local object space. In this step the GPU transforms the vertices by multiplying them with several matrices to get on-screen coordinates. It also performs clipping, and sets up per-vertex texture coordinates and colour values for use later in the pipeline.

Much of this step can be programmed with vertex shaders. We will however mainly work in two dimensions on texture data, so the fixed functionality pipeline is sufficient.

2.4.2 Pixel operations

Pixel data can be transferred from the application to textures on the GPU, from the GPU framebuffers to texture memory and finally back to the application. This step also converts pixel data if the formats differ between the GPU and the application.

2.4.3 Rasterize

During rasterization the primitives from the vertex operations is converted to fragments. Each fragment corresponds to a pixel in the frame buffer.

2.4.4 Fragment operations

The GPU takes samples from textures in the texture memory by looking up the per-fragment texture coordinates from the rasterizer. For normal 3D rendering these samples are colours which are applied to the fragment on the frame buffer.

This step is programmable with fragment shaders, which can do arbitrary texture lookups. Thus a texture can represent any form of lookup table, e.g. a convolution matrix, dilation mask etc. With the latest generation of GPUs a fragment shader can output to several different frame buffer objects, FBOs, per fragment.

Fragment shaders are executed on the GPU in parallel on specialised hardware. This results in massive potential speed gains on easily parallelized algorithms.

2.5 OpenGL and imaging

OpenGL is one of the most popular graphics APIs (Application Program Interface) for real-time graphics processing, the other one is Microsoft's Direct3D. One of OpenGL's strengths is that it includes facilities for imaging as well as 3D rendering. Since OpenGL version 1.2 it has supported convolutions, colour space conversion and histogram generation with the OpenGL Imaging Subset [Shreiner et al., 2004]. OpenGL's platform independence is also valued by the typical academic imaging developers.

Although OpenGL has supported a few vital imaging operations, it has seen limited use because of the fixed nature of the graphics hardware. With the latest hardware much of the formerly fixed pipeline has become programmable, thus making OpenGL a more viable choice for imaging [Rost, 2004].

2.5.1 OpenGL Shading Language

There are many ways to program the fragment and the vertex processors, including several high-level programming languages (as we have documented in [Bergquist and Titlestad, 2005]).

We decided to use GLSL for our master thesis. GLSL is currently the highest level shading language and it has very good integration with the rest of the OpenGL pipeline. The compiler is a part of the OpenGL driver which makes it easier to optimise for the hardware, but a bit harder to debug because we can't look at any generated assembly code. We have used compilers from both NVIDIA and ATI.

GLSL is fully integrated in OpenGL 2.0. To get OpenGL 2.0 support on all platforms we use the GLee (GL Easy Extension) library.

2.5.2 Limitations

We have encountered limitations in both the GLSL compilers we have used. The ATI compiler did not support the `for` keyword at all, while the NVIDIA compiler requires all loops to be unrollable, because the GPU only allows static indexing into arrays. Listings 2.1 and 2.2 shows a simple loop before and after unrolling.

Listing 2.1: A simple loop

```
1 int array[3];
2 int sum = 0;
3 for (int i = 0; i < 3; i++) {
4     sum = sum + array[i];
5 }
```

Listing 2.2: The same loop after unrolling

```
1 int array[3];
2 int sum = 0;
3 sum = sum + array[0];
4 sum = sum + array[1];
5 sum = sum + array[2];
```

The shader programs can only handle a limited number of instructions. If we use many instructions within a loop which has to be unrolled many times, we will easily reach this limitation.

The limitations above are rooted in hardware: The GPU has limited support for branching and flow-control. Another limitation is the immaturity of the compilers themselves. NVIDIA's compiler sometimes uses more temporary registers than the hardware allows, resulting in a linking failure which is nearly impossible to debug.

Originally texture caching is optimised for reading a small amount of texture data from a small subsection of a large texture. When we design algorithms for the GPU we will have to keep this in mind, and never read data from random locations in a texture. Cache misses can be even more costly on the GPU than on a modern CPU. See [Buck, 2005] for an in-depth discussion of performance and cache misses.

2.6 GPGPU techniques

2.6.1 Converting scatter to gather

Many computer algorithms, including image processing algorithms, use a scatter technique to write data to a location that depends on the input data. Any C code like `a[i] = x;` is a scatter operation where `i` determines where x is written to. However, the fragment processor, which is our most important tool for computing on the GPU, is not able to do scatter operations. It can read from computed locations (using texture look-ups), but can only write to its preassigned pixel.

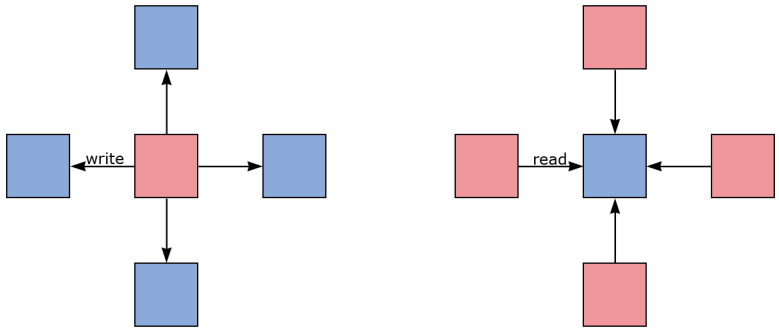


Figure 2.4: Left: Scatter operations compute where to write to. Right: Gather operations compute where to read from.

A prime example of a scatter algorithm is the standard way of computing a histogram for an image:

Listing 2.3: Computing a histogram

```
1 foreach pixel p in image
2 {
3     histogram[p.value] += 1;
4 }
```

For some problems we can convert the algorithm from scatter to gather and use them efficiently on the GPU. For other problems, like finding a histogram, it is more efficient to use an altogether different algorithm. We can also use the vertex shader to do scattering by rendering points and letting the vertex shader decide where to put the points, but this method fails if several points are put at the same position, and is way less efficient than rendering large rectangles.

To convert a scatter operation to a gather operation we can use an address sorting technique [Buck, 2005]. For each value we compute, we also store the address it would be saved at. Then we sort the data by its address field and finally collapse succeeding data elements where the address is the same.

2.6.2 Stream operations and stream reduction

A common way to look at GPGPU operations as opposed to normal CPU operation is to view the data as a stream, with the GPGPU program acting as a filter or a kernel acting on the stream. In listings 2.4 and 2.5 we show what this concept would look like with a normal programming language.

Listing 2.4: Normal program flow

```
1 for (int x = 0; x < image.x(); x++)
2 {
3     for (int y = 0; y < image.y(); y++)
4     {
5         image[x,y] = image[x,y] * 2;
6     }
7 }
```

Listing 2.5: Stream based program flow

```
1 Program program = "output = input * 2;";
2 filter.setProgram(program);
3 applyFilter(image, filter);
```

When you look at the flow chart in figure 2.5 it becomes apparent that the process can be conceptualised as data streams and filters. The fragment processor gets programmed with the fragment shader, then vast amounts of data pass through it and the program is applied to the data as they pass by. Algorithms which fit well with the streaming analogy tend to be effective on the GPU. Algorithms which only operate on small amounts of data in a sequential fashion are less efficient since they effectively make the stream narrower.

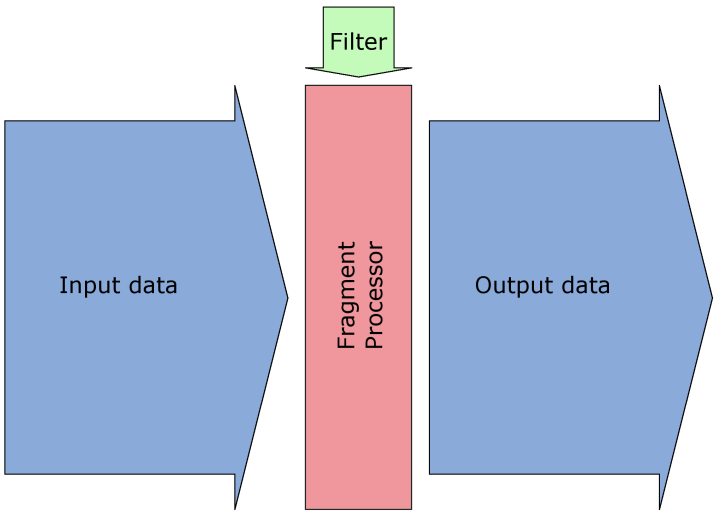


Figure 2.5: The stream model. Lots of data flows through the processor, and the filter is applied to it.

A lot of image processing operations return something other than an image, for example a maximum value operation. Since we can't loop through the image and remember the maximum value we will have to find it in another way. A common technique to solve such problems is to use stream reduction. We compute many local maximums for subsets of the input stream, and pass the reduced output data on as input data for the next iteration. This operation is repeated until we are left with only one output value.

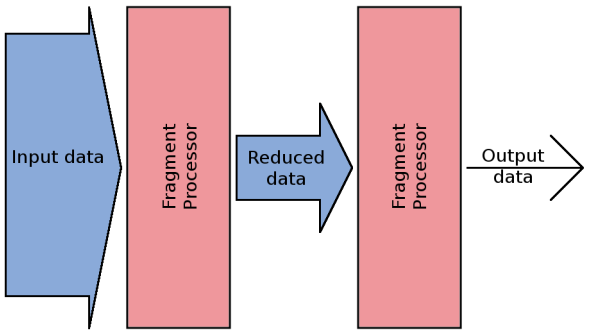


Figure 2.6: Stream reduction. The data stream is reduced in iterations until we have a single output value.

3 Implementation

3.1 Introduction

In this chapter we look at what we have implemented, i.e. the program we have written for PFI. The general work flow of the program, how we store data and miscellaneous filters we have implemented.

3.1.1 Program work flow

The general work flow of the program is shown in figure 3.1:

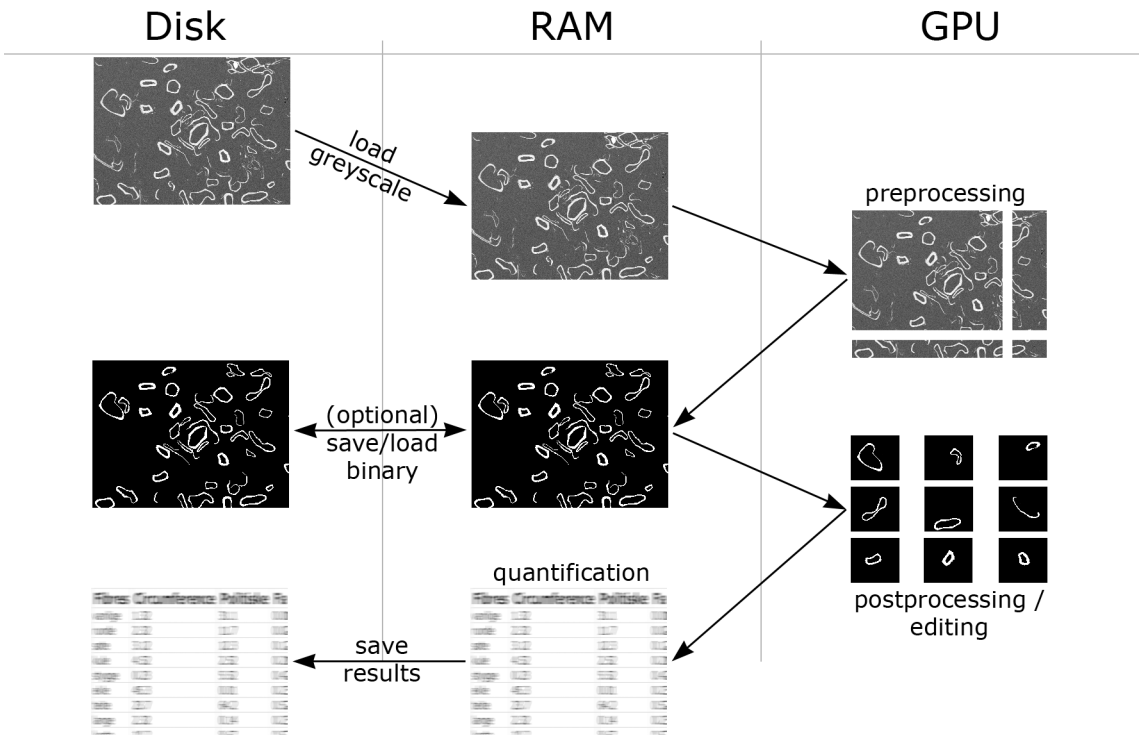


Figure 3.1: The program work flow.

1. Load a greyscale image from disk.
2. Divide it so it will fit in textures and upload it to the GPU.
3. Do preprocessing (noise filtering and thresholding) on it there.
4. Download the image back to RAM.
5. Run the last part of the segmentation (labeling).

6. Upload each individual object as an image to the GPU.
7. Do post-processing (generate a skeleton, etc.) and manual editing on the objects.
8. Download the objects back from the GPU and do quantification (compute circumference, area, average width, etc.).
9. Save the results to disk.

Step 2 is an important limitation in our program. The images are generally too large to fit in a single texture with the current generation GPUs; the two graphics cards we have tested have had 1024x1024 and 2048x2048 as texture size limits and our input images are 2560x1920. As a result we have to split up the images into smaller sub-images before uploading them. To avoid edge artefacts in the middle of the image we make the images overlap with enough pixels to keep the artefacts within the overlap. Splitting the image into smaller sub-images can have adverse effects on GPU performance because we have to change FBOs several times per filtering pass.

3.1.2 Using color channels for storing data

Throughout our program we pass the images around with four channels (R, G, B and A) and 8 bits per channel for a total of 32 bits. The four channels are normally used to store the red, green, blue and alpha components of an image. The GPU is designed to work on these four values simultaneously, so even though our input image only has one channel we try to use all the available channels as much as possible.

Many imaging operations require temporary images to be allocated. Instead of allocating extra space we try to use the four channels as efficiently as possible. Under preprocessing we use the R channel to store the filtered input image and the G channel to store the binary thresholded image. The B and A channels are used to store results from various mathematical morphology operations.

Figure 3.2: From left to right: The R, G, B and A channels of the image during post-processing.

In the postprocessing step we use the four channels similarly: We store the filtered input image in the R channel so the user can see the original greyscale image while simultaneously editing the binary image. The G channel is used to store the binary image where all the editing is done and on which the final quantification is done. The B channel is used to store a thinned version of the binary image and the A channel is used to store the distance transform, where each pixel stores its euclidean length from the skeleton in channel B.

If we had needed more channels, several of the channels mentioned above could be merged into one. In fact, there is more than enough precision on the standard GPU datatypes to store the input image (8 bit), the binarised image (1 bit), and the distance transform (7 bit) in a single channel. In cases where pixels are more than 128 pixels (the 7 bit boundary) away from the skeleton, they probably should be removed from the image anyway, so 7 bit should be quite enough to store the distance transform. To keep the complexity down we have chosen to not merge channels prior to it is really needed.

3.2 Preprocessing

In our implementation we upload the image to the GPU, run several filters on the image, and

download it back to CPU space for labelling. A "filter" here is a series of render-to-texture pass with a fragment shader program onto an FBO. We use two textures in this process, one for reading and one for writing, and swap which texture to read from after each pass.

3.2.1 Median filter

Even with the limitations of a naive median filter on the GPU, we have chosen it for our implementation. The kernel size should never exceed 5x5 simply because we would lose too much information. Even with a 3x3 kernel there is much information loss compared to a more advanced smoothing filter such as SUSAN. Using a multi-pass implementation also generates more overhead from binding FBOs.

To find the median value we use a simple bubble sort algorithm to sort the neighboring pixels, implemented as a double for-loop. We have included if-tests to stop the sorting when we have sorted half the data, but we suspect that the if-test effectively is ignored by the current generation GPUs.

An alternative to using bubble sort would be to construct a sorting network to sort the neighboring pixels. A sorting network is a series of max-min operations on pairs of input values, and is a bit more suited for sorting on the GPU, since they don't use flow control. However, using sorting networks is less flexible because we can't vary their size in run-time like we can with the bubble sort. As shown in chapter 4 our bubble sort median filter is also fast enough for our use.

3.2.2 SUSAN smoothing filter

Implementing SUSAN on the GPU was not as tricky as we had imagined. The original SUSAN source code is very low-level C-code, using pointer arithmetics liberally and several LUTs (look-up tables) for increased performance. Pointer arithmetics don't work at all on the GPU, so we converted it to normal integer iterators. Two LUTs are pre-computed, one which gives nearer pixels higher priority and one to do the brightness thresholding. While implementing LUTs as textures on the GPU would not be a big problem, we have chosen to do the computation in the inner loop instead. The reason for this decision is that random texture lookups are much more susceptible to cache misses. GPUs are better optimised for crunching numbers than looking up random data.

While implementing the SUSAN filter we had a few compiler related problems. The shader program is very close to the maximum instruction count after loop unrolling. In the final shader version there are a number of unnecessary instructions which could be removed to give more room to do other stuff, but when we tried to remove them the compiler ran out of temporary registers. This is clearly a compiler limitation (it used more registers with our simplified program, resulting in an error), which highlights how immature the GPU compilers are.

3.2.3 Thresholding

The first part of segmentation is binarization; labeling each pixel in the image as either background or foreground. We have used a global thresholding method to do binarization, where we get the threshold level from the image's histogram.

We know the histogram of our input images consist of two large peaks and automatically obtain the threshold value by locating these two large peaks and selecting the mean of the two values. This is a simplification of the threshold method suggested by [Reme, 2000], where the value is set to $\frac{1}{3}$ between the baseline of the peaks (see figure 3.3).

To generate the histogram we use a fragment shader to gather the amount of fragments at each intensity through the use of occlusion queries[1]. For a histogram with n buckets we use n passes, where each pass discards all the fragments outside the range of the bucket. The occlusion query

[1] Occlusion queries are described in detail in [Sekulic, 2004].

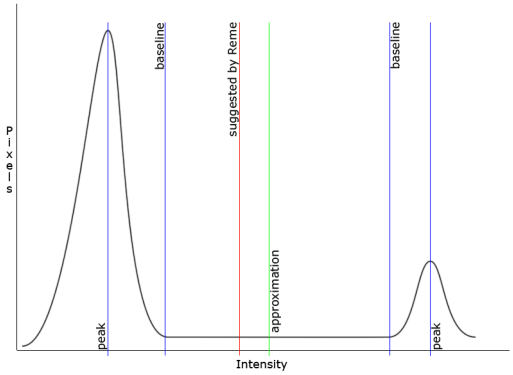


Figure 3.3: Histogram thresholding

object reports how many fragments got rendered, and the value is stored in the bucket. For a perfect histogram we would have to use $n = 256$, but in practice a third of that is good enough for deciding a threshold value.

An alternative approach would be to use stream reduction as discussed in 2.6. With stream reduction we would make many histograms for small parts of the image (say 16x16 pixels), and then add the small histograms together in multiple passes until we have one histogram for the entire image. This would take less passes than the occlusion query approach, but would be much more complex to implement because of the way we split images into sub-images to fit on a texture.

3.2.4 Labeling

Because downloading an image from the GPU to the CPU is a very costly operation, we tried to do everything between loading the image and quantifying the data on the GPU. This would involve the second step of segmentation, labeling. In labeling we assign a unique label to each object separately such that every pixel in the object has the label. This is usually done by looping through all the pixels in the image and flood filling all unlabeled objects with a unique color.

Unfortunately the "looping through all pixels" part is impossible on the GPU, because the pixels are processed in a parallel manner. In [Ueland and Botnen, 2005] they use a seeded region growing method, where the user selects a point within the object which the GPU uses to grow a region until the region matches the entire segment. This user-assisted approach is not good enough for our purpose, where we have many regions and try to minimize the workload of the user.

Instead of having a seed selected by the user, we label every pixel in a region by the pixel in the region with the lowest x and y coordinates. We use the B and A channels of each pixel to store the x and y coordinates they point to. They point to the lowest, leftmost pixel they know is in the region. The algorithm uses pointer doubling to find the right pixel to point to. Figure 3.4 shows an example where all the pixels in a segment are made to point to the same pixel in four iterations.

1. Set all foreground pixels to point to themselves.
2. For all pixels p:
   (a) For all 8 neighbouring pixels $p_{neighbour}$:
   (b) Set p to point to the minimum of the pixels p and $p_{neighbour}$ is pointing to.
3. Repeat step 2 until no pixels are changed.

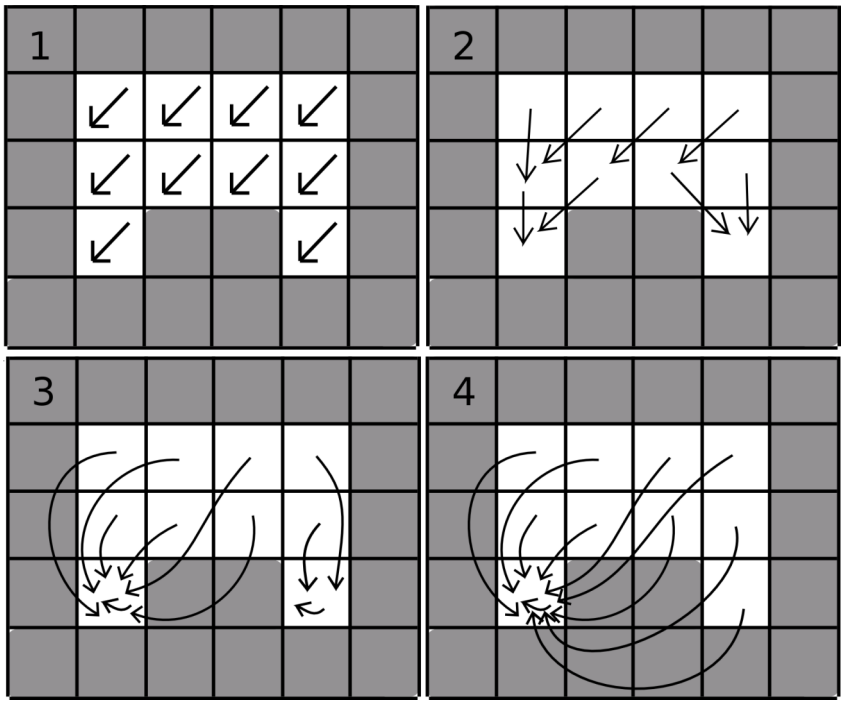


Figure 3.4: Labeling a segment in four iterations.

The complexity of this labeling algorithm is $\Omega(\log n)$ in the best case, when no pixel points to local minima, because of the pointer doubling algorithm we use. In the worst case nearly all pixels will point to a local minimum, and we only get one changed pixel per iteration, so we have a worst case complexity of $O(n)$ – that is n iterations to ensure that a n pixel large segment is labeled similarly.

By changing the algorithm to only checking its four nearest neighbours instead of eight, we get four-connected segments instead of eight-connected segments.

A big problem with this approach is where segments are split by sub-image boundaries. If the segment is split in two by the sub-images, the algorithm would find two segments, one for each sub-image. We would have to use a post-processing pass to merge the two parts of the segments. Doing the post-processing pass on the CPU would be quite easy, but that would defeat the idea of never downloading the image back to the CPU. Doing it on the GPU would add much complexity if at all possible.

3.3 Mathematical morphology

We use a lot of mathematical morphology operations, especially in the post-processing stage. Implementing mathematical morphology operations in a fragment shader is quite easy compared to more intelligent operations, and they perform quite well as we will show in chapter 4.

Many morphological operations such as watershed and thinning are run in iterations until the image converges, i.e. no pixel is changed in an entire iteration. We have implemented this on the GPU with occlusion queries, which is a way to count how many fragments where rendered. When the query reports zero rendered fragments we know the image has converged.

3.3.1 Segmenting

Although we mostly used morphology to do post-processing, we did try to use a hybrid thresholding-and-watershed algorithm for segmenting. The idea was to use a conservative global threshold where the threshold value is slightly higher than the optimal value, i.e. the regions are slightly

smaller than they should be. We run GPU-based labeling as discussed above, and then use the watershed morphological algorithm from the conservative threshold and down to the assumed optimal threshold. The effect would be that segments that are connected at the optimal threshold, but not at the conservative threshold, would be considered as two individual segments. This is often the case where two fibres lie close to each other.

3.3.2 Finding a skeleton

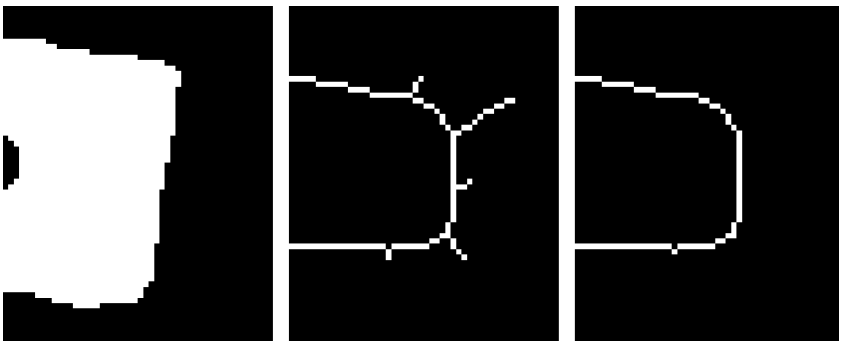


Figure 3.5: A part of a segment, the skeleton found by morphological thinning, and the result of applying morphological pruning on the skeleton.

The skeleton of a fibre is used to compute its length and average width, and it's also used by many post-processing algorithms. To find the skeleton of an image we first use morphological thinning with a set of four 3x3 kernels. The algorithm is run until it converges on a skeleton (i.e. it runs an entire iteration without removing any pixels). The process is described in detail in [Davies, 2005]. The basic morphological thinning operation leaves a lot of "spurs", as illustrated in figure 3.5.

To remove the spurs we use another set of kernels a fixed number of times (not until convergence) to avoid removing the actual skeleton. This is called pruning. All algorithms for finding a skeleton have to balance the amount of spurs versus not removing parts of the optimal skeleton. The problem domain helps us a bit because all whole, non-split fibres have exactly one hole in the segment, the lumen. So if the segment has a hole we can prune away all spurs without losing information. This obviously does not work for split fibres, which have no holes in the segment at all.

The thinning and pruning operations were easy to implement by passing the right kernels to the hit-or-miss shader.

3.3.3 Distance transformation

Distance transformation is the process of labeling each pixel in a set with the distance to some object. A popular use of distance transformation is in guided thinning; the pixels in a segment are first labeled with the distance to the nearest background pixels, and the distance transform is used as a topographic map to find the "ridge" in the middle of the segment. After some experiments we found the standard thinning algorithm described above to be more suited for GPU computation.

The distance transform is also used to remove fibrils. After finding the skeleton, we label all pixels with the distance to the skeleton. Pixels that are further away from the skeleton than a given confidence interval, are removed from the segment. With the right confidence interval only pixels that are part of fibrils or other statistical anomalies are removed.

3.4 Visualization

The most cumbersome part of the old system in use at PFI is the way the images are edited.

The thresholded image and the original are loaded as two files in an image processing program, with one window for each image. The engineer edits the thresholded image and has to scroll the original image to see what should be edited.

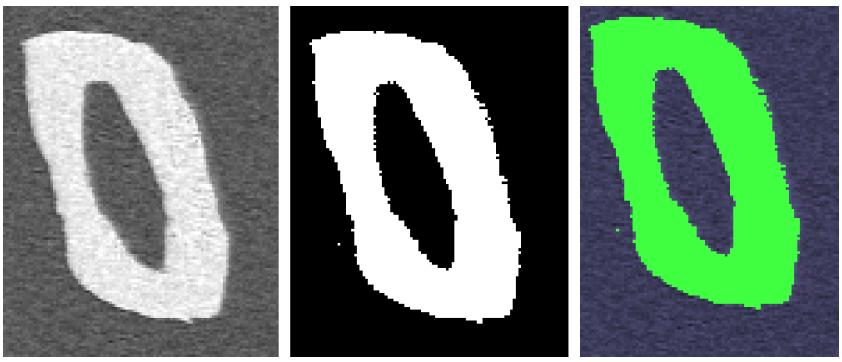


Figure 3.6: A fibre from an input image, the same fibre after thresholding and a blend of the input and thresholded images.

In our implementation we have stored the input image together with the thresholded image, skeleton and distance transform in the four different channels of the same texture on the GPU. How the data are displayed on the screen is entirely up to us. We can choose to only use one channel, or blend the channels together in any way we want. When choosing a threshold or editing the thresholded channel this allows us to show the original image in the background. Figure 3.6 shows one way of blending images.

4 Results

In this chapter we present the results we achieved through our work with the GPU. We will compare our GPU-results with the more conventional methods and look at our results of the image processing.

4.1 Performance: GPU vs CPU

All our measurements have been conducted on an Intel Xeon 2.66GHz with a GeForce 6600 graphics card. We wanted to find out if using GPGPU methods on commodity hardware would be feasible for analysing paper fibre cross-sections. If we had used state-of-the-art equipment we would probably have seen even bigger differences since graphics cards have evolved much more than CPUs the last few years.

We compare our results with the results of ImageJ. ImageJ is a java application with several image processing filters. PFI is using ImageJ and has been and is still developing plug-ins for it. There exist ports of the SUSAN filter to ImageJ, but we prefer to use the original C-code provided by Stephen M. Smith, [Smith and Brady, 1995], in our comparison on the SUSAN filter. The ImageJ SUSAN filter is just a Java wrapper on top of the original C-code, and because we are using C++ and not Java, comparing against the C-code is more faire to the original implementation. On the histogram we only mention our times on the GPU since histogram generation will always be very fast on the CPU. The time unit is milliseconds (ms).

Histogram	GPU Shader average (256 buckets)
128x128	29ms
2048x2048	1275ms
2560x1920	1656ms

Histogram generation is a good example of an algorithm that the GPU architecture doesn't handle very well. Histogram generation is way faster on the CPU since it only requires a simple loop over the image. Our shader version has to go over the images *buckets* number of times as opposed to the CPU version which only needs one pass. Since we only need to run the histogram generation algorithm once this won't affect the performance as much as for example SUSAN and various mathematical morphology filters that are run several times.

Median	ImageJ	GPU Shader average
128x128	15ms	0.5ms
2048x2048	2985ms	112ms
2560x1920	3516ms	147ms

The median filter is a very simple filter, which we thought wouldn't gain much improvement being run on the GPU. Actually, for smaller images we were right. When the images start to get very large, on the other hand, we see that the median filter scales much better on the GPU.

SUSAN	C-code	GPU Shader average
128x128	28ms	35ms
2048x2048	6880ms	767ms
2560x1920	8828ms	973ms

A more complex filter, as the SUSAN smoothing one, gains much more from being implemented on the GPU. About ten times faster on large images is a noticeable difference while working on them.

Dilate	ImageJ	GPU Shader
128x128	16ms	0.3ms
2048x2048	156ms	44ms
2560x1920	187ms	59ms

GPUs seem almost designed for mathematical morphology, as they operate in massive parallel and are optimised for close neighbourhood data look-ups. However, some of the simplest morphological operations (like thinning and watershed) require many passes and do very little computing per pass. For an algorithm to be more effective on the GPU than on the CPU it should do a minimum number of operations per input value (for current generation hardware the number is about 8, found with heuristics at SINTEF [Dokken, 2006]). We have only timed our dilation algorithm, but all mathematical morphology algorithms will have similar results because their implementations are very similar.

Overhead	Median	Susan	Dilate
128x128	250ms	687ms	248ms
2048x2048	281ms	735ms	280ms
2560x1920	313ms	796ms	312ms

On large images where a filter is going to be applied many times in succession, the GPU will outperform the CPU equivalent. But because of the overhead of binding FBOs and initialising shader programs the GPU won't perform as well on smaller images and with only one pass. We also found the overhead of transferring data between the CPU and GPU to be noticeable, especially when downloading data from the GPU to the CPU.

If we are not able to run all the algorithms on the GPU, but have to download it to the CPU for some operations and back to the GPU afterwards, the overhead of transferring data would be greater than the time reduced by running the filter on the GPU.

As described in 3.1.1 we usually want to run one pass of median, followed by some passes of SUSAN, a pass of histogram generation and a thresholding pass. After these steps we want to segment the image, but since advanced segmentation algorithms are complex and inefficient on

the current generation GPUs, we would have to push the data back to the CPU for segmentation. After segmentation we do post-processing to correct errors on fibre segments and select which segments to quantify. Because of all the overhead of pushing data around we would be better off just doing everything up to the final post-processing on the CPU with todays hardware and compilers. If an application needs to run a filter like the SUSAN filter a couple of times, on the other hand, using the GPU would be a good idea.

Our attempt at doing all the segmentation on the GPU, as mentioned in 3.2.4, did not perform well compared to a CPU-based approach. As we mentioned it had a worst-case complexity of $O(n)$. That means it has to run the fragment shader on all the pixels in the image – whether they are foreground or background pixels – a number of times equal to the length of the worst case segment. The effect is that we do very much computation for a very small change per iteration. One way to combat this would be to use a stencil buffer to mask out all the background pixels. Unfortunately the current implementations of FBOs do not support stencil buffers.

In [Ueland and Botnen, 2005], they had performance problems when doing many iterations in their seeded region growing segmentation algorithm. They operated on 3D data, using many 2D textures slices to make a 3D volume. The problems they had seem to be related to the overhead of frequently changing FBOs while running the algorithm. We have experienced some of the same problems for large input images that have to be split up into smaller images before processing. However, this limitation is gone with the latest generation of GPUs which handle 4096x4095 pixel large textures or more, easily large enough for our 2560x1920 input images.

4.2 Image processing results

Both for the median filter and SUSAN filter we use a simple bubble sort when finding the median. Since the GPU works in parallel it would be much faster to sort using a sorting network, but since we are only sorting a few numbers we decided not to take the time to implement a sorting network. We reckon the time saved will be negligible.

The SUSAN smoothing filter, described in 2.3.2, preserves edges while smoothing as opposed to the median filter that smoothes everything. The problem with the median filter is that smoothing everything makes close fibres grow together, as shown in figure 4.1.

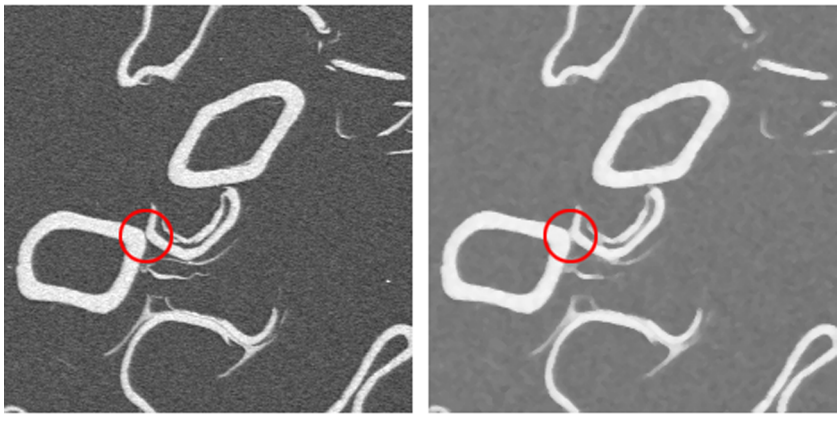


Figure 4.1: a) original image. b) median filtered image.

We found our SUSAN filter to be too preserving, leading to false positives during thresholding, as seen in figure 4.2 b). Our solution is to use one pass of median filter with a small 4-connectivity kernel to remove the worst salt-and-pepper noise first. Then using at least one (usually more) passes of SUSAN afterwards depending on the image at hand, figure 4.2 c).

In figure 4.3 you can see the difference between 50 passes of median and 50 passes of SUSAN. We clearly see how the SUSAN filter preserves the fibre structures better than the median filter.

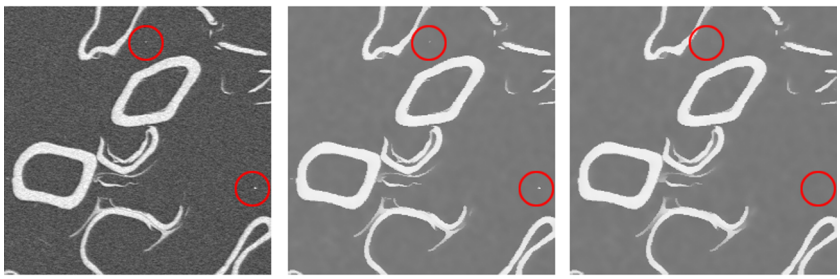


Figure 4.2: a) original image. b) SUSAN filtered image. c) one pass of median then SUSAN filtered image..

Also notice how the SUSAN filter preserves one of the noise particles through all the 50 passes.

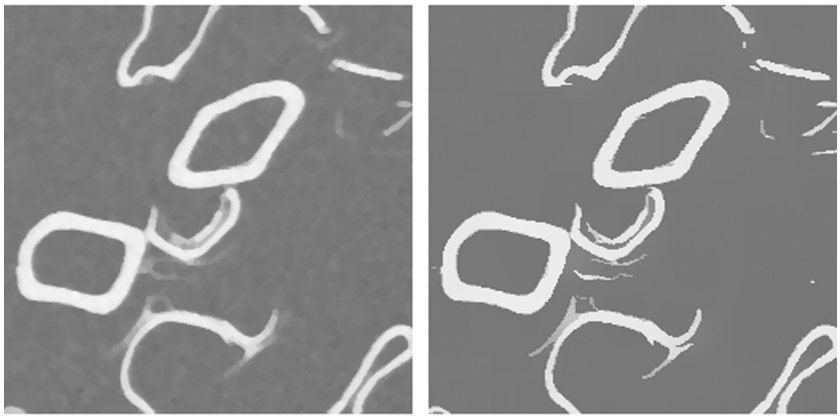


Figure 4.3: a) 50 passes of median filter. b) 50 passes of SUSAN filter.

5 Future work

5.1 GPU based image processing

The field of GPGPU is sure to grow the coming years. The next generation GPUs are faster and better suited for GPGPU work, with support for integer and boolean mathematics (the current generation only supports floating point) and even more pipelines for even more data throughput. Hopefully the high level shading language compilers will mature as well, and debugging tools will improve.

Because we have used GLSL as our language of choice for shader programs, all the shaders we have implemented should perform better as the hardware gets better. There are of course many other image processing operations that could be implemented. With new hardware the entire input image should fit in one texture instead of having to be split up into smaller sub-images. This would enable us to do the whole process of segmentation and labeling on the GPU. Other segmentation schemes could also be explored, including water-shed and similar approaches.

With new hardware the penalty of downloading the image back to the CPU would also be much smaller. We have used an AGP system to be compatible with commodity hardware, but PCI-E is already the norm of new computers. When the AGP bottleneck is removed, the CPU can be used for tasks that are better suited for the CPU than the GPU.

While we have been working at PFI several libraries aimed at making GPGPU easier have emerged, including the OpenVIDIA project ([Fung and Mann, 2005]) and SINTEF's Shallows library[1]. Future projects should use existing libraries to avoid having to design their own framework like we have.

5.2 Visualization and editing

Since most of our time was spent doing the GPU based image processing filters, there is still some work to be done in the visualization and editing part of the program for it to be really useful. Implementing the basic editing operations should not be too difficult using Qt's existing painting capabilities, which are designed to work with OpenGL.

5.3 Intelligent post-processing

Much work on the post-processing part of the program can be done to recognise objects that can be discarded (not interesting fibres) or objects that need editing before they are quantified. In our work we have made the foundation that's necessary to build a more intelligent system on top. By using artificial intelligence methods like neural nets the program could do automatically

[1] Shallows is available at `http://shallows.sourceforge.net/`

much of the manual work that is done today.

6 Conclusions

When doing image processing on paper fibre cross-section images, we found the combination of median and SUSAN filters to be the best for smoothing the images to remove noise while preserving the fibres. We also found our thresholding approximation to give good results.

Using the GPU for image processing enables us to use more computing intensive tasks while still keeping the program highly interactive. Instead of running a batch process to do all the image processing we can let the user decide parameters and see how they affect an image in runtime. This makes image processing more accurate and adaptable for different input images.

With the current generation of graphics processing units a GPU solution performs equally well as its CPU counterpart due to the overhead of splitting images to fit on smaller textures and moving data between the GPU and CPU. While not gaining much performance wise, a GPU implementation would give some benefits to the user interactivity.

Since this field is still young, with immature compilers and debugging tools and few solid libraries, it's harder to implement and debug algorithms on the GPU. This makes development of GPU based image processing programs more time-consuming than the normal approach.

We expect the field to mature greatly in the coming years, making development easier and faster. Already there are some debugging tools that analyse shader programs and report performance profiles, but they are mostly aimed at the game development market and Microsoft's DirectX.

Bibliography

[Bergquist et al., 2004] Bergquist, J., Leknes, J., Skeide, C., Stangvik, E., and Titlestad, H. (2004). Utvikling av rutiner for tverrsnittanalyse av papirfibre. Bachelor's Thesis, HiST.

[Bergquist and Titlestad, 2005] Bergquist, J. and Titlestad, H. (2005). GPU based image processing for analysis of paper fibres.

[Buck, 2005] Buck, I. (2005). Taking the plunge into GPU computing. In Pharr, M., editor, *GPU Gems 2*, chapter 32, pages 509–519. Addison Wesley.

[Davies, 2005] Davies, E. R. (2005). *Machine Vision: Theory, Algorithms and Practicalities*. Number 0-12-206093-8. Morgan Kauffmann.

[Dokken, 2006] Dokken, T. (2006). The gpgpu-project, status and results. why use the gpu? Presented at the "GPU as a computational resource" Workshop.

[Fung and Mann, 2005] Fung, J. and Mann, S. (2005). Openvidia: parallel gpu computer vision. In *MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia*, pages 849–852, New York, NY, USA. ACM Press.

[Gonzalez and Woods, 2002] Gonzalez, R. C. and Woods, R. E. (2002). *Digital Image Processing*. Number 0-201-18075-8. Prentice Hall.

[Hagen and Holen, 2004] Hagen, M. and Holen, R. (2004). Segmentation of absorption mode x-ray tomographic images of paper. Master's thesis, Norwegian University of Science and Technology (NTNU).

[Henden and Bache-Wiig, 2005] Henden, P. C. and Bache-Wiig, J. (2005). Individual fiber segmentation of three-dimensional microtomograms of paper and fiber-reinforced composite materials. Master's thesis, Norwegian University of Science and Technology (NTNU).

[Reme, 2000] Reme, P. A. (2000). *Some effects of wood characteristics and the pulping process on mechanical pulp fibres*. PhD thesis, Norges teknisk-naturvitenskaplige universitet.

[Rost, 2004] Rost, R. J. (2004). *OpenGL Shading Language*. Number 0-321-19789-5. Addison Wesley.

[Sekulic, 2004] Sekulic, D. (2004). Efficient occlusion culling. In Fernando, R., editor, *GPU Gems*, chapter 29, pages 487–503. Addison Wesley.

[Shreiner et al., 2004] Shreiner, D., Neider, J., Woo, M., and Davis, T. (2004). *OpenGL Programming Guide*. Number 0-321-17348-1. Addison Wesley.

[Smith and Brady, 1995] Smith, S. M. and Brady, J. M. (1995). SUSAN – A new approach to low level image processing. Technical Report TR95SMS1c, Chertsey, Surrey, UK.

[Ueland and Botnen, 2005] Ueland, H. and Botnen, M. (2005). Gpu segmentation. Master's thesis, NTNU.

[Viola et al., 2003] Viola, I., Kanitsar, A., and Gröller, M. E. (2003). Hardware-based nonlinear filtering and segmentation using high-level shading languages. In G. Turk, J. van Wijk, K. M., editor, *Proceedings of IEEE Visualization 2003*, pages 309–316. IEEE.

A Shaders

A.1 Median

Listing A.1: Median smoothing filter fragment shader

```
uniform sampler2D Input;

/* Length is actual number of elements,
 */

const int MaskLength = 5;

uniform ivec4 Mask[MaskLength];

/* Import from library: */
vec2 toTexCoord(ivec2 offset);

/* StupidSort[tm] for finding median */
float median(float values[MaskLength])
{
    for (int k = 0; k < MaskLength; k++)
    {
        for (int l = 0; l < (MaskLength-1 ); l++)
        {
            /* We don't have to sort more than half the array to find the median.*/
            if ((l < MaskLength - k) && (k < (MaskLength/2)+1))
            {
                if (values[l] > values[l+1])
                {
                    float tmp = values[l];
                    values[l] = values[l+1];
                    values[l+1] = tmp;
                }
            }
        }
    }

    return values[MaskLength/2];
}

void main()
{
    vec4 pix = texture2D(Input, gl_TexCoord[0].st);

    float values[MaskLength];

```

```
42      for(int i = 0; i < MaskLength; i+=1)
43      {
44          values[i] = texture2D(Input, gl_TexCoord[0].st
45                                + toTexCoord(Mask[i].st)).r;
46      }
47
48      pix.r = median(values);
49
50      gl_FragColor = pix;
51  }
```

A.2 SUSAN

Listing A.2: SUSAN smoothing filter fragment shader

```
/* SUSAN (Smallest Univalue Segment Assimilating Nucleus) smoothing filter.
 *
 * Adapted from C code by Stephen Smith, available at:
 * http://www.fmrib.ox.ac.uk/~steve/susan/
 */

uniform sampler2D Input;
uniform float Threshold;   /* Brightness threshold (0-256) */

const int Radius = 4;       /* Radius of susanMask */
const int MaskLength = 37; /* Number of elements in SusanFilter.susanMask */

uniform ivec4 Mask[MaskLength];

/* Import from library: */
vec2 toTexCoord(ivec2 offset);

/* StupidSort[tm] for finding median */
const int MedianMaskLength=5;
uniform ivec4 MedianMask[MedianMaskLength];

float median(float values[MedianMaskLength])
{
    for (int k = 0; k < MedianMaskLength; k++)
    {
        for (int l = 0; l < (MedianMaskLength-1 ); l++)
        {
            /* We don't have to sort more than half the array to find the median.*/
            if ((l < MedianMaskLength - k) && (k < (MedianMaskLength/2)+1))
            {
                if (values[l] > values[l+1])
                {
                    float tmp = values[l];
                    values[l] = values[l+1];
                    values[l+1] = tmp;
                }
            }
        }
    }

    return values[MedianMaskLength / 2];
}

/* Thresholds an element in the mask by looking at the difference between
 * the element's value and the center pixel.
 */
float brightnessThreshold(float difference)
{
    float temp = difference / Threshold;
    temp = temp*temp;
    temp = 100.0*exp(-temp);
    return temp;
}

/* Each element in the mask is weighted by how far it is from the center. */
```

```
56  float distanceFactor(int x, int y)
57  {
58      return (100.0 * exp( float( ((x*x)+(y*y)) ) / float( (-Radius*Radius)) ));
59  }
60
61  /* The 256.0 and 100.0 factors here are mostly useless, but when we tried to
62   * remove them the NVIDIA compiler ran out of temporary registers.
63   * I suggest a small cleanup of factors when the GLSL compilers are a bit
64   * more mature.
65   */
66  void main()
67  {
68      vec4 pix = texture2D(Input, gl_TexCoord[0].st);
69      float center = pix.r;
70
71      float area = 0.0;
72      float total = 0.0;
73
74      for(int i = 0; i < MaskLength; i+=1)
75      {
76          float brightness = texture2D(Input, gl_TexCoord[0].st
77                                       + toTexCoord(Mask[i].st)).r * 256.0;
78          float tmp = distanceFactor(Mask[i].s, Mask[i].t)
79                                      * brightnessThreshold((center * 256.0)
80                                      - brightness);
81          area += tmp;
82          total += tmp * brightness;
83      }
84
85      int tmp = int(area) - 10000;
86
87      /* To avoid a divide by zero we just use the median of surrounding pixels
88       * if tmp equals 0.
89       */
90      if(tmp == 0)
91      {
92          float values[MedianMaskLength];
93          for(int i = 0; i < MedianMaskLength; i+=1)
94          {
95              values[i] = texture2D(Input, gl_TexCoord[0].st
96                                    + toTexCoord(MedianMask[i].st)).r;
97          }
98          center = median(values);
99      }
100     else
101     {
102         center = ((total - (center*256.0*10000.0)) / float(tmp)) / 256.0;
103     }
104
105     pix.r = center;
106     gl_FragColor = pix;
107 }
```

A.3 Thresholding

Listing A.3: Thresholding fragment shader

```
uniform sampler2D Input;
uniform float Threshold;
void main()
{
    /* Read from the r-channel and write to g-channel */

    vec4 value = texture2D(Input, gl_TexCoord[0].st);

    value.g = step(Threshold, value.r);

    gl_FragColor = value;
}
```

A.4 Labeling

Listing A.4: The label filter's initializing pass fragment shader

```
/* Initialize the labeling filter: Set the pixels to point to their own
texture coordinates in the B and A channels. */
uniform sampler2D Input;
void main()
{
    vec4 pix = texture2D(Input, gl_TexCoord[0].st);

    if(pix.g > 0.5)
    {
        pix.ba = gl_TexCoord[0].st;
    }
    else
    {
        pix.ba = vec2(1.0, 1.0);
    }
    gl_FragColor = pix;
}
```

Listing A.5: The label filter's main pass fragment shader

```
uniform sampler2D Input;
const int MaskLength = 5;

uniform ivec4 Mask[MaskLength];

/* Import from library: */
vec2 toTexCoord(ivec2 offset);

vec2 minmin(vec2 one, vec2 two)
{
    if (one.x < two.x)
    {
        return one;
    }
    else if (one.x > two.x)
    {
        return two;
    }
    else if (one.y <= two.y)
    {
        return one;
    }
    else
    {
        return two;
    }
}

void main()
{
    vec4 pix = texture2D(Input, gl_TexCoord[0].st);

    if(pix.g < 0.5)
    {
        gl_FragColor = vec4(pix.r, pix.g, 1.0, 1.0);
```

```
36      }
37      else
38      {
39          vec2 pos = pix.ba;
40
41          for(int i = 0; i < MaskLength; i+=1)
42          {
43              vec4 pix2 = texture2D(Input, gl_TexCoord[0].st + toTexCoord(Mask[i].st)
44              if (pix2.g > 0.5) {
45                  pos = minmin(pos, texture2D(Input, pix2.ba).ba);
46              }
47          }
48
49          /* Discard the fragment if what it's pointing to is not updated. */
50          if (pix.ba == pos)
51          {
52              discard;
53          }
54          pix.ba = pos;
55      }
56      gl_FragColor = pix;
57  }
```