Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

TDT4590 Complex Computer Systems — Specialization Project

LINE DETECTION ON GPUS USING CUDA

by

Åsmund Eldhuset

Supervisor: Dr. Anne C. Elster
In cooperation with Schlumberger

Kolbotn, January 6, 2009

Abstract

Line detection is a highly useful tool for analysing seismic images. Algorithms that were created for analysis of two-dimensional images can be used on three-dimensional seismic data sets by slicing the data set, but this requires significantly more computation than analysis of a single image. Therefore, there is a need to parallelise such algorithms in order to improve performance. NVIDIA's platform for parallel computation, CUDA, is an attractive choice due to being fairly simple to program and its ability to leverage cheap, small-scale supercomputers in the form of graphics processing units (GPUs).

This project investigates a line detection algorithm developed by Carsten Steger. We port an implementation for single core CPU to run on a high-end NVIDIA GPU, using the CUDA programming environment. For comparison, we also port the algorithm to OpenMP. The results that are obtained for the two most time-consuming steps of the five-step algorithm are promising, but the design of some of the other steps poses challenges for the porting to CUDA.

Acknowledgements

I would like to thank the following persons and companies:

- My supervisor, Dr. Anne C. Elster, for getting me in touch with Schlumberger, for coping with my work habits and for taking time off her vacation to provide valuable feedback in the final stages of the project.
- Schlumberger and two of the employees at their Trondheim office, Rune Fevang and Wolfgang Hochweller, for providing me with Steger's code and seismological data for testing.
- NVIDIA, who has donated GPUs for the HPC lab of our department through Elster's membership in their Professor Partnership Program.

Contents

List of Figures

List of Tables

List of Symbols and Abbreviations

Abbreviation	Description	Definition
CUDA	Compute Unified Device Architecture	page 10
FFT	Fast Fourier Transform	page 5
GPU	Graphical Processing Unit	page 2
SIMD	Single Instruction Multiple Data	page 11
SIMT	Single Instruction Multiple Thread	page 11
SM	Streaming Multiprocessor	page 14
SP	Scalar Processor	page 14

Chapter 1

Introduction

An essential tool for analysing geological formations in the search for oil is *reflection seismology*, in which waves caused by explosions other kinds of shocks are sent through the ground[14]. The time it takes for the waves to be reflected back is used to estimate the nature of the underlying geology. Line detection on such images can provide valuable information about *faults*, which are rifts in rock layers. Some faults permit oil and gas to flow through them, while others block oil and/or gas.

Schlumberger is a worldwide oil field service provider with which our department has cooperated on several student projects. They intend to use line detection to discover and link faults, since reservoir simulations require information about the fault structures. In addition, they hope that line detection may aid the detection of salt formations, which lack a clearly defined structure in the three-dimensional seismic results.

Although seismic measurements usually produce three-dimensional data sets, algorithms for two-dimensional line detection may still be used by slicing a 3D data set (describing a volume) into many 2D data sets (describing surfaces, or cross-cuts through the volume). This additional dimension causes a large increase in computational requirements. Ordinary images typically contain a few millions of pixels, while 3D data sets may contain billions of points. Thus, algorithms that are adequate for single image analysis may not suffice for seismic analysis.

One such algorithm was developed by Carsten Steger and is described in his report entitled "An Unbiased Detector of Curvilinear Structures"[12]. Among other advantages, the algorithm handles noisy images well, and it corrects the bias in line position that is often introduced by other algorithms when the two sides of a line have different contrast. He did not give the algorithm a name, so for the purposes of this project, we have named it *Steger's algorithm*. The use of this algorithm was suggested by Schlumberger, who hope that this project will provide useful information to them if they want to parallelise their own, proprietary edge detection algorithms.

NVIDIA is a world leadning developer and manufacturer of graphical processing units (GPUs). Their most recent GPU architecture is called Tesla, for which they have created an application development platform called CUDA. CUDA allows developers to write programs in C with a small set of extensions. It has gained popularity due to being a fairly simple and cheap way to develop parallel applications.

The purpose of this project is to investigate how Steger's implementation of his line detection algorithm might be ported to CUDA in order to leverage the parallelisation opportunities offered by CUDA. We will also port the implementation to OpenMP (an API for writing multithreaded programs in C) and compare the performance gains to that of CUDA.

Outline

In Chapter 2, we give a brief walkthrough of the mathematics that is required to understand Steger's algorithm, before presenting the algorithm itself. We also explain the CUDA programming model and the hardware organisation of the Tesla architecture.

In Chapter 3, we describe our initial analysis of Steger's implementation, how we ported four of the five algorithm steps, and why we did not succeed in porting one of the steps. We also describe our timing approach.

In Chapter 4, we provide a timing analysis of the three implementations (Steger's original one, and our CUDA and OpenMP ports). We compare the results, discuss the advantages and disadvantages of the two parallelisation approaches, and discuss how well Steger's algorithm is suited for parallelisation.

Finally, in Chapter 5, we present our conclusions and suggestions for future work.

Chapter 2

Background

In this chapter, we provide a quick review of the mathematics that are required to understand the line detection algorithm, and then present the algorithm itself. We then present the CUDA programming model and the Tesla hardware architecture. We give a short introduction to OpenMP, and give a brief description of Amdahl's law, an important formula related to the performance of parallel applications.

2.1 Mathematical background

2.1.1 Convolution

Convolution is a mathematical operation that takes as input two functions and produces a new function which can be interpreted as telling to what extent the original functions "match" if their graphs are aligned with each other. It is defined as

$$(f * g)(t) = \int_{-\infty}^{\infty} f(a)g(t-a)\, da. \tag{2.1}$$

Convolution in two dimensions is defined similarly:

$$(f * g)(s,t) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(a,b)g(s-a,t-b)\, da\, db. \tag{2.2}$$

Differentiation A highly useful property of convolution is that differentiation of one of the functions corresponds to differentiation of the result[5, p. 188]:

$$\frac{d}{dx}(f * g) = \frac{df}{dx} * g = f * \frac{dg}{dx} \tag{2.3}$$

Discrete convolution If both functions to be convolved are discrete, the integrals can be replaced by summations, and if they are finite in extent, the limits can be narrowed. If the range of f is $[-n, n] \times [-m, m]$, then we have:

$$(f * g)(s,t) = \sum_{a=-n}^{n} \sum_{b=-m}^{m} f(a,b)g(s-a, t-b). \tag{2.4}$$

Typically, we are interested in applying this only to values of s, t that are within a certain "interesting" range of g (g might represent an image). Then we must choose what happens if $s - a$ and $t - b$ end up outside of this range. Typical choices is to define g to be zero outside of that range, or that g is periodic, or that values outside of the range mirror the values inside the range (so that $g(s,t) = g(s,-t)$ and $g(s,t) = g(-s,t)$).

Kernels In this context, f is often referred to as a *convolution mask* or *convolution kernel*. When f has a small range and g represents an image, convolution is usually interpreted as applying some effect defined by f (e.g., blurring or sharpening) to each point (i.e., pixel) of g, where the effect on each point depends on the value of the point itself and of the neighbouring points.

Integrated kernels Even if the kernel f is continuous, we can perform discrete convolution by creating a discrete version f_d of f like this:

$$f_d(s,t) = \int_{s-1/2}^{s+1/2} \int_{t-1/2}^{t+1/2} f(a,b)\, da\, db, \tag{2.5}$$

and calculating $f_d * g$ with the discrete formula. This is called an *integrated kernel*.

Separability Discrete two-dimensional functions with a rectangular range (such as images and convolution kernels) can be expressed as matrices. If it is possible to express such a function as a product of a column vector and a row vector, the function is called *separable*. If a convolution kernel f is separable, so that $f = c \times r$, convolution can be performed by first convolving each row of g with r, and then convolving each of columns of the results with c. This is simpler and more efficient than using 2.4 directly (the time complexity is $\Theta(wh(w+h))$ as opposed to $\Theta(w^2h^2)$, but it should be noted that not all kernels are separable.

2.1.2 Fourier transform

The Fourier transform takes as input one function (normally interpreted as a function from time to wave amplitude, and thus often referred to as a *signal*) and produces a function that describes what frequencies are present in the original signal. The Fourier transform does not appear in Steger's paper, but it is often used to calculate discrete convolutions. This is due to the *convolution theorem*, which states that the Fourier transform of the convolution of two functions is equal to the product of the transforms of the original functions:

$$\mathcal{F}\left\{f * g\right\} = k\mathcal{F}\left\{f\right\} \cdot \mathcal{F}\left\{g\right\} \tag{2.6}$$

where k is a constant factor that depends on how the Fourier transform is defined (different scientific branches use different scaling factors for convenience in their Fourier transform definitions). For a proof, see [6]. Taking the inverse transform on both sides yields

$$f * g = \mathcal{F}^{-1}\left\{k\mathcal{F}\left\{f\right\} \cdot \mathcal{F}\left\{g\right\}\right\}, \tag{2.7}$$

which provides us with an even more efficient way to calculate convolutions, due to the existence of the Fast Fourier Transform algorithm (FFT). The FFT requires $\Theta(n \lg n)$ time for a discrete function of n elements, as opposed to directly evaluating the summations that define discrete convolution, which requires $\Theta(n^2)$ operations.

In higher dimensions, we can utilise the fact that the Fourier transform is separable, and perform a series of one-dimensional transforms. For two-dimensional images of width w and height h, this means that the direct calculation requires $\Theta(w^2h^2)$ operations (or $\Theta(wh(w+h))$ if the mask is separable), while the FFT approach only requires $\Theta(wh \lg w \lg h)$ operations independently of whether or not the mask is separable.

2.1.3 Gaussian kernels

A commonly used convolution kernel in image analysis, due to its ability to "smooth out" the function (or image) it is being convolved with, is the *Gaussian kernel*, which we present here together with its first and second derivatives, which will be used later on:

$$g_\sigma(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \tag{2.8}$$

$$g'_\sigma(x) = \frac{-x}{\sqrt{2\pi}\sigma^3} e^{-\frac{x^2}{2\sigma^2}} \tag{2.9}$$

$$g''_\sigma(x) = \frac{x^2 - \sigma^2}{\sqrt{2\pi}\sigma^5} e^{-\frac{x^2}{2\sigma^2}} \tag{2.10}$$

The two-dimensional Gaussian kernel is defined as follows, and it has the desirable property of being separable:

$$g_\sigma(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} = g_\sigma(x)g_\sigma(y) \tag{2.11}$$

The results of convolving the line functions with the Gaussian kernels and its derivatives involve the integral of the kernel:

$$\phi_\sigma(x) = \int_{-\infty}^{x} e^{-\frac{t^2}{2\sigma^2}}\, dt \tag{2.12}$$

Since this integral cannot be analytically solved, it is normally convenient to precalculate some of its values (using numerical integration), store them in a lookup table and use interpolation when a function value is needed.

2.2 Line detection. Steger's algorithm

In this section, we outline the algorithm presented in [12]. The goal of this algorithm is to analyse two-dimensional grey value images and detect the locations and widths of the lines in the image. A *scale-space*[1] parameter σ is used to determine the range of widths one is interested in.

2.2.1 Mathematical models of lines

[12] begins with discussions of models of lines in one dimension — in essence, what you would see if you took a two-dimensional surface containing ridges and made a cross cut perpendicular to one of the ridges. Three types of lines are discussed ($2w$ and h are the width and height of the lines, respectively): *Parabolic lines*, described by $h(1-(x/w)^2)$ for $|x| \leq w$; *bar-shaped lines*, having a constant height h for $|x| \leq w$; and finally *asymmetric bar-shaped lines*, whose surroundings have different heights:

$$f_a(x,w,a) = \begin{cases} 0, & x < -w \\ 1, & |x| \leq w \\ a, & x > w \end{cases}, \tag{2.13}$$

where a is the height of the surroundings on the right-hand side. Arbitrary line heights h could have been considered, but h will drop out of all calculations of interest[12, pp. 6–7]. This kind of profile is regarded as the most likely one in real images.

Parabolic lines can obviously be detected by looking for points where the first derivative is zero. However, since an image typically will contain noise, looking at differences between neighbouring pixels will not yield much useful information for estimating the derivatives. According to [13], the solution to the noise problem is to perform convolution with a Gaussian kernel, which in essence will smooth out the image. The σ parameter defines the extent of the

[1]Scale-space is a strategy for representing an image as a collection of images at different levels of detail — see [13]

smoothing — the higher the value is, the less detail will be left in the image, and only thicker lines will remain. Thus, we can use σ to place a lower threshold on how thick lines we want to detect.

If we apply the Gaussian kernel to a bar-shaped line (be it asymmetrical or not), the profile becomes curved so that its derivatives are nonzero in most places — then, we can use the zero point of the first derivative to detect the centre of the curve and the zero point of the second derivative to detect the edges. The *responses*, (that is, the convolution results) we get for a line of width $2w$ and surrounding height a when we use the scale parameter σ, are:

$$\begin{aligned}
r(x,\sigma,w,a) &= g_\sigma(x) * f_a(x,w,a) = \phi_\sigma(x+w) + (a-1)\phi_\sigma(x-w) \\
r'(x,\sigma,w,a) &= g'_\sigma(x) * f_a(x,w,a) = g_\sigma(x+w) + (a-1)g_\sigma(x-w) \\
r''(x,\sigma,w,a) &= g''_\sigma(x) * f_a(x,w,a) = g'_\sigma(x+w) + (a-1)g'_\sigma(x-w)
\end{aligned}$$

Since the line position l is the point where $r'(l,\sigma,w,a) = 0$, we get:

$$l = -\frac{\sigma^2}{2w}\ln(1-a) \tag{2.14}$$

which shows that the line will be detected in the wrong position if the line is asymmetric. We cannot directly compensate for this since we do not know the true values of w and a, but step 5 of the algorithm provides a way of correcting the bias.

The magnitude of r'' can be used to determine whether a line is *salient* (that is, strong enough). If this approach is to make sense, r'' should have a minimum at the line location. However, it turns out that the line must not be too wide for this to be the case:

$$\sigma \geq \frac{w}{\sqrt{3}} \tag{2.15}$$

This means that σ not only imposes a lower limit on the line widths that are detected (due to the smoothing of the Gaussian kernel), but also an upper limit.

2.2.2 Step 1 — Convolution

The code related to this step is located in `convol.cu` (original) and `convol.cu` (new).

Given the line model just discussed, there is a need to calculate the convolution of the image with the Gaussian kernel in order to smooth the image, and the later steps require the first and second derivatives of the convolution[2]. Due to Equation 2.3, we see that instead of calculating those derivatives directly,

[2]The result of the first convolution will actually turn out not to be needed — it is shown here for completeness, but the application never computes it.

we can perform convolutions with the derivatives of the Gaussian kernel instead[3]. When performing discrete convolution, one must choose what to do at the boundaries of the image; since our goal is to detect lines, the image is mirrored off the boundaries (rather than letting the image be zero outside of its actual boundary) so that we avoid huge derivatives at the boundary.

2.2.3 Step 2 — Ridge detection

The code related to this step is located in `position.c` (original) and `position.cu` (new).

The idea behind this step is most easily illustrated in one dimension. Rather than simply checking where the values of the derivative is zero or goes from positive to negative between neighboring pixels, they use a second-order Taylor polynomial to estimate the value of the "image function". If r, r' and r'' are the functions that are obtained when convolving with the Gauss kernel and its derivatives, the Taylor polynomial is $p(x) = r + r'x + \frac{1}{2}r''x^2$. Since we are interested in the point where $p'(x) = 0$, we get $p'(x) = r' + r''x = 0$, and thus $p(x) = -\frac{r'}{r''}$. Conveniently, r disappears, so step 1 does not have to calculate it. The advantage of this approach is that we can get a subpixel location for each line point.

In two dimensions, the same idea can be applied, but now we have a problem in that we do not know the direction of the line in each pixel. We need this information because the derivative can only be required to be zero in the direction perpendicular to the line (in case the height of the line changes along the direction of the line). It turns out that the direction perpendicular to the line can be determined by finding the eigenvectors and eigenvalues of the so-called *Hessian matrix*, which contains all of the second derivatives of r with respect to the image coordinates x and y. Given the longest eigenvector $\vec{n} = (n_x, n_y)$, the point where the directional derivative along n is zero is given by $\vec{p} = (p_x, p_y) = t\vec{n} = (tn_x, tn_y)$, where

$$t = -\frac{r_x n_x + r_y n_y}{r_{xx} n_x^2 + 2r_{xy} n_x n_y + r_{yy} n_y^2}. \tag{2.16}$$

For each pixel where both p_x and p_y fall inside the range $\left[-\frac{1}{2}, \frac{1}{2}\right]$, the following is done:

- $\vec{n}$ is stored and associated with the pixel.
- The "global coordinates" of the point where the derivative along n was determined to be zero, namely $(c + tn_x, r + tn_y)$, where r and c are the

[3]This is obviously more computationally expensive, but I assume that it gives higher accuracy, since we can get precise values for the derivatives of the kernel, while the derivatives of the convolution would have to be computed by comparing neighbouring pixels.

row and column numbers, respectively, are stored and associated with the pixel.

- If the eigenvalue associated with the eigenvector $\vec{n}$ is higher than a certain lower threshold (called *hysterisis threshold*), the pixel will be marked as being a line point. If the eigenvalue is also higher than another given hysterisis threshold, the pixel is also marked as a possible starting point for a line.

2.2.4 Step 3 — Line point joining

The code related to this step is located in `link.c`.

The possible starting points from step 2 are examined from top left to bottom right. If we assume that lines will not make "too sharp" turns, it is sufficient to look at three of the eight neighbouring pixels (the choice of neighbours depends on the angle of the normal). For instance, if the normal indicates that the line itself has an angle between $-22.5°$ and $22.5°$, we can consider the three pixels to the (upper, middle and lower) right. For each of the three neighbour pixels that contain a line point, the distance d and angle difference β between the current point and the neighbour point are measured, and the point that minimises $d + c\beta$ is chosen ($c = 1$ is used in this implementation) as the next point of the line. This is repeated so that successive points of the line are detected, until the end of the line or a point that is already part of another line is reached. In the latter case, that point is marked as a junction, and the line that was encountered is split into two lines (so that in total, there are three lines that end in the junction).

Each line has a normal, but it might point to either side of the line. While traversing the line points, the normals are oriented so that they always point to the right (relative to the traversal direction). This is to ensure that the later steps will get a consistent idea of the left and right sides of a line.

While the high-level description of this step is fairly simple, the implementation is rather long and tricky, as it contains a lot of trigonometry, searching, sorting, and array reallocations. This is expected to cause problems for the GPU implementation.

2.2.5 Step 4 — Line width determination

The code related to this step is located in `width.c`, primarily in the function `compute_line_width`.

In order to determine the line width, we must look for edges on both sides of each line point, in the directions perpendicular to the line. Due to Equation 2.15, it is sufficient to go a distance of $\sigma\sqrt{3}$ on each side, but the existing implementation uses a slightly larger value to be on the safe side. The Bresenham line drawing algorithm[11] is used in a slightly modified form to determine which pixels should be examined. We are looking for the point where the absolute value of the gradient (a vector whose elements are the first derivatives in the x and the y directions) has the greatest value. An approach similar to the one in step 2 (finding eigenvectors of a Hessian matrix) is used to determine, for each pixel, the subpixel location of the maximum gradient. This corresponds to locating the zero point of r''.

Missing edge points For some line points, edge points (and thereby line width) might not be detected. This will happen if the edge of the line is wide and the grey values change slowly, and in junctions, where the line might grow too wide. For such points, the width will be linearly interpolated from the closest surrounding line points that have a width. If the point at the start (or end) of a line does not have a width, the width of that point will be set to the width of the first (or last) point that has a width.

2.2.6 Step 5 — Bias removal

The code related to this step is located in `width.c`, primarily in the function `fix_locations`. The bias function to be discussed is implemented in `correct.c`.

In step 4, we detected the points e_l and e_r where $r''(e_l, \sigma, w, a) = r''(e_r, \sigma, w, a) = 0$. These expressions can be interpreted as a function from w and a to e_l and e_r (given a fixed σ — in this step, we can arbitrarily set $\sigma = 1$ because it will drop out from the results we are interested in). We would like to invert this function in order to determine w and a (the true line width and asymmetry) from e_l and e_r (the observed edge locations), but we need more information. It turns out that the ratio of the gradient at the detected edges, $|r'(e_l, \sigma, w, a)|/|r'(e_r, \sigma, w, a)|$, which can be observed in the image, provides the extra information about the relationship between these variables. By using a root finding algorithm (since the equations cannot be solved analytically), it is possible to create an inverse function that will tell us w and a given the edge locations and the gradient ratio. Selected values of this function are precomputed, and interpolation is used to obtain other values during the bias removal step.

2.3 CUDA

CUDA, short for Compute Unified Device Architecture, is a programming model and software environment for developing applications for GPUs. NVIDIA, the corporation that developed CUDA, has also created the Tesla architecture, which extends their traditional GPU architecture to provide a hardware environment for running CUDA applications. Most of the information in this section is based on NVIDIA's programming guide[9].

2.3.1 Using GPUs for calculation

Traditionally, GPUs have been built around a fixed pipeline[3]:

Input Vertices (grouped into triangles) describing the geometry of the scene, along with textures to be applied to the geometry, are input to the GPU.

Vertex processing The scene coordinates of the vertices are transformed to screen coordinates.

Rasterisation For each triangle, the screen coordinates that are covered by the triangle are determined.

Fragment processing Each such screen coordinate is referred to as a *fragment*. If the triangle was textured, an appropriate colour for the fragment will be determined from the texture.

Framebuffer processing Fragments that map to the same pixel are being combined in specified ways. For instance, an opaque fragment will hide a fragment that is behind it, while colours for transparent fragments must be blended together.

Originally, the operations in each step were hardwired in the GPU, but in the last decade, the vertex and fragment stages have been programmable. Such programs are referred to as *vertex shaders* and *pixel shaders*. Although the original intent was to allow for customisable graphics effects, people soon started utilising shaders for general-purpose computation. This, however, is quite hard, since it requires that the problem domain be represented as a texture and that the calculations be represented as vertex and pixel shader operations. The most recent GPUs, both from NVIDIA and AMD, take a *unified processor* approach in which the hardware consists of a number of (fairly) general-purpose processing cores, and the cores are dynamically assigned to different stages of the pipeline. This also means that it is much easier to write general-purpose computations for GPUs. See [10, 3] for more information.

2.3.2 Programming model

NVIDIA call their approach to parallelism SIMT — Single Instruction Multiple Thread. The idea is that each instruction is being executed by several threads in parallel. The difference from a standard SIMD (Single Instruction Multiple Data) architecture is that the programmer is not required to be aware of the width of the parallelism (when programming vector machines or e.g. using SSE instructions, one must know how many elements will be processed in parallel). Also, the number of threads can be varied, as opposed to many SIMD architectures, where the number of parallel elements is fixed. Since parallelism is implemented with threads, it is also possible for the threads to take different branches through the code. The threads allow for fine-grained data parallelism, and threads can be grouped into sets called *blocks* and *grids* that can be used to implement coarse-grained data parallelism and task parallelism, respecively.

CUDA programs are divided into *host code*, which will execute on a regular CPU, and *kernel code*, which will execute on a CUDA-enabled device in the same computer (typically a GPU). Both parts are written in C with a few extensions. The most prevalent one is the keyword `__global__`, which among other things can be used to declare a function to be a kernel. The host code and the kernel code can be written in the same source files. The compiler front end moves the host code to normal C files that are compiled separately, and generates GPU assembly code. The latter is assembled into object files that can be transferred to and executed on a CUDA device.

The host program controls the CUDA device and decides when to launch kernels to be executed by multiple threads on the device. Threads that are launched together are referred to as a *grid*, and they all execute the same kernel. Several grids may run simultaneously. The threads in a grid are grouped into *blocks* of equal size. A kernel is launched by specifying its name and parameters together with a *launch configuration*: `kernelName<<<blocks, threadsPerBlock, sharedMemory, stream>>>(parameterList);`. `parameterList` is a regular C function parameter list; the parameters are automatically copied to the device (for pointers, only the pointer itself is copied; it must point to device memory in order to be valid in the kernel). The meaning of the launch configuration parameters are as follows:

blocks The number of blocks in the grid.

threadsPerBlock The number of threads in each block. The total number of threads is thus $blocks \cdot threadsPerBlock$.

sharedMemory (optional) Threads within the same block have access to a *shared memory* (discussed later). This parameter specifies how much shared memory should be available to each block. If omitted, the block gets no shared memory.

stream (optional) *Streams* can be used to control concurrency. A sequence of kernel launches and memory operations on the same stream will run sequentially, while actions on different streams may run concurrently. If omitted, a default stream is used.

Threads within a block may be indexed in one, two, or three dimensions, depending on what maps best to the problem domain (this is specified in the launch configuration). Blocks may be indexed in one or two dimensions. Inside a kernel, the variables `blockDim`, `blockIdx`, and `threadIdx` can be used to obtain the thread's indices (from which a unique thread number from 0 to $(blocks \cdot threadsPerBlock - 1)$ can be easily computed).

Synchronisation Kernel launches are asynchronous, which means that the host code continues to run immediately after the kernel has been launched. This allows (independent) computations to be performed on the GPU and the CPU in parallel. If the host needs to wait for the GPU to complete a task, it can call `cudaThreadSynchronize()`. This is not necessary if `cudaMemcpy()` is used to get results back from the GPU; that function is synchronous and will only begin after the kernel has completed and will not return until the memory transfer is complete. However, it may be beneficial to utilise the asynchronous facilities that are available (the stream concept mentioned above, and asynchronous memory transfer operations) — this allows interleaving of communication and computation, which in some situations can save a considerable amount of time.

Barrier synchronisation of all threads within the same block can be performed by calling `__syncthreads()` from the kernel. It can be called from within a conditional, but then it is the programmer's responsibility to ensure that all threads in the block actually will reach the call. There is no mechanism for directly synchronising blocks with each other, but on newer cards, atomic integer operations are available, which means that global locks can be implemented manually if it is really necessary (but this will be terribly slow due to the low speed of the global memory).

Logical memory model The following memory regions are available to a thread:

Local memory Private to each thread. Disappears when the thread completes.

Shared memory Accessible by all threads in the same block, but not by any other threads. Disappears when the threads in the block complete.

Global memory Accessible by all threads in all blocks in all grids. Data remains in global memory until overwritten or deallocated, and does not disappear when a grid completes.

Texture memory Read-only memory that efficiently supports a number of different addressing modes.

Constant memory Read-only memory that is optimised for the case where many threads access the same value

2.3.3 Tesla architecture

Tesla is NVIDIA's most recent GPU architecture, which supports the CUDA programming model. The core processing element in a Tesla device is called a *Streaming Multiprocessor* (SM). A GPU contains several SMs — for instance, the NVIDIA Tesla C870 card contains 16 of them. Each SM contains the following (see Figure 2.1):

- Eight *Scalar Processors* (SP)
- One instruction decoder
- Two special function units (SFU) for functions such as sine, cosine, square root and exponential
- Eight register files, one per SP
- Shared memory common to all eight SPs

When a grid is launched, each block is mapped onto an SM. Each SM may contain several blocks, and blocks never move between SMs. The SM is responsible for scheduling and executing all threads within its blocks.

Note that the first generation CUDA devices did not support double precision floating point operations at all. More recent devices do support doubles, but they only have one double precision unit per SM, which means that use of double precision will incur a drastic performance hit.

Warps There is only one instruction decoder per SM, since all threads executing on the SPs will ideally execute the same operations. It decodes one instruction per four SP cycles, so threads are grouped into groups of 32 threads called *warps*. After an instruction has been decoded, the first eight threads in the current warp are executed, then the next eight threads, and so on. The first 16 and last 16 threads of a warp are called *half-warps*.

Physical memory model The logical memory model is implemented with the help of the following physical memories (see Figure 2.1):

Registers Similar to the registers of ordinary processors. Data in the local memory is preferrably put here. Each register is assigned to one thread, so the register file on an SM must be divided between all threads in all blocks on that SM. If the threads require more registers than what is available in the SM, *registers may be spilled to global memory*. Register access time is normally zero cycles.

Shared memory There is one physical shared memory per SM; this is divided between all blocks on the SM. Shared memory is never spilled, and threads in one block may not touch the shared memory of other blocks on the same SM. Accessing shared memory is almost as fast as accessing registers, depending on how the threads in the block access it (see *bank conflicts* below).

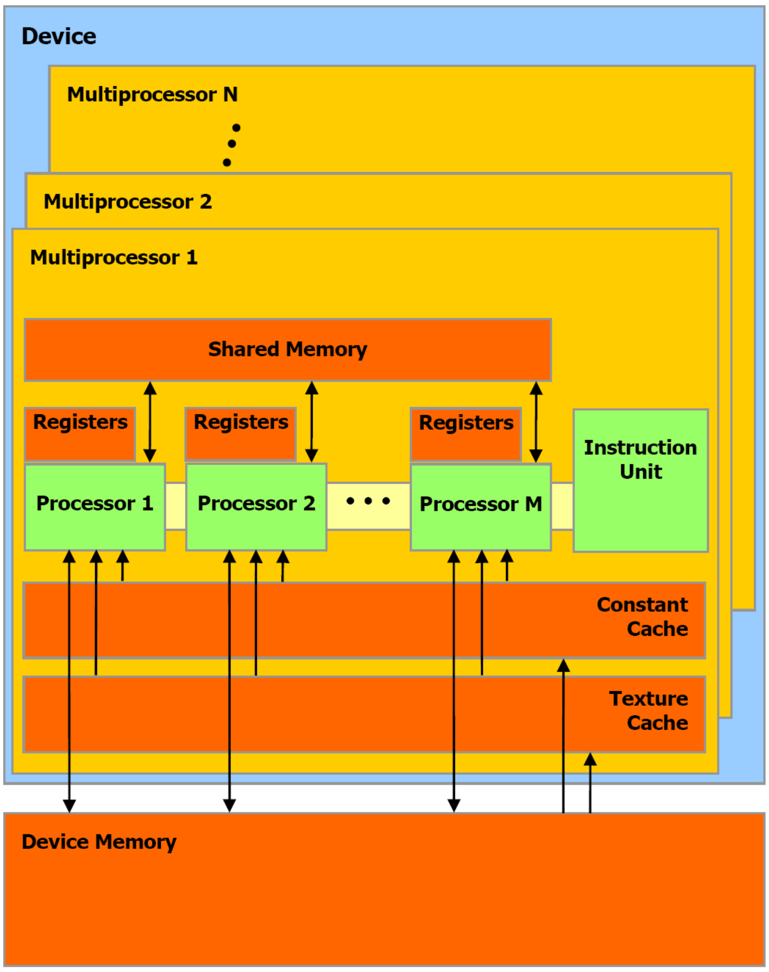


Figure 2.1: The Tesla architecture. Illustration from[9].

Global memory Two orders of magnitude slower than registers[4, Table 1]. Data can be transferred between global memory and the computer's system memory with DMA. Not cached, except for the constant caches mentioned below.

Texture memory Implemented as a global memory cache on each SM.

Constant memory Implemented as a global memory cache on each SM.

The most important thing to realise here is that local memory is an abstraction; there is no physical local memory, and if a kernel has too many local variables, some of them will get spilled to global memory. It should also be noted that concurrent writes to the same location in shared or global memory are resolved by letting one of the threads win (which one is undefined, but one of the writes is guaranteed to succeed).

Banks The shared memory is organised into 16 *banks*, each with a width of 32 bits. Successive 32-bit words belong to successive banks, so a memory address m will map to bank number $\lfloor m/4 \rfloor \mod 16$. All threads in a half-warp may access the shared memory simultaneously if no bank is being accessed by more than one thread. As opposed to global memory, the threads are not required to access adjacent addresses that are aligned to a certain multiple of bytes. If any of the banks is being accessed by two or more threads, a penalty is incurred since the memory accesses must be serialised. Shared memory accesses in the two half-warps that make up a warp will never conflict with each other. Note that there is also a broadcast mechanism that allows the contents of a word in one of the banks to be provided to all threads, so that the scenario where all threads read from the same address is conflict-free.

Divergent warps While older GPUs required all threads to follow the same path, the CUDA programming model allows the use of if/else constructs that will take different branches in different threads and loops where the loop count varies between threads. However, since the SM is structured to make all threads in the same warp execute the same instructions, the following must be done with *divergent warps*, that is, warps where threads take different branches: all chosen execution paths are executed in serial until the paths converge again. Masking is used to make sure that each thread only sees the result of the branches the thread actually chose. Thus, divergent warps may be highly expensive, in particular if all threads follow different parts[4]. Note that threads in different warps do not affect each other. The fairly small warp size allows more liberal use of branches, as opposed to earlier architectures where the SIMD width was greater and branches were more expensive[8].

Coalesced reads and writes Global memory can deliver data at its peak bandwidth only when it is being accessed in a certain manner. Requests to global memory are serviced in *transactions*, whose size can be 32, 64, or 128 bytes. If all threads in a half-warp access data of certain word sizes (see [9, sec. 5.1.2.1] for details) from the same memory segment, and all threads access the segment in sequence, the requests can be serviced in a single transaction (the sequence restriction has been lifted on more recent devices) — this is called *coalesced memory access*. Otherwise, the request must be broken down into several transactions, which is slower.

Occupancy The number of blocks on each SM is limited by the following factors. The numbers in parentheses indicate the limits for the C870.

- Each SM supports a maximum number of threads (768):
  $blocksPerSM \cdot threadsPerBlock \leq threadLimit$

- The shared memory (16384 B) on an SM must be divided between all blocks on the SM (there is no mechanism to swap out shared memory to global memory, and blocks never move between SMs — so two blocks cannot use the same shared memory region): $blocksPerSM \cdot sharedMemory \leq memoryLimit$

- Each SM supports a maximum number of warps (24):
  $blocksPerSM \cdot \lfloor threadsPerBlock/32 \rfloor \leq warpLimit$

If none of these equations allow $blocksPerSM$ to be at least 1, the launch configuration is obviously invalid. However, simply being valid is not necessarily enough — typically, one is interested in maximising the *occupancy*, that is, the amount of available threads, warps, and shared memory that is actually being used (of course, there has to be an integral number of blocks on each SM, and any remaining SM resources that are not used by these blocks are wasted). There is also a limit on the number of threads per block (512 on the C870).

In addition to these hard limits, one will normally also want to consider the limit on registers. The available registers on an SM (8192 on the C870) must be divided between all threads on the SM. Only local variables can be placed in registers, but local variables can also be spilled to global memory, and hence this is not a hard limit. However, for performance reasons, it is obviously desirable not to use more local memory than the registers can hold.

The "CUDA occupancy calculator", provided by NVIDIA and available at `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`, is a handy tool to calculate occupancy given the GPU model, the number of threads per block, numbers of registers per thread (this must be calculated manually by the programmer from the kernel source code) and the amount of shared memory per block.

2.4 OpenMP

OpenMP is an API for writing multithreaded programs in C and a number of other languages. One of its most important features is its support for *declarative programming* — rather than using data structures and function calls to spawn and handle threads, it is possible to use compiler directives to simply state the desired form of parallelisation. The compiler, together with a runtime system that will be linked into the program, will handle the actual thread management.

We will mostly employ the `parallel for` construct, which in its simplest form looks like this:

```
#pragma omp parallel for
for (i = 0; i < limit; ++i) {
  // Loop code here
}
```

This indicates that the loop is suitable for data parallelism. The OpenMP runtime environment will determine a suitable number of threads (based on the hardware of the computer the program is being run on) and distribute the loop iterations between those threads. If desired, the number of threads can be manually set to a value `n` by appending `num_threads(n)` to the **`#pragma`** line.

Variables inside the directive scope can have different kinds of semantics:

- A *shared* variable exists in one location, common to all threads (which must handle synchronisation properly themselves, if needed).
- A variable that is *private* is duplicated — each thread gets its own copy of the (uninitialised) variable.
- If initialization of private variables is needed, the variable can be marked as *firstprivate*, which will cause each thread to get its own copy that is initialised with the value the variable had prior to the directive.

This is controlled by listing the variables in the directive, and preferrably use **`default`**`(none)` to demand that all variables be explicitly mentioned:

```
#pragma omp parallel for default(none) shared(a, b) \
    private(i) firstprivate(limit)
for (i = 0; i < limit; ++i) {
  a[i] = b[i];
}
```

A directive that is useful for task parallelism is `parallel sections`. If several sections of code can be assigned to one thread each, one can do the following:

```
#pragma omp parallel sections
{
#pragma omp section
  {
    // Code for first thread
  }
#pragma omp section
  {
    // Code for second thread
  }
}
```

The directive **`#pragma`** `omp barrier` can be used for barrier synchronization among all threads. Each thread can obtain its own sequence number and the total number of threads by calling `omp_get_thread_num()` and `omp_get_num_threads()`, respectively.

2.5 Amdahl's law

In 1967, Gene Amdahl argued[1] that parallel processing was *not* a good way to improve performance, based on the following observation: If we have a computation that consists of a sequence of serial steps that take a total time of t_s, and a certain percentage f of these steps can be performed in parallel using p processors[4], then the total time for the sequential part of the calculation will be ft_s, and if we can distribute the remaining workload equally over the p processors, the parallel part will take the time $(1-f)t_s/p$. Since the sequential part must presumably be completed before the parallel computations can begin (or the other way around), these times must be added together for a total time of $t_p = ft_s + (1-f)t_s/p$, and we arrive at *Amdahl's law*[5] for the speedup $S(p)$:

$$S(p) = \frac{t_s}{t_p} = \frac{t_s}{ft_s + (1-f)t_s/p} = \frac{1}{f + (1-f)/p} = \frac{p}{1 + f(p-1)} \tag{2.17}$$

The most significant aspect of this formula is that it highlights the importance of f as a severely limiting factor for the potential speedup of parallelising. Amdahl used this to claim that parallel programming was not a good idea. However, since parallelism is currently regarded as the primary way of improving performance in HPC[2], the modern interpretation is that speedups can come arbitrarily close to p if only we can make f small enough, and thus, one should focus on reducing f when parallelising a calculation. On the other hand, since f in most situations cannot (even approximately) become equal to zero[6], Amdahl's law provides an upper bound on performance gain for a specific problem given the best nonzero f we can achieve: As p tends to infinity, the speedup converges to $1/f$. Again, it should be noted that these bounds are optimistic, as they are based on very simplifying assumptions.

[4]These are quite optimistic calculations, since we assume that the calculation can be parallelised without incurring communication penalties or extra computation steps, and that the parallel processors are as fast as the sequential one.

[5]Amdahl did not actually state this formula in his article, but it has been derived later (in many different forms) from his article.

[6]Calculations in which $f \approx 0$ and where the assumptions about independence between the parallel parts hold are called *embarrassingly parallel*.

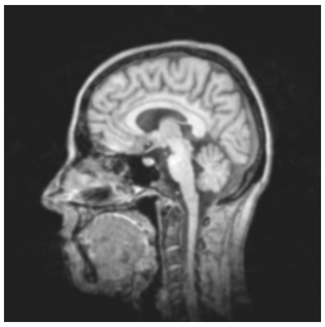
Chapter 3

Methodology

When using CUDA, it is fairly simple to express many kinds of parallel computations. However, due to the Tesla architecture, there are a lot of factors to take into consideration in order to achieve good performance, in particular when it comes to memory accesses and division of workload among threads. In this chapter, we will describe how four of the five steps of the algorithm were transformed from their sequential implementation to CUDA and OpenMP. We will also explain why we did not succeed in porting one of the steps. In addition, we describe our approach to timing the performance of our implementations.

3.1 Test images

We have used four different images for our testing, as shown in Figure 3.1. Figure 3.1(a) is one of the images used by Steger, which we used during development for correctness testing (by comparing the output of our rewritten application to that of the original one). Since it is somewhat small, we repeated it 16 times to get a larger image (but obviously with the same ratio of line points to image area), as shown in Figure 3.1(b). Figure 3.1(c) consists of 293 diagonal lines, and is meant to test the case where an image contains many long lines. Finally, Figure 3.1(d) is a seismic image (a vertical cross cut through a data set consisting of $201 \times 301 \times 1250$ points) provided by Schlumberger, which gives the best impression of what our application will be used for. Unfortunately, we did not get this data set until fairly late in the project, so during most of the project, only the first three images were available to us.


(a) MR scan (256 × 256)

(b) Repetitions of MR scan (1024 × 1024)

(c) Diagonal stripes (1024 × 1024)

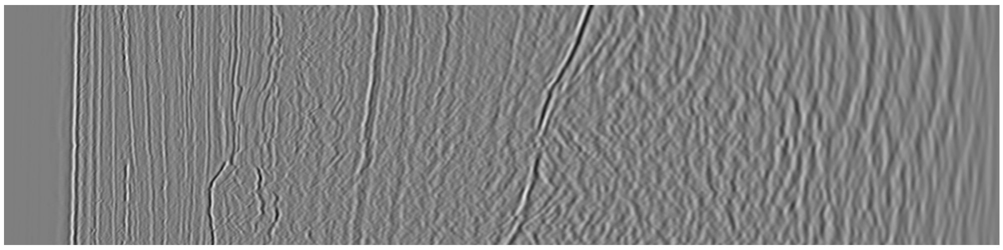

(d) Seismic image (1250 × 301)

Figure 3.1: The images used for testing.

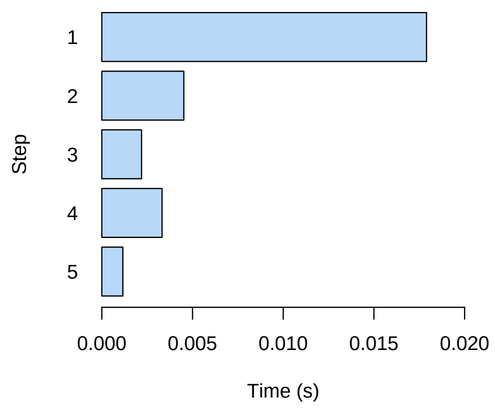
3.2 Porting the application

3.2.1 Initial analysis

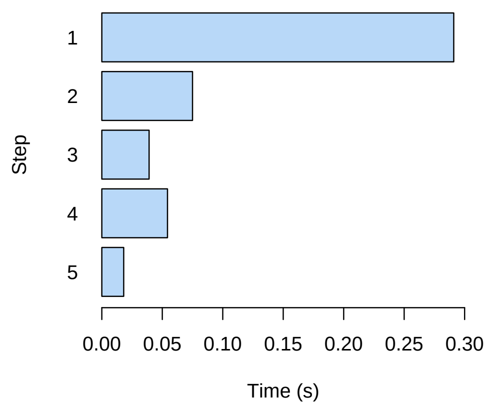
Schlumberger provided us with Steger's own implementation of the algorithm. It is written in C and is quite well-structured; each main step of the algorithm resides in its own set of functions in a separate file. This allowed us to take an iterative approach where we could take one algorithm step at a time and port it independently of the others. In order to determine which steps would yield the biggest gains from parallelisation, we timed the performance of the original implementation on three of the images. The results are shown in Figure 3.2.

Figure 3.2 shows that for images that only contain a moderate number of lines, the convolution phase dominates the run time, but if there are a lot of long lines, the last three phases begin to take a considerable amount of time too. Since we expect that the normal scenario for Schlumberger will be a moderate number of lines, we chose to first focus on the convolution step and convert the remaining steps if time allowed.

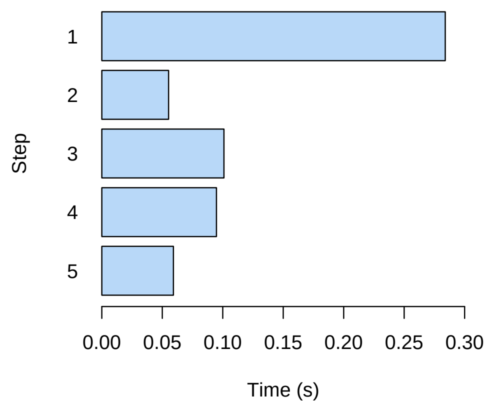
Amdahl's law (2.17) gives us an upper bound on what we can gain by this approach. If we use the values from the experiment that yielded Figure 3.2(b), the last three steps took 0.112 seconds out of the total time of 0.478 seconds, which yields a "sequential fraction" of $f = 0.112/0.478 = 0.234$. If we insert



(a) Input image: Figure 3.1(a).



(b) Input image: Figure 3.1(b).



(c) Input image: Figure 3.1(c).

Figure 3.2: Time spent by the original implementation on the different phases for three different images; $\sigma = 1.2$. Results averaged over 100 runs.

this into Amdahl's law and let the number of processors go to infinity, we get a speedup of $1/f = 4.27$. Keep in mind that this is a very optimistic bound, so in practice, the speedups will be lower. Still, this should be a good place to start.

3.2.2 General considerations

Floating point precision The existing implementation used double precision floating point values throughout the entire algorithm (except for representing the original image, which is stored as a byte array). Since current CUDA devices only have one ALU per SM that is capable of double precision calculations, a penalty factor would be incurred by using doubles. In addition, Steger mentions that the numerical calculations of the convolution masks uses an approximation error of 10^{-4}, since "for images that contain grey values in the range [0, 255] this precision is sufficient"[12]. Thus, we felt that the application could safely be rewritten to exclusively use single precision. In order to verify this, we compared the output of the program before and after rewriting it to use single precision. Since the program is capable of outputting line point coordinates to text files, such a comparison is straightforward. One can use tools like `diff` if the resulting files are equal or differ only in a few lines, or write a simple program that parses the files and compares the values. We have done the latter; see Section 4.1.1.

Memory allocation Since the original program is intended to analyse a single image each time it is launched, memory allocations are scattered around the code — the memory required by one phase is typically allocated in the beginning of the code for that phase and then freed at the end. This is not a problem as long as all one wants to do is to sporadically analyse single images. However, in seismic analysis, it is desirable to analyse a sequence of images that are slices from a three-dimensional data set, and in that case, a bit of time is wasted on memory allocations and deallocations. Assuming that all images are of the same size, it would be better to allocate memory for all phases in the beginning of the program and free everything at the end, unless the available memory is too small. This is only possible for the first two phases, however, since the memory requirement of the last three phases depends on the number of lines that are detected. Also, the third phase (line point linking) uses dynamic reallocations. Possible workarounds for this is to impose limits on the number of lines and line points that can be detected, or to find an upper bound on the number of line points and perform preallocation based on that bound. However, since the allocations take a fairly small amount of time compared to the algorithm steps themselves, we decided to be satisfied with preallocating memory for the first two steps.

Data transfer Data transfer between GPU and system memory often ends up taking a considerable amount of a GPU-accelerated application's time[7]. An obvious optimisation is therefore to avoid unnecessary data transfers — data that has been calculated by the GPU and is needed in a later stage should remain there as long as it is needed.

If possible, communication and computation can be interleaved so that calculations (both on the GPU and the CPU) take place during data transfer in order to hide the transfer latency. This has proven difficult to do here, however, since each step depends on the previous ones.

3.2.3 Step 1 — Convolution

This step lends itself well to parallelisation. The convolution problem is almost embarrassingly parallel — the calculation of the result value of a pixel is independent of (and needs no information from) the calculation of all other pixels, but requires as input the values of many neighbouring pixels. Since there are many memory accesses compared to the amount of computation that takes place, the way the memory accesses are structured might have a big impact on efficiency.

FFT

The existing implementation used a direct approach (nested loops) to calculate the convolution. However, as mentioned in Section 2.1.2, using Fourier transforms is normally a more efficient way. However, in this case (as long as σ remains fairly small), we have a small convolution mask (because the Gaussian kernels rapidly become approximately zero within a short distance from the origin). With small masks, the direct approach might still be faster. We wrote the following code to measure the performance of the CUFFT library; the results are presented in Table 4.2.

```
#include <cufft.h>
// In the following, we assume that 'image' and 'mask'
// point to device memory that has already been filled
cufftComplex * imageTransform, * maskTransform;
cufftHandle plan, inversePlan;
cudaMalloc((void**)&imageTransform, width * height * sizeof(
    cufftComplex));
cudaMalloc((void**)&maskTransform, width * height * sizeof(
    cufftComplex));
cufftPlan2d(&plan, width, height, CUFFT_R2C);
cufftPlan2d(&inversePlan, width, height, CUFFT_C2R);
cufftExecR2C(plan, mask, maskTransform);
// All steps above this point can be performed once at program
// startup - only the next three lines need to pe performed
// for each convolution
cufftExecR2C(plan, image, imageTransform);
multiply<<<blocks, threads>>>(imageTransform, maskTransform);
cufftExecC2R(reversePlan, transform, imageDevice);
// These two steps can be performed once at program termination
cufftDestroy(plan);
cufftDestroy(inversePlan);
```

Here, we assume that `blocks` and `threads` have been set to suitable values given the image size. Note that the mask can be transformed once (at program startup) rather than once for each image. The `multiply()` kernel for multiplying two images with each other (pixelwise multiplication, not matrix multiplication) looks like this:

```
__global__ static void multiply(float * a, float * b, long
    width, long height) {
  long r, c = blockIdx.x * blockDim.x + threadIdx.x;
  if (c >= width) return;
  for (r = 0; r < height; ++r)
    a[c] *= b[c];
}
```

Direct convolution

Given the results from the FFT tests and our belief that the mask will normally be small, we should be better off by porting the old code. The simplest possible way of porting existing code to CUDA is to let one thread to all of the work, so this was selected as the first step (at a point where the author had just started learning CUDA). The following must be done:

- Prefix the functions `convolve_rows_gauss()` and `convolve_cols_gauss()` with `__global__` to make them kernels.
- Rewrite `convolve_gauss()` to:
  - Allocate memory for the image and the convolution on the GPU
  - Copy the image to the GPU
  - Launch a single block with a single thread that executes `convolve_rows_gauss()`
  - Copy the convolution result back to main memory
  - Free the GPU memory

This is the basic way of structuring CPU/GPU cooperation.

Having only one thread obviously undermines the entire point of using GPUs. The work should be divided between many threads, but we are free to choose the granularity ourselves. At the other extreme end of the granularity scale, we can assign a single pixel to each thread (which may work well, since CUDA devices typically have very low thread overhead), but it is also possible to assign some other number of pixels to each thread. Ideally, different granularity choices should have been explored, but we ran out of time and ended up with assigning one column to each thread.

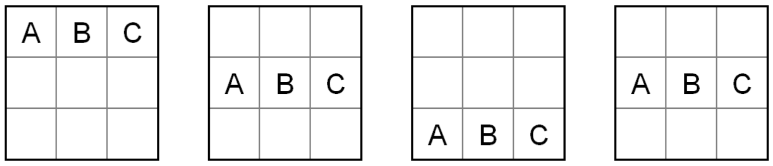
Coalesced memory accesses Since the Gaussian convolution kernel is separable, the convolution can be performed in two passes: first convolving each row independently of the others, and then each column. After the initial porting, the performances of these two steps differed drastically. The reason for this is the memory access patterns. In `convolve_rows_gauss()`[1], a column was assigned to each thread, and each thread traversed its column from top to bottom. In the inner loop, each thread reads the pixels in a small neighbourhood around the current pixel, from top to bottom — see Figure 3.3(a). This gives coalesced memory accesses. Since `convolve_cols_gauss` essentially performs the same calculation, just rotated ninety degrees, we first attempted to assign a row to each thread. However, as can be seen from Figure 3.3(b), each thread will then access values from different rows, which is very suboptimal — one memory transaction will be issued for each thread. A better solution is to assign a column to each thread again, as shown in Figure 3.3(c). This is much better, since threads access successive values in the same row, but in most iterations, the address accessed by the first thread is not aligned properly, which will degrade performance. Thus, the column step is still slower than the row step. There may be better ways of structuring the column step that will ensure completely coalesced accesses — perhaps by making the threads preload values in a coalesced manner, or by caching values since each input value is used several times.

On a related note: when the input image is loaded, it is represented as a byte array. The original code used those byte values during the first convolution, but this caused a performance hit in the kernel due to the fact that in order to obtain coalesced reads, each thread must access at values at least 32 bits wide. Therefore, we convert it to floating point immediately after reading it[2].

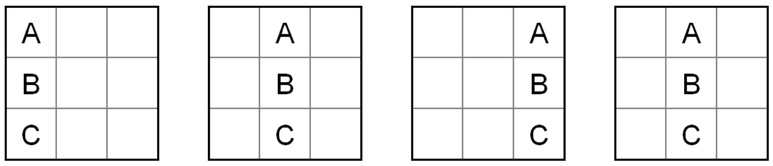
Constant memory The mask is being precalculated by the CPU, and it could in principle be put in the constant memory, which is optimised for the case where many threads access the same memory location simultaneously. However, we did not succeed in dynamically allocating constant memory (we might

[1]The convolution functions seemed to be named oddly in the original implementation, since the inner loop in `convolve_rows_gauss()` iterates over pixels in the same *column*, but we stuck to those names.

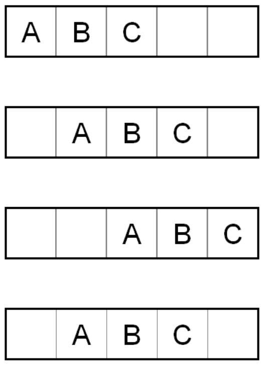
[2]To conserve memory, one might as well do the conversion while reading the image, but we tried not to modify more files than necessary.



(a) The CUDA version of `convolve_rows_gauss()` — each thread is given one column.



(b) The first attempt of converting `convolve_cols_gauss()` to CUDA — each thread is given one row.



(c) The final CUDA version of `convolve_cols_gauss()` — each thread calculates values for one given column, but needs to access neighbouring columns.

Figure 3.3: Memory access patterns for the convolution step. The mask width is 3. Three adjacent threads, labeled A, B, and C, are shown. The illustrations in each subfigure show, for successive time steps, which pixels are being accessed by which threads. In the first three time steps, the threads read the pixels from the input image that are needed to calculate one result pixel, and in the fourth time step, the result pixel is written (to another image).

have done something wrong, of course). We could use a predefined maximum size, but this would limit the possible sigma values. Therefore, we decided to put the masks in shared memory (to which the threads have very fast access) instead. The mask must be loaded from gloabl memory into shared memory at each kernel launch, and this is done by the first threads in each block. As a consequence, it is required that the block size is at least as big as the mask width in order for the entire mask to be loaded.

OpenMP The original implementation uses three different `for` loops: one for the interior part of the image, and one for each of the two border regions on each side, since the calculation of pixels close to the border might require pixels that fall outside of the image. As long as the mask size is small, the border regions are so small that attempting to parallelise them will be difficult due to thread management overhead. They can either be left unparallelised so that the master thread will handle them (which will be slow for large sigma values), or the loops can be combined (with a small cost due to the `if` statements that must be put inside the loop). We opted for the latter solution.

3.2.4 Step 2 — Ridge detection

This step is embarrassingly parallel. The results for each pixel are based only on the corresponding pixel values from the convolutions. In order to benefit from coalesced memory accesses, it is obviously desirable to assign neighbouring pixels in the same row to successive threads. The only question that remains is how many pixels within a column each thread should handle. However, since our initial (and easy to implement) choice of assigning an entire column to each thread yielded good results (see Figure 4.1) and further optimisations would only yield negligible overall improvements, we chose not to spend more time on this step.

The OpenMP implementation was even simpler; only a `parallel for` directive was added to the outermost loop. However, since the number of CPU threads is very limited compared to the number of GPU threads, we cannot expect to see the same speedup factors.

3.2.5 Step 3 — Line point joining

Unfortunately, this step involves dynamic arrays (which must be implemented by reallocating arrays that need to grow beyond their allocated size), but kernels cannot request memory (re)allocations. One can probably get around this by either setting limits on the sizes of the arrays[3], or by allocating a huge amount

[3]This should only be done after careful study of the problem domain (seismic imaging in this case) to make sure that the limits are reasonable and will not reduce the usefulness of the

of memory and implementing a memory allocator manually — however, due to the low speed of global memory, the author fears that this will be inefficient.An approach that might yield higher performance is to use shared memory, but one must keep in mind that the amount of shared memory is very limited (and that the more shared memory each block uses, the less is the number of blocks each SM can hold, so that parallelism decreases).

In addition, we face dependency problems because we do not know in advance which points belong to which lines. If we let two threads start on two different points, it might turn out that these are actually part of the same line. Then, one of the threads will have to yield to the other and let the other thread complete the traversal of the line, or perhaps the results that the two threads have accumulated so far (the points they have encountered on the line) can be combined. Still, one might get synchronisation issues.

While reallocations will not be a problem when using OpenMP, the other aforementioned problems will still be present, and so we have chosen not to implement this step, neither in CUDA nor in OpenMP.

3.2.6 Steps 4 and 5 — Line width; bias removal

Step 4 begins with a calculation of the gradient of the image. This calculation is embarrassingly parallel, and we wrote a separate CUDA kernel for it (`compute_gradient()` in `position.cu` in the `width` folder on the CD-ROM (see Appendix B)). Unfortunately, the remaining code is not as simple.

Since the line points were linked in step 3, we do not have the same kinds of dependencies in the final two steps — each line is independent of the other lines. However, there are dependencies within each line: there may be line points where the width could not be detected, and the width of those points will have to be interpolated from the surrounding pixels. It therefore seems that, at least as a first step, we should assign one thread to each line. We are doubtful about the efficiency of this approach, however. Lines may go in arbitrary directions, and for each line point, several surrounding pixels perpendicular to the line must be examined in order to locate the maximum gradient. Thus, we cannot expect to achieve coalesced memory accesses, and we believe this to be detrimental to performance (in particular in the absence of a cache — but it might be possible to use the shared memory to emulate a cache). In order to avoid penalties for divergent branches, we let the block size be 1 and assign one block to each line.

We started porting `width.c` to CUDA along the lines described above, but did not get the time to make any further improvements[4]. It was not entirely

application.

[4]the author admits that he spent too much time working on the first two steps, and also failed to take into account the amount of time that would have to be spent on producing this report

trivial, due to the fact that the input to these steps does not consist of simple arrays. The data structure for a detected line looks like this:

```
typedef struct {
  long  num;          /* number of points */
  float *row;         /* row coordinates of the line points */
  float *col;         /* column coordinates of the line points */
  float *angle;       /* angle of normal (measured from the row
      axis) */
  float *response;    /* response of line point (second
      derivative) */
  float *width_l;     /* width to the left of the line */
  float *width_r;     /* width to the right of the line */
  float *asymmetry;   /* asymmetry of the line point */
  float *contrast;    /* contrast of the line point */
  contour_class
      cont_class;     /* contour class (e.g., closed, no_junc) */
} contour;
```

Step 3 generates a collection of lines that is represented as an array of pointers to the individual `contour` instances. Since linked structures cannot be copied directly to the GPU[5], and we believed we should avoid any overhead that might be incurred by a large number of calls to `cudaMemcpy`, we decided that, for each field of `contour`, the value(s) of that field for *all* lines should be packed into one common array. Thus, we we made arrays called `cont_row`, `cont_col`, `cont_angle` and so on, each of length equal to the total number of line points. Then, for each line, the entires of `row` are copied into `cont_row`, and similarly for the other fields. These arrays are given as parameters to the kernels for step 4 and 5.

Since it is not possible to pass different parameter values to the different threads, all threads will receive the same array base addresses, and need to find out where the data for that thread's line resides. We accomplish this through the array `offsets`. Thread number i must skip all line points from all of the $i-1$ preceding threads, so `offsets` will be filled with cumulative sums of the number of line points. For instance, assume that four lines have been detected, and that they contain 3, 7, 6, and 8 points, respectively. Then, `offsets` will contain `{0, 3, 10, 16}`. The first action by thread number i is to advance the pointers to the arrays by `offset[i]`. The same strategy is used for a number of supporting arrays, such as `grad_l` and `grad_r`, which are required for temporary storage during the computation (so for these arrays, no data needs to be copied from CPU to GPU).

Step 5 mostly employs the same strategies as described above, and since it uses the same input data as step 4, as well as some of the results produced by

[5]They can be represented in the GPU, but will have to be rebuilt from scratch.

that step, we do not need to perform any CPU-GPU data transfers between steps 4 and 5. In addition, step 5 involves a large lookup table for the precalculated values of the inverse bias function; this table is placed in constant memory in order to speed up access to it.

The conversion to OpenMP was simpler since we did not need to convert the data structures. Again, we used the `parallel for` directive. The only problem was that the original implementation preallocates a number of arrays that are used to hold intermediate values for the line that is currently being processed; the sizes of these arrays are equal to the length of the longest line. We structured the OpenMP code the same way, but since the threads work on different lines, they need separate arrays[6].

3.3 Performance measurement

3.3.1 Timing

CUDA provides a `clock()` function that can be called from within a kernel. The difference between the returned values of two calls to `clock()` in the same kernel (within the execution of the same grid) tells the number of cycles the SM has spent between the two calls[9, sec. 4.3.3]. This is *not* the same as the time spent on that particular thread, since several blocks may be mapped onto the same SM, and there may be several warps in each block. Even if there was only one block with one warp, each thread would at most execute once every four cycles. This does, however, provide a more realistic picture of the time consumption, since it will also take into account stalls due to waiting for data from global memory.

In the host part of the code, a `clock()` function is also available, but its precision depends on the hardware and on the operating system, and on our test machines, the precision was too low, on the order of a few hundredths of a second. Therefore, at the expense of portability, we opted for a much more high-precision timer that is available on Intel processors and PowerPC, called the Time Stamp Counter. This counter is incremented on each clock cycle (on PowerPC, it might be controlled by a separate clock[15]), and it may be read by using the `rdtsc` instruction. Timing utilities have been implemented in `timer.c`, and they employ the function `rdtsc()`, taken from [15], that uses the aforementioned instruction. The function is located in `timer.h`, and if the application must be ported to an architecture without `rdtsc`, it is sufficient to change that file. Note that the processor frequency must be defined in `CPU_HZ` in `timer.c`.

[6]We implemented this by having the master thread allocate larger arrays and let each thread locate its appropriate array segment in its first loop iteration.

The advantage of this approach is that it can be used on all three implementations (the original one, CUDA, and OpenMP), as long as one takes care to call `cudaThreadSynchronize()` after the CUDA kernel launches one wants to time. The drawback of this approach is that it measures *wall time* (the amount of time that has passed), not the actual time spent by the CPU and GPU on this process (after all, the process may be preempted by the operating system), but averaged over many runs, this should impact all three implementations equally. If one is unfortunate, the process (or the OpenMP threads) will be moved between different CPUs (having different timer values) by the operating system, but this should be detectable as anomalies in the measurements, which can then be discarded. An alternative approach would be to use functionality provided by CUDA for directly timing kernel launches ([9, Sec. 4.5.3.8]), but this would only work for the CUDA implementation.

OpenMP The timing of the OpenMP code was problematic, since there was a large variance in the run times (in particular for the convolution step), probably caused by thread preemption. We believe that the best times provide the best picture of how fast the code may potentially run, and we chose to exclude the poorest results. This was done by sorting the timing results by the time spent on convolution and excluding the slowest 25% of the results. Since the code for the last two steps is intertwined, we obtained the timings for Figure 4.1 by measuring the total time for those steps, and measuring the time spent by the master thread (which takes more time than the other threads) on the code related to step 5. This time is then subtracted from the total time. While the individual results acquired this way are not entirely accurate, the sum of the times reported for steps 4 and 5 is correct.

3.3.2 Image sequences

The first kernel launch or memory operation is quite expensive due to setup overhead caused by the driver (around 0.12 seconds). In our measurements, we have launched the application several times in a row in order to get averaged results, which means that this cost has been incurred repeatedly. When a sequence of images is to be analysed, it would be more efficient to rewrite the application to process all of the images in a single run. We have taken care to make this a fairly simple job. As a proof of concept, we have integrated the line detector with a simple video player, located in `video.c`. This is almost entirely transparent to the line detector code[7]; the video player code receives a pointer

[7]Currently, the video player is a part of the same application as the single image analyser (a command line switch determines whether a video or an image will be processed), but it should be simple to separate them — or even better, to separate the core code into a dynamically loadable library.

to a callback function that it calls in order to process a video frame. The code for the algorithm itself is completely unaware of the video player.

Since the application is intended to be used for analysing image sequences and we have structured our memory allocations as mentioned in Section 3.2.2, we have chosen to omit the initial memory allocations and the final deallocations from the timings.

Chapter 4

Results

We have succeeded in porting four of the five steps of Steger's algorithm to CUDA. Unfortunately, due to time constraints, the design of the algorithm, and the author's lack of experience with CUDA, we have been able to obtain speedups only for the first two steps. In this chapter, we review timing results and discuss what might be done with the remaining steps. We also compare the performance of the CUDA implementation to that of the OpenMP implementation.

4.1 General

Licence Steger's implementation was released under the GNU General Public Licence (GPL) version 2, and therefore our modified application is released under the same licence.

4.1.1 Floating point precision

The original code, which uses double precision during several calculations, was run on some of the images from Figure 3.1 and was asked to output a text file describing the lines that were found. The same was done with our CUDA program. We wrote a small python program that compared the resulting files and calculated the deviations. The results are shown in Table 4.1. We regard these errors as highly acceptable (keep in mind that the locations are given in terms of pixel coordinates, so even the greatest deviation we found here only corresponds to around one hundredth of a pixel). This test has been run on the code version that can be found in the `cuda` folder on the CD-ROM (see Appendix B), which includes the CUDA code for only the first two steps, but uses floating point in all steps. We seem to have a small bug in our CUDA version of the last two steps, because one of the line points ends up with a high deviation in

Image	Size	σ	# of lines	# of line points	Maximum relative error
Figure 3.1(a)	256×256	1.2	142	2516	$3.324 \cdot 10^{-4}$
Figure 3.1(d)	1250×301	1.4	372	16503	$1.197 \cdot 10^{-2}$

Table 4.1: Floating point inaccuracies due to conversion from double precision to single precision. Sigma values were chosen subjectively to make a reasonable number of lines appear, given the image.

the value for the detected line contrast; apart from this single error, the results produced by the CUDA code are correct.

4.1.2 Memory limitations

One should be aware that Steger's algorithm requires a lot of memory if the image is large — this may be particularily problematic on the GPU, where memory cannot easily be added. The ridge detection step is the most demanding one; it requires five `float` arrays as input, each of size equal to the original image (these are produced by the convolution step), and produces five `float` arrays and one `byte` array to be sent to the linking phase. Thus, the memory requirement for this phase is at least $41 \cdot width \cdot height$ bytes. This is not a problem for the images that we use in this project, but it will prevent analysis of images that contain tens of millions of pixels. The requirements for the last three phases are not as severe, as they depend on the total number of detected line points, which is usually quite a bit smaller than the number of pixels.

4.2 Testing environment

The experiments were run on a machine containing an Intel Core2 Quad Q9550 with four cores at 2.83 GHz, with 4 GB of system memory. Note that neither Steger's implementation nor our CUDA rewrite uses CPU threads or more than one process. The compilers are `gcc` version 4.2.4 and `nvcc` version 0.2.1221. We are running Ubuntu with Linux kernel version 2.6.24-21.

The GPU is a Tesla C870, which contains 16 SMs with a total of 128 SPs, and 1.5 GB of global memory. The frequency of each SP is 1.35 GHz.

4.3 Benchmarking. Discussion

The timing results for the seismic image (Figure 3.1(d)) for two different sigma values that seemed appropriate for the image (depending on whether one would like to detect the thin lines in the rightmost part of the image or only the thicker ones) are shown in Figure 4.1. Note that for the CUDA version, the time required for the initial memory allocation (the first of which is very slow due to driver overhead) has been omitted, since this is a cost that will only be incurred once even if a sequence of images is analysed.

For reference, we repeat the titles of the steps, since the graphs are only labeled with the step numbers:

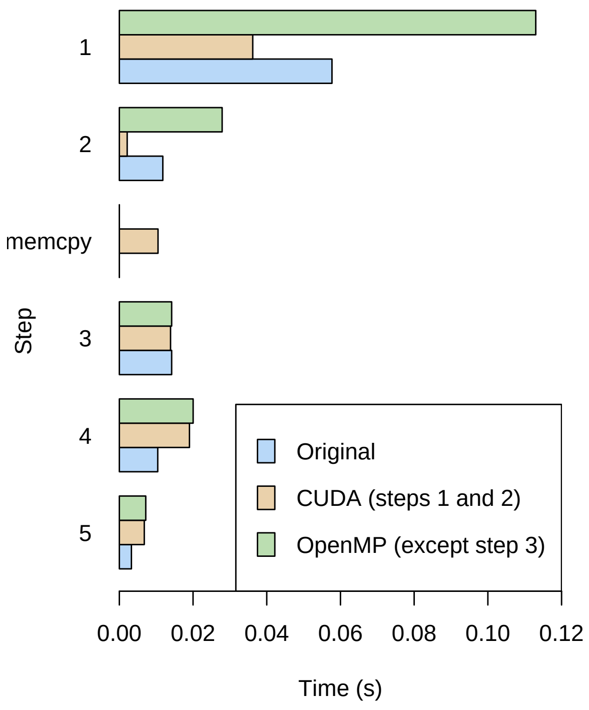
1. Convolution
2. Ridge detection
3. Line point joining
4. Line width determination
5. Bias removal

We will now discuss the results for each step and how they may be improved.

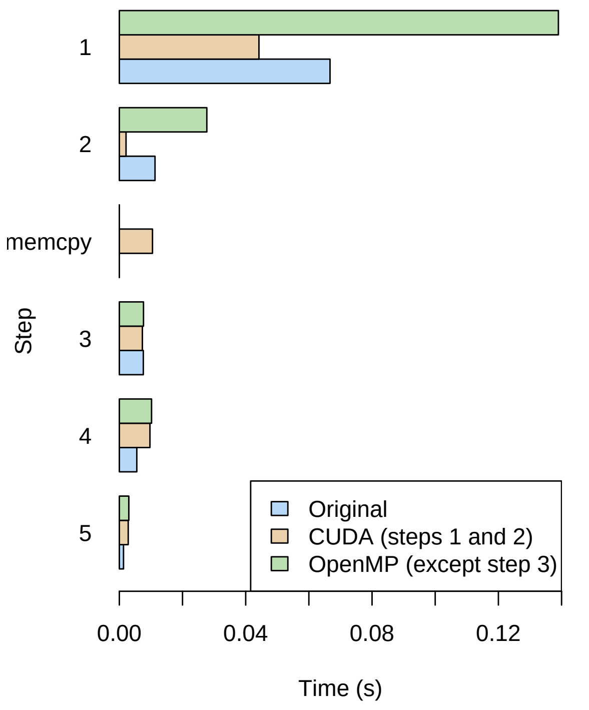
4.3.1 Step 1 — Convolution

Direct convolution vs. FFT Since we are using a separable convolution mask, it should be expected that the run time increases linearly in the mask width (not in the total mask size). For large σ values, the convolution mask may grow large enough that the FFT approach (whose runtime is independent of σ since it requires that the mask is of the same size as the image) may become worthwile. However, a large sigma value means that only very thick lines should be detected, and in that case, one might be better off by downscaling the image before feeding it to the line detector. Still, should some situation warrant such large sigma values, we recommend that our convolution implementation be discarded in favor of the FFT approach presented in Section 3.2.3. The time required by the three central lines of that code is presented in Table 4.2. In order to be comparable to the other convolution timings, they must be multiplied by five (since five convolutions are performed in the algorithm). The reason that the 1250×301 transform is slower than the 1024×1024 one is probably that FFT is most efficient when the image size is a power of two (which it probably cannot be expected to be in real world situations).

Row convolution vs. column convolution Since the memory access patterns of the two convolution substeps are different (the accesses of the column step are not properly aligned), their performance differ — the column step is around 2.3 times slower than the row step. It should be possible to further restructure the column step so that perfectly coalesced accesses are achieved.



(a) Input image: Figure 3.1(d), $\sigma = 1.4$



(b) Input image: Figure 3.1(d), $\sigma = 1.8$

Figure 4.1: Timing results for analysis of the seismic image at two different sigma values. Notice the scale; the lower diagram is compacted. Results averaged over 100 runs; see Section 3.3.1 for notes on OpenMP timing.

Image size	FFT time
256×256	0.000731
1024×1024	0.00695
1250×301	0.00929

Table 4.2: Time (in seconds) required for a single FFT followed by a multiplication with a pretransformed mask and then an inverse transform. Results averaged over 5 runs.

Thread granularity Regrettably, we did not get the time to explore the consequences of different thread granularities; rather, we simply assign an entire column to each thread (both in the row phase and in the column phase). This might not be optimal. Another consequence of this is that if the width of the image is too low (less than 512 in our case, since we have 16 SPs and we use 32 threads in each block) we do not utilise all available SPs, and performance will degrade. This ought to be improved.

OpenMP The OpenMP version required fairly little effort, and gave speedups of between 1.96 and 2.08 for the seismic image.

4.3.2 Step 2 — Ridge detection

Not surprisingly, this step is the one that benefited the most from parallellisation. This is probably due to the fact that there is quite a bit of calculation involved per pixel, and that the calculation of each pixel does not depend on any neighbouring pixels (as opposed to the convolution, where the calculation of each pixel requires the values for several of the neighbouring pixels). Therefore, there are no memory access patterns that can cause trouble as long as we let successive threads operate on successive pixels in the same row, so that we achieve coalesced global memory access. The speedup for the seismic image is around 13.1, independently of the sigma value. Unfortunately, this speedup does not impact the overall run time much, as long as some of the other steps remain unimproved.

It is particularily interesting to see how little work was required to achieve this speedup — one of the `for` loops in `compute_line_points()` was removed and replaced by an index calculation to determine which column to operate on. Thus, this is a great example of how easy it can be to use CUDA in certain situations.

The OpenMP version was also very simple to write, and it showed speedups of around 2.4 for the seismic image.

4.3.3 Step 3 — Line point joining

As discussed in 3.2.5, we expect this step to be difficult to parallelise well on any platform, due to the dependencies between the points. A CUDA implementation will be particularily difficult since this step also relies on dynamic memory reallocation. Fortunately, as long as the number of lines remains moderate, this step is fairly fast. However, as long as this step needs to be executed on the CPU, we are forced to copy a lot of intermediate results from step 2 from GPU to main memory, which is an extra performance penalty. In the graphs, there is a bar entitled "memcpy" to represent the time required for this copy operation. We felt that this should be presented separately from the time for the calculations themselves, in particular because potential future improvements may result in all steps being performed on the GPU, thereby eliminating this copy operation.

4.3.4 Steps 4 and 5 — Line width; bias removal

While we did manage to port these steps to CUDA, it regrettably resulted in severely degraded performance. When the CUDA version is run on the seismic image, the time required for the width step is between 4.7 and 4.9 times more than for the sequential implementation, and the bias step is between 3.3 and 3.5 times slower. The gradient calculation in the beginning of step 4 is by itself faster than the sequential version (after all, it is embarrassingly parallel). We did, however, not use the CUDA gradient calculation in our timings for Figure 4.1, since it is responsible for a fairly negligible fraction of the total time (and the gains are diminished by the time it takes to copy the result back to main memory).

In our implementation, the amount of parallelism is limited to the number of detected lines. We believe it will be hard to increase parallelism beyond that, since there are dependencies within each line (sequences of missing values may need to be interpolated from neighbouring points). Also, the arithmetic intensity is fairly low (meaning that there is not much computation per value that is read from memory), which may result in threads stalling while waiting for data from memory. The greatest problem is most likely related to the memory access pattern — in each iteration, a thread needs to load several values (line point position, gradient values etc.) that are located in different places in the global memory, and in the next iteration, it will move to any one of the eight neighbouring pixels. Other threads will work in different areas of the image. In such situations, coalesced memory accesses are hard to achieve. It will be challenging to overcome this, but it might possibly be done by issuing several threads per block and letting one thread act as a master thread and the others as slaves — for each iteration in the loop that walks through the points on the line, the threads can cooperate on loading neighbouring values from global memory into shared memory (thus achieving coalesced reads). After the loading step, the slaves wait for the master to complete the calculation of that line point. On the next few iterations, the master thread might find the values it needs in shared memory. This approach essentially amounts to using shared memory to emulate a cache. The drawback is that this will give good results only for horizontal lines. A better option might be to arrange the data differently in memory. If all the values that describe a line point are located successively in memory (rather than being stored in separate arrays), threads could cooperate on loading all required values for a point in a coalesced manner (rather than loading values for neighbouring points).

The OpenMP version was quite a bit simpler to write, and it shows speedups of around 2 for the width step and around 2.4 for the bias step. A cache conscious approach might be able to improve the run time.

4.3.5 Overall performance

With only two of the five steps having been speeded up by CUDA, the overall gains will clearly be limited. The total speedup of the implementation that uses CUDA for the first two steps and the original code (rewritten to use single precision) for the last three steps is 2.06 and 2.45 for the seismic images with sigma values 1.4 and 1.8, respectively. Images with many lines will cause the last three steps to dominate the overall run time, and any advantage from using the GPU is lost — however, we expect the number of lines to generally be moderate when doing seismic analysis. Still, a speedup factor of only a little more than two leaves something to be desired, given the hardware we are using. We have discussed a number of possible improvements, so with more time available and a larger degree of CUDA proficiency, it should be possible to improve the performance.

It is interesting to note that the OpenMP implementation (where all steps except the third have been ported) shows speedups of 1.87 and 2.03 for the same images, with far less coding effort.

Chapter 5

Conclusions and future work

The purpose of this project was to port an edge detection algorithm to CUDA in order to make it run on GPUs. Speedups for such algorithms would be useful for the oil field service provider Schlumberger, who intend to employ edge detection in the analysis of seismic images. By their suggestion, we chose an algorithm developed by Carsten Steger, which consists of five steps to be performed in sequence. We ported four of the steps to CUDA (the third step proved to be too challenging), but only achieved speedups on the two most time-consuming steps. The resulting speedup factors were 3.13 on the first step (convolution) and 13.1 on the second step (ridge detection). However, the unparallelised portion of the code limited the total gains. The code that combined the CUDA implementation of the first two steps with the original implementation of the remaining steps gave an overall speedup of between 2.06 and 2.45 on an image that is representative for the kind of analysis that Schlumberger wants to perform, using sigma values between 1.4 and 1.8. While this speedup certainly is not negligible, it is far from being impressive, and one must consider whether it is worth the investment in graphics cards and development time. In particular, since the OpenMP version we made yielded comparable overall speedups (between 1.87 and 2.03) while requiring less development time, the option of using multicore processors instead should be considered.

Why did we fail to obtain higher speedups? While the author's limited experience with CUDA certainly must take part of the blame (in particular for failing to produce better results on the convolution step), we also believe that the last three steps of Steger's algorithm are not particularily well suited for implementation on the Tesla GPU architecture due to their memory access patterns. The third step (line point joining) also seems to be difficult to parallelise in general.

Future work

Clearly, the last two steps need major improvements if their CUDA versions are to be of any use. The third step should be parallelised if possible — as long as the number of lines is moderate, it is a fast step (so improvements here will not greatly benefit the overall runtime, unless major speedups on all other steps are achieved), but an added benefit of having the entire computation reside on the GPU is that a lot of memory transfers between GPU to CPU will be avoided. Possible actions that could be taken in order to achieve notable overall speedups include:

- Changing the memory access pattern for the column convolution so that its performance will approach that of the row convolution.
- Refining the thread granularity in the convolution step.
- Reducing the number of memory reads in the convolution step by performing all calculations that involve a certain value once it has been loaded, and possibly also perform all five of the required convolutions simultaneously.
- Making more thorough measurements of the relationship between image size, sigma values and the performances of the direct convolution and the FFT based convolution. Then, the code could dynamically determine which method should be used.
- Porting step 3. If at all possible, it might require a major reworking of the code or the adoption of an entirely new strategy. A naïve parallelisation strategy which simply lets different threads start out at different line points will suffer from the fact that two line points may turn out to lie on the same line. In addition, the random access patterns generated by this step and the need for dynamic memory allocation cause major complications.
- In steps 4 and 5 (width determination and bias removal), one might try to use "slave" threads to preload data from global memory into shared memory in order to speed up memory access for the main threads, and to organize data differently in memory so that coalesced memory reads can be achieved.

As mentioned, the algorithm steps that caused problems for us are easier to implement on a traditional multiprocessor computer than on a GPU, since ordinary threads are allowed to perform memory allocations themselves and it is easier to efficiently use caches on an ordinary processor than to obtain coalesced memory accesses on a GPU. Examining the performance of the OpenMP implementation on processors with more cores might be worthwile.

Bibliography

[1] G. AMDAHL, *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*, in Proceedings of the AFIPS spring joint computer conference, 1967, pp. 483–485. [cited at p. 19]

[2] K. ASANOVÍC, R. BODIK, B. CATANZARO, J. GEBIS, P. HUSBANDS, K. KEUTZER, D. PATTERSON, W. PLISHKER, J. SHALF, S. WILLIAMS, AND K. YELICK, *The Landscape of Parallel Computing Research: A View from Berkeley*, Tech. Rep. UCB/EECS-2006-183, Electrical Engineering and Computer Sciences — University of California at Berkeley, December 2006. [cited at p. 19]

[3] D. BLYTHE, *Rise of the Graphics Processor*, Proceedings of the IEEE, 96 (2008). [cited at p. 11]

[4] S. CHE, M. BOYER, J. MENG, D. TARJAN, J. W. SHEAFFER, AND K. SKADRON, *A performance study of general-purpose applications on graphics processors using CUDA*, Journal of Parallel and Distributed Computing, 68 (2008). [cited at p. 15, 16]

[5] C. GASQUET, P. WITOMSKI, AND R. RYAN, *Fourier Analysis and Applications*, Springer, 1998. [cited at p. 3]

[6] E. KREYSZIG, *Advanced Engineering Mathematics*, Wiley, 9 ed., 2006. [cited at p. 5]

[7] L. C. LARSEN, *Utilizing GPUs on Cluster Computers*, Master's thesis, Norwegian University of Science and Technology, 2006. [cited at p. 25]

[8] J. NICKOLLS, I. BUCK, M. GARLAND, AND K. SKADRON, *Scalable Parallel Programming with CUDA*, ACM Queue, 6 (2008). [cited at p. 16]

[9] NVIDIA, *CUDA programming guide 2.0*, June 2008. `http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf`. [cited at p. ix, 10, 15, 16, 32, 33]

[10] J. D. OWENS, M. HOUSTON, D. LUEBKE, S. GREEN, J. E. STONE, AND J. C. PHILLIPS, *GPU Computing*, Proceedings of the IEEE, 96 (2008). [cited at p. 11]

[11] D. F. ROGERS, *Procedural Elements for Computer Graphics*, McGraw-Hill Book Company, 1985. [cited at p. 9]

[12] C. Steger, *An Unbiased Detector of Curvilinear Structures*, Tech. Rep. FGBV-96-03, Forschungsgruppe Bildverstehen (FG BV), Informatik IX — Technische Universität München, July 1996. [cited at p. 1, 6, 24]

[13] B. M. ter Haar Romney, L. M. Florack, A. H. Salden, and M. A. Viergever, *Higher order differential structure of images*, Image and Vision Computing, 12 (1994). [cited at p. 6]

[14] O. Yilmaz and S. M. Doherty, *Seismic Data Processing (Investigations in Geophysics, Vol 2)*, Society of Exploration, 1987. [cited at p. 1]

[15] K. Yoshii, *Time-stamp counter*. `http://www.mcs.anl.gov/~kazutomo/rdtsc.html`. [cited at p. 32]

Appendices

Appendix A

Code

In this appendix we have put the most important source files for our CUDA implementations. The full source code for our CUDA and OpenMP implementations as well as Steger's original code can be found on the accompanying CD-ROM; see Appendix B for instructions.

A.1 CUDA implementation — steps 1 and 2

This is the CUDA code that actually produced speedups. The full code is in the `cuda` folder on the CD-ROM.

A.1.1 `convol.cu`

```
/*  Convolve an image with the derivatives of Gaussians; part
   of detect-lines.
    Copyright (C) 1996-1998 Carsten Steger

    This program is free software; you can redistribute it and/
        or modify
    it under the terms of the GNU General Public License as
        published by
    the Free Software Foundation; either version 2, or (at your
         option)
    any later version.

    This program is distributed in the hope that it will be
        useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty
         of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See
       the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public
       License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
       */


#define CUDA_FILE
extern "C" {
#include "lines.h"
#include "timer.h"
}
#include <cuda.h>


/* Constants */

/* 1/sqrt(2*PI) */
#define SQRT_2_PI_INV 0.398942280401432677939946059935



/* Local function prototypes */

__global__ void convolve_rows_gauss __P((float * floatImage,
   float *mask, long n,
                                    float *h, long width, long
                                        height));
__global__ void convolve_cols_gauss __P((float *h, float *mask,
    long n, float *k,
                                    long width, long height));
static float *compute_gauss_mask_0 __P((long *num, double sigma
   ));
static float *compute_gauss_mask_1 __P((long *num, double sigma
   ));
static float *compute_gauss_mask_2 __P((long *num, double sigma
   ));

void checkCUDAError(const char* msg);


void checkCUDAError(const char *msg)
{
    cudaError_t err = cudaGetLastError();
    if( cudaSuccess != err)
    {
        fprintf(stderr, "Cuda error: %s: %s.\n", msg,
                                   cudaGetErrorString( err) );
        exit(EXIT_FAILURE);
    }
}

extern "C" {

/* Functions to compute the integral, and the 0th and 1st
    derivative of the
   Gaussian function 1/(sqrt(2*PI)*sigma)*exp(-0.5*x^2/sigma^2)
        */

/* Integral of the Gaussian function */
double phi0(double x, double sigma)
{
  return normal(x/sigma);
}



/* The Gaussian function */
double phi1(double x, double sigma)
{
  double t;

  t = x/sigma;
  return SQRT_2_PI_INV/sigma*exp(-0.5*t*t);
}



/* First derivative of the Gaussian function */
double phi2(double x, double sigma)
{
  double t;

  t = x/sigma;
  return -x*SQRT_2_PI_INV/pow(sigma,3.0)*exp(-0.5*t*t);
}
```

```
/* Functions to compute the one-dimensional convolution masks
    of the 0th, 1st,
   and 2nd derivative of the Gaussian kernel for a certain
      smoothing level
   given by sigma.  The mask is allocated by the function and
      given as the
   return value.  The caller must ensure that this memory is
      freed.  The
   output is intended to be used as an array with range [-num:
      num].  Therefore,
   the caller should add num to the return value.  Examples for
       the calling
   sequence can be found in convolve_gauss.  Examples for the
      usage of the
   masks are given in convolve_rows_gauss and
      convolve_cols_gauss. */

/* Gaussian smoothing mask */
static float *compute_gauss_mask_0(long * num, double sigma)
{
  long   i, n;
  float *h, *mask;

  n = MASK_SIZE(MAX_SIZE_MASK_0,sigma); /* Error < 0.001 on
      each side */
  h = (float *) calloc(2*n+1,sizeof(float));
  mask = h + n;
  for (i=-n+1;i<=n-1;i++)
    mask[i] = phi0(-i+0.5,sigma) - phi0(-i-0.5,sigma);
  mask[-n] = 1.0 - phi0(n-0.5,sigma);
  mask[n] = phi0(-n+0.5,sigma);
  *num = n;
  return h;
}



/* First derivative of Gaussian smoothing mask */
static float *compute_gauss_mask_1(long * num, double sigma)
{
  long   i, n;
  float *h, *mask;

  n = MASK_SIZE(MAX_SIZE_MASK_1,sigma); /* Error < 0.001 on
      each side */
  h = (float *) calloc(2*n+1,sizeof(float));
  mask = h + n;
  for (i=-n+1;i<=n-1;i++)
    mask[i] = phi1(-i+0.5,sigma) - phi1(-i-0.5,sigma);
  mask[-n] = -phi1(n-0.5,sigma);
  mask[n] = phi1(-n+0.5,sigma);
  *num = n;
  return h;
}



/* Second derivative of Gaussian smoothing mask */
static float *compute_gauss_mask_2(long * num, double sigma)
{
  long   i, n;
  float *h, *mask;

  n = MASK_SIZE(MAX_SIZE_MASK_2,sigma); /* Error < 0.001 on
      each side */
  h = (float *) calloc(2*n+1,sizeof(float));
  mask = h + n;
  for (i=-n+1;i<=n-1;i++)
    mask[i] = phi2(-i+0.5,sigma) - phi2(-i-0.5,sigma);
  mask[-n] = -phi2(n-0.5,sigma);
  mask[n] = phi2(-n+0.5,sigma);
  *num = n;
  return h;
}

}



/* Convolve an image with the derivatives of a Gaussian
    smoothing kernel.
   Since all of the masks are separable, this is done in two
       steps in the
   function convolve_gauss.  Firstly, the rows of the image are
        convolved by
   an appropriate one-dimensional mask in convolve_rows_gauss,
       yielding an
   intermediate float-image h.  Then the columns of this image
       are convolved
   by another appropriate mask in convolve_cols_gauss to yield
       the final
   result k.  At the border of the image the gray values are
       mirrored. */
```

```
/* Convolve the rows of an image with the derivatives of a
    Gaussian. */
__global__ void convolve_rows_gauss(float * image, float * mask
    , long n, float * h, long width, long height)
{
  long      j, r, c, l;
  float     sum;
  extern __shared__ float sharedData[];

  float * sharedMask = sharedData + n;

  c = blockIdx.x * blockDim.x + threadIdx.x;
  if (threadIdx.x < 2 * n + 1)
    sharedData[threadIdx.x] = mask[threadIdx.x]; // WARNING:
        assumes that width and blockDim >= 2 * n + 1
  __syncthreads();
  if (c >= width) return;

  // Inner region
  for (r=n; r<height-n; r++) {
    l = LINCOOR(r,c,width);
    sum = 0.0;
    for (j=-n;j<=n;j++)
      sum += (image[l+j*width])*sharedMask[j];
    h[l] = sum;
  }
  // Border regions
  for (r=0; r<n; r++) {
    l = LINCOOR(r,c,width);
    sum = 0.0;
    for (j=-n;j<=n;j++)
      sum += (image[LINCOOR(BR(r+j),c,width)])*sharedMask[j];
    h[l] = sum;
  }
  for (r=height-n; r<height; r++) {
    l = LINCOOR(r,c,width);
    sum = 0.0;
    for (j=-n;j<=n;j++)
      sum += (image[LINCOOR(BR(r+j),c,width)])*sharedMask[j];
    h[l] = sum;
  }
}



/* Convolve the columns of an image with the derivatives of a
    Gaussian. */
```

```
__global__ static void convolve_cols_gauss(float * h, float *
    mask, long n, float * k, long width, long height)
{
  long      j, r, c, l;
  float     sum;
  extern __shared__ float sharedData[];

  float * sharedMask = sharedData + n;

  c = blockIdx.x * blockDim.x + threadIdx.x;
  if (threadIdx.x < 2 * n + 1)
    sharedData[threadIdx.x] = mask[threadIdx.x]; // WARNING:
        assumes that width and blockDim >= 2 * n + 1
  __syncthreads();
  if (c >= width) return;

  /* Inner region */
  if (c >= n && c < width - n) {
    for (r=0; r<height; r++) {
      l = LINCOOR(r,c,width);
      sum = 0.0;
      for (j=-n;j<=n;j++)
        sum += h[l+j]*sharedMask[j];
      k[l] = sum;
    }
  }
  /* Border regions */
  else if (c < n) {
    for (r=0; r<height; r++) {
      l = LINCOOR(r,c,width);
      sum = 0.0;
      for (j=-n;j<=n;j++)
        sum += h[LINCOOR(r,BC(c+j),width)]*sharedMask[j];
      k[l] = sum;
    }
  }
  else {
    for (r=0; r<height; r++) {
      l = LINCOOR(r,c,width);
      sum = 0.0;
      for (j=-n;j<=n;j++)
        sum += h[LINCOOR(r,BC(c+j),width)]*sharedMask[j];
      k[l] = sum;
    }
  }
}
```

```
extern "C" {

static cputimer_t timer;
static float *h, *hDevice, *imageDevice, *maskDevice[3], *mask
    [3];
static long imageMemSize, maskSize[3];

void initCuda(long width, long height, long cudaDeviceNumber) {
  long i;
  startTimer(&timer);
  cudaSetDevice(cudaDeviceNumber);
  imageMemSize = width * height * sizeof(float);
  cudaMalloc((void **) &hDevice, imageMemSize);
  printTimeAndReset(&timer, "first time alloc");
  cudaMallocHost((void **)&h, imageMemSize);
  cudaMalloc((void **) &imageDevice, imageMemSize);
  mask[0] = compute_gauss_mask_0(&maskSize[0], opts.sigma);
  mask[1] = compute_gauss_mask_1(&maskSize[1], opts.sigma);
  mask[2] = compute_gauss_mask_2(&maskSize[2], opts.sigma);
  for (i = 0; i < 3; ++i) {
    cudaMalloc((void **) &maskDevice[i], (maskSize[i] * 2 + 1)
        * sizeof(float));
    cudaMemcpy(maskDevice[i], mask[i], (maskSize[i] * 2 + 1) *
        sizeof(float), cudaMemcpyHostToDevice);
  }
  initCuda_position(width, height);
  printTimeAndReset(&timer, "subsequent allocs");
}

void cleanupCuda() {
  long i;
  cleanupCuda_position();
  for (i = 0; i < 3; ++i)
    cudaFree(maskDevice[i]);
  cudaFree(imageDevice);
  cudaFree(hDevice);
  cudaFreeHost(h);
}



/* Convolve an image with a derivative of the Gaussian. */
void convolve_gauss(float * floatImage, float * kDevice, long
    width, long height, double sigma, long deriv_type)
{
  float  *hr, *hc;
  long    nr, nc, rowOp, colOp;
  long    numBlocks;
  if (deriv_type == DERIV_R) cudaMemcpy(imageDevice, floatImage
      , imageMemSize, cudaMemcpyHostToDevice); // The image only
       needs to be copied once per five convolutions

  switch (deriv_type) {
    case DERIV_R:
      rowOp = 1;
      colOp = 0;
      break;
    case DERIV_C:
      rowOp = 0;
      colOp = 1;
      break;
    case DERIV_RR:
      rowOp = 2;
      colOp = 0;
      break;
    case DERIV_RC:
      rowOp = 1;
      colOp = 1;
      break;
    case DERIV_CC:
      rowOp = 0;
      colOp = 2;
      break;
  }
  hr = maskDevice[rowOp];
  hc = maskDevice[colOp];
  nr = maskSize[rowOp];
  nc = maskSize[colOp];

  cudaThreadSynchronize();

  numBlocks = (long)ceil(width / 32.0f);
  convolve_rows_gauss<<<numBlocks, 32, 2 * nr + 1>>>(
      imageDevice,hr,nr,hDevice,width,height);
  cudaThreadSynchronize();
  checkCUDAError("convolve rows");


  convolve_cols_gauss<<<numBlocks, 32, 2 * nc + 1>>>(hDevice,hc
      ,nc,kDevice,width,height);
  cudaThreadSynchronize();
  checkCUDAError("convolve columns");
}
```

```
}
```

A.1.2 position.cu

```
/*  Extract line points from an image; part of detect-lines.
    Copyright (C) 1996-1998 Carsten Steger

    This program is free software; you can redistribute it and/
       or modify
    it under the terms of the GNU General Public License as
       published by
    the Free Software Foundation; either version 2, or (at your
        option)
    any later version.

    This program is distributed in the hope that it will be
       useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty
        of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See
       the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public
       License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
       */



#define CUDA_FILE
extern "C" {
#include "lines.h"
#include <time.h>
#include "timer.h"



/* Constants */

/* The pixel boundaries need to be enlarged slightly since in
   practice it
   frequently happens for neighboring pixels a and b that pixel
       a says a
   maximum lies within pixel b and vice versa.  This presents
      no problem since
   linking algoritm will take care of this. */
#define PIXEL_BOUNDARY 0.6



void checkCUDAError(const char* msg);


void checkCUDAError(const char *msg)
{
    cudaError_t err = cudaGetLastError();
    if( cudaSuccess != err)
    {
        fprintf(stderr, "Cuda error: %s: %s.\n", msg,
                                  cudaGetErrorString( err) );
        exit(EXIT_FAILURE);
    }
}

/* Local function prototypes */

/*static void compute_line_points __P((float *ku[5], byte *
   ismax, float *ev,
                                     float *nx, float *ny,
                                        float *px,
                                     float *py, long width,
                                        long height,
                                     double low, double high,
                                        long mode));*/



/* Solve the linear equation a*x+b=0 and return the result in t
     and the number
   of solutions in num. */
__device__ __host__ void solve_linear(float a, float b, float *
     t, long * num)
{
  if (a == 0.0f) {
    *num = 0;
  } else {
    *num = 1;
    *t = -b/a;
  }
}
```

```
/* Compute the eigenvalues and eigenvectors of the Hessian
    matrix given by
   dfdrr, dfdrc, and dfdcc, and sort them in descending order
       according to
   their absolute values. */
__device__ __host__ void compute_eigenvals(float dfdrr, float
    dfdrc, float dfdcc, float2 * eigval, float4 * eigvec)
{
  float theta, t, c, s, e1, e2, n1, n2; //, phi;

  /* Compute the eigenvalues and eigenvectors of the Hessian
      matrix. */
  if (dfdrc != 0.0f) {
    theta = 0.5*(dfdcc-dfdrr)/dfdrc;
    t = 1.0f/(fabs(theta)+sqrt(theta*theta+1.0f));
    if (theta < 0.0f) t = -t;
    c = 1.0f/sqrt(t*t+1.0f);
    s = t*c;
    e1 = dfdrr-t*dfdrc;
    e2 = dfdcc+t*dfdrc;
  } else {
    c = 1.0f;
    s = 0.0f;
    e1 = dfdrr;
    e2 = dfdcc;
  }
  n1 = c;
  n2 = -s;

  /* If the absolute value of an eigenvalue is larger than the
      other, put that
     eigenvalue into first position.  If both are of equal
         absolute value, put
     the negative one first. */
  if (fabs(e1) > fabs(e2)) {
    eigval->x = e1;
    eigval->y = e2;
    eigvec->x = n1;
    eigvec->y = n2;
    eigvec->z = -n2;
    eigvec->w = n1;
  } else if (fabs(e1) < fabs(e2)) {
    eigval->x = e2;
    eigval->y = e1;
    eigvec->x = -n2;
    eigvec->y = n1;
    eigvec->z = n1;
    eigvec->w = n2;
  } else {
    if (e1 < e2) {
      eigval->x = e1;
      eigval->y = e2;
      eigvec->x = n1;
      eigvec->y = n2;
      eigvec->z = -n2;
      eigvec->w = n1;
    } else {
      eigval->x = e2;
      eigval->y = e1;
      eigvec->x = -n2;
      eigvec->y = n1;
      eigvec->z = n1;
      eigvec->w = n2;
    }
  }
}

}

/* For each point in the image determine whether there is a
    local maximum of
   the second directional derivative in the direction (nx[l],ny
       [l]) within the
   pixels's boundaries.  If so, set ismax[l] to 2 if the
       eigenvalue ev[l] is
   larger than high, to 1 if ev[l] is larger than low, and to 0
        otherwise.
   Furthermore, put the sub-pixel position of the maximum into
       (px[l],py[l]).
   The parameter mode determines whether maxima (dark lines
       points) or minima
   (bright line points) should be selected.  The partial
       derivatives of the
   image are input as ku[]. */
__global__ static void compute_line_points(float * ku0, float *
     ku1, float * ku2, float * ku3, float * ku4, byte * ismax,
    float * ev, float * nx, float * ny, float * px, float * py,
    long width, long height, float low, float high, long mode)
{
  long   r, c, l;
  float  k0, k1, k2, k3, k4;
```

```
float2  eigval;
float4  eigvec;
float  a, b, t;
long   num;
float  n1, n2;
float  p1, p2;
float  val;
c = blockIdx.x * blockDim.x + threadIdx.x;
for (r=0; r<height; r++) {
    l = LINCOOR(r,c,width);
    k0 = ku0[l];
    k1 = ku1[l];
    k2 = ku2[l];
    k3 = ku3[l];
    k4 = ku4[l];
    ev[l] = 0.0f;
    nx[l] = 0.0f;
    ny[l] = 0.0f;
    compute_eigenvals(k2,k3,k4,&eigval,&eigvec);
    if (mode == MODE_LIGHT)
      val = -eigval.x;
    else
      val = eigval.x;
    if (val > 0.0f) {
      ev[l] = val;
      n1 = eigvec.x;
      n2 = eigvec.y;
      a = k2*n1*n1+2.0f*k3*n1*n2+k4*n2*n2;
      b = k0*n1+k1*n2;
      solve_linear(a,b,&t,&num);
      if (num != 0) {
        p1 = t*n1;
        p2 = t*n2;
        if (fabs(p1) <= PIXEL_BOUNDARY && fabs(p2) <=
            PIXEL_BOUNDARY) {
          if (val >= low) {
            if (val >= high)
              ismax[l] = 2;
            else
              ismax[l] = 1;
          }
          nx[l] = n1;
          ny[l] = n2;
          px[l] = r+p1;
          py[l] = c+p2;
        }
      }
      }
  }
}



extern "C" {
static float * k[5], * kDevice[5];
static byte *ismax, *ismaxDevice;
static float *ev, *n1, *n2, *p1, *p2;
static float *evDevice, *n1Device, *n2Device, *p1Device, *
   p2Device;
static int *num_resultDevice;

void initCuda_position(long width, long height) {
  long i, size = width * height * sizeof(float);
  for (i = 0; i < 5; ++i) {
    k[i] = (float *) malloc(size);
    cudaMalloc((void **) &kDevice[i], size);
  }
  cudaMalloc((void **) &num_resultDevice, sizeof(int));
  ismax = (byte *) xcalloc(width*height,sizeof(*ismax));
  ev = (float *) xcalloc(width*height,sizeof(*ev));
  n1 = (float *) xcalloc(width*height,sizeof(*n1));
  n2 = (float *) xcalloc(width*height,sizeof(*n2));
  p1 = (float *) xcalloc(width*height,sizeof(*p1));
  p2 = (float *) xcalloc(width*height,sizeof(*p2));
  cudaMalloc((void **) &ismaxDevice, width*height*sizeof(*ismax
      ));
  cudaMalloc((void **) &evDevice, width*height*sizeof(*evDevice
      ));
  cudaMalloc((void **) &n1Device, width*height*sizeof(*n1Device
      ));
  cudaMalloc((void **) &n2Device, width*height*sizeof(*n2Device
      ));
  cudaMalloc((void **) &p1Device, width*height*sizeof(*p1Device
      ));
  cudaMalloc((void **) &p2Device, width*height*sizeof(*p2Device
      ));
}

void cleanupCuda_position() {
  long i;
  cudaFree(p2Device);
  cudaFree(p1Device);
  cudaFree(n2Device);
  cudaFree(n1Device);
  cudaFree(evDevice);
  cudaFree(ismaxDevice);
  free(p2);
  free(p1);
  free(n2);
  free(n1);
  free(ev);
  free(ismax);
  cudaFree(num_resultDevice);
  for (i = 0; i < 5; ++i)
    cudaFree(kDevice[i]);
}



/* Main routine to detect lines in an image of dimension width
    * height.  The
   extracted lines are returned in result, while num_result is
      the number of
   detected lines.  The parameter sigma is the amount of
      smoothing that the
   Gaussian kernel performs, while low and high are the
      hysteresis thresholds
   used in the linking algorithm.  With mode, either bright or
      dark lines can
   be selected.  The parameter compute_width determines whether
       the line width
   should be extracted, while correct_pos determines whether
      the line width
   and position correction should be applied. */
void detect_lines (byte *image, long width, long height,
    contour ***result, long *num_result, double sigma, double
    low, double high, long mode, bool compute_width, bool
    correct_pos, bool extend_lines)
{
  long   i;
  float  *floatImage;
  long floatSize = width * height * sizeof(float);
  cputimer_t timer;
  long numBlocks;

  // Convert image to float
  floatImage = (float *) xcalloc(width * height, sizeof(float))
      ;
  for (i = 0; i < width * height; ++i)
    floatImage[i] = image[i];
```

```
// Step 1 - Convolution
startTimer(&timer);
convolve_gauss(floatImage,kDevice[0],width,height,sigma,
   DERIV_R);
convolve_gauss(floatImage,kDevice[1],width,height,sigma,
   DERIV_C);
convolve_gauss(floatImage,kDevice[2],width,height,sigma,
   DERIV_RR);
convolve_gauss(floatImage,kDevice[3],width,height,sigma,
   DERIV_RC);
convolve_gauss(floatImage,kDevice[4],width,height,sigma,
   DERIV_CC);
printTimeAndReset(&timer, "convolve");

// Step 2 - Ridge detection
cudaMemset(ismaxDevice, 0, width*height*sizeof(*ismaxDevice))
   ;
cudaMemset(evDevice, 0, floatSize);

numBlocks = (long)ceil(width / 32.0f);
compute_line_points<<<numBlocks, 32>>>(kDevice[0],kDevice[1],
   kDevice[2],kDevice[3],kDevice[4],ismaxDevice,evDevice,
   n1Device,n2Device,p1Device,p2Device,width,height,low,high,
   mode);
cudaThreadSynchronize();
checkCUDAError("ridge detection");
printTimeAndReset(&timer, "ridge detection");

for (i = 0; i < 2; ++i) { // Note: only the first two are
   used by cpu later
  cudaMemcpy(k[i], kDevice[i], width * height * sizeof(float)
     , cudaMemcpyDeviceToHost);
}
cudaMemcpy(num_result, num_resultDevice, sizeof(int),
   cudaMemcpyDeviceToHost);
cudaMemcpy(ismax, ismaxDevice, width * height * sizeof(byte),
    cudaMemcpyDeviceToHost);
cudaMemcpy(ev, evDevice, width * height * sizeof(float),
   cudaMemcpyDeviceToHost);
cudaMemcpy(n1, n1Device, width * height * sizeof(float),
   cudaMemcpyDeviceToHost);
cudaMemcpy(n2, n2Device, width * height * sizeof(float),
   cudaMemcpyDeviceToHost);
cudaMemcpy(p1, p1Device, width * height * sizeof(float),
   cudaMemcpyDeviceToHost);
cudaMemcpy(p2, p2Device, width * height * sizeof(float),
   cudaMemcpyDeviceToHost);
  printTimeAndReset(&timer, "memcpy from device");

  // Step 3 - Line point joining
  compute_contours(ismax,ev,n1,n2,p1,p2,k[0],k[1],result,
     num_result,sigma,
                  extend_lines,mode,low,high,width,height);
  printTimeAndReset(&timer, "contours");

  // Steps 4 and 5 - Line width determination and bias removal
  if (compute_width)
    compute_line_width(k[0],k[1],width,height,sigma,mode,
        correct_pos,*result,*num_result);
  printTimeAndReset(&timer, "width and bias");

  free(floatImage);
}
}
```

A.2 CUDA implementation — steps 4 and 5

This is the code for the last two steps (`width.cu`), and the required memory allocation code (`position.cu`). The full code is in the `width` folder on the CD-ROM. Note that not all of the files in that folder have been cleaned up, so there may be some stray pieces of commented-out code and unnecessary output statements. Also, as this code was written somewhat hastily, we forgot to call the appropriate memory deallocation functions at the end of the algorithm — in short, this is *not* production code.

A.2.1 `width.cu`

```
  /*  Extract the width of a line for each line point; part of
     detect-lines.
    Copyright (C) 1996-1998 Carsten Steger

    This program is free software; you can redistribute it and/
       or modify
    it under the terms of the GNU General Public License as
       published by
    the Free Software Foundation; either version 2, or (at your
       option)
    any later version.

    This program is distributed in the hope that it will be
       useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty
         of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See
        the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public
        License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
        */



#define CUDA_FILE

#include <cuda.h>
#include "correct.h"

extern "C" {
#include "lines.h"
}


/* Constants */

/* This constant is introduced because for very narrow lines
    the facet model
   width detection scheme sometimes extracts the line width too
        narrow.  Since
   the correction function has a very steep slope in that area,
        this will lead
   to lines of almost zero width, especially since the bilinear
        interpolation
   in correct.c will tend to overcorrect.  Therefore it is wise
        to make the
   extracted line width slightly larger before correction.  */
#define LINE_WIDTH_COMPENSATION 1.05

/* Minimum line width allowed (used for outlier check in
    fix_locations()) */
#define MIN_LINE_WIDTH 0.1

/* Maximum contrast allowed (used for outlier check in
    fix_locations()) */
#define MAX_CONTRAST 275.0
```

```
#define NUM_THREADS 3


/* Compute the eigenvalues and eigenvectors of the Hessian
   matrix given by
   dfdrr, dfdrc, and dfdcc, and sort them in descending order
      according to
   their absolute values. */
__device__ void compute_eigenvals(float dfdrr, float dfdrc,
   float dfdcc, float2 * eigval, float4 * eigvec)
{
  float theta, t, c, s, e1, e2, n1, n2;

  /* Compute the eigenvalues and eigenvectors of the Hessian
     matrix. */
  if (dfdrc != 0.0f) {
    theta = 0.5f*(dfdcc-dfdrr)/dfdrc;
    t = 1.0f/(fabs(theta)+sqrt(theta*theta+1.0f));
    if (theta < 0.0f) t = -t;
    c = 1.0f/sqrt(t*t+1.0f);
    s = t*c;
    e1 = dfdrr-t*dfdrc;
    e2 = dfdcc+t*dfdrc;
  } else {
    c = 1.0f;
    s = 0.0f;
    e1 = dfdrr;
    e2 = dfdcc;
  }
  n1 = c;
  n2 = -s;

  /* If the absolute value of an eigenvalue is larger than the
     other, put that
     eigenvalue into first position.  If both are of equal
        absolute value, put
     the negative one first. */
  if (fabs(e1) > fabs(e2)) {
    eigval->x = e1;
    eigval->y = e2;
    eigvec->x = n1;
    eigvec->y = n2;
    eigvec->z = -n2;
    eigvec->w = n1;
  } else if (fabs(e1) < fabs(e2)) {
    eigval->x = e2;
    eigval->y = e1;
    eigvec->x = -n2;
    eigvec->y = n1;
    eigvec->z = n1;
    eigvec->w = n2;
  } else {
    if (e1 < e2) {
      eigval->x = e1;
      eigval->y = e2;
      eigvec->x = n1;
      eigvec->y = n2;
      eigvec->z = -n2;
      eigvec->w = n1;
    } else {
      eigval->x = e2;
      eigval->y = e1;
      eigvec->x = -n2;
      eigvec->y = n1;
      eigvec->z = n1;
      eigvec->w = n2;
    }
  }
}

/* Solve the linear equation a*x+b=0 and return the result in t
     and the number
   of solutions in num. */
__device__ void solve_linear(float a, float b, float * t, long
   * num)
{
  if (a == 0.0f) {
    *num = 0;
  } else {
    *num = 1;
    *t = -b/a;
  }
}



/* Modified Bresenham algorithm.  It returns in line all pixels
     that are
   intersected by a half line less than length away from the
       point (px,py)
   along the direction (nx,ny).  The point (px,py) must lie
       within the pixel
   of the origin, i.e., fabs(px) <= 0.5 and fabs(py) <= 0.5. */
```

```
__device__ void bresenham_width(float nx, float ny, float px,
    float py, float length, offset * line, long * num_points)
{
  int i, n, x, y, s1, s2, xchg, maxit;
  float e, dx, dy, t;

  x = 0;
  y = 0;
  dx = ABS(nx);
  dy = ABS(ny);
  s1 = SGN(nx);
  s2 = SGN(ny);
  px *= s1;
  py *= s2;
  if (dy > dx) {
    t = dx;
    dx = dy;
    dy = t;
    t = px;
    px = py;
    py = t;
    xchg = 1;
  } else {
    xchg = 0;
  }
  maxit = ceil(length*dx);
  e = (0.5f-px)*dy/dx-(0.5f-py);
  n = 0;
  for (i=0; i<=maxit; i++) {
    line[n].x = x;
    line[n].y = y;
    n++;
    while (e >= -1e-8) {
      if (xchg) x += s1;
      else y += s2;
      e--;
      if (e > -1) {
        line[n].x = x;
        line[n].y = y;
        n++;
      }
    }
    if (xchg) y += s2;
    else x += s1;
    e += dy/dx;
  }
  *num_points = n;
```

```
}



/* Fill gaps in the arrays master, slave1, and slave2, i.e.,
   points where
   master=0, by interpolation (interior points) or
      extrapolation (end points).
   The array master will usually be the width of the line,
      while slave1 and
   slave2 will be values that depend on master[i] being 0, e.g
      ., the gradient
   at each line point.  The arrays slave1 and slave2 can be
      NULL. */
__device__ void fill_gaps(float * master, float * slave1, float
     * slave2, long num_points, float * row, float * col)
{
  long    i, j, k, s, e;
  float  m_s, m_e, s1_s, s1_e, s2_s, s2_e, d_r, d_c, arc_len,
      len;

  for (i=0; i<num_points; i++) {
    if (master[i] == 0) {
      for (j=i+1; j<num_points; j++) {
        if (master[j] > 0)
          break;
      }
      m_s = 0;
      m_e = 0;
      s1_s = 0;
      s1_e = 0;
      s2_s = 0;
      s2_e = 0;
      if (i > 0 && j < num_points-1) {
        s = i;
        e = j-1;
        m_s = master[s-1];
        m_e = master[e+1];
        if (slave1 != NULL) {
          s1_s = slave1[s-1];
          s1_e = slave1[e+1];
        }
        if (slave2 != NULL) {
          s2_s = slave2[s-1];
          s2_e = slave2[e+1];
        }
      } else if (i > 0) {
        s = i;
        e = num_points-2;
        m_s = master[s-1];
        m_e = master[s-1];
        master[e+1] = m_e;
        if (slave1 != NULL) {
          s1_s = slave1[s-1];
          s1_e = slave1[s-1];
          slave1[e+1] = s1_e;
        }
        if (slave2 != NULL) {
          s2_s = slave2[s-1];
          s2_e = slave2[s-1];
          slave2[e+1] = s2_e;
        }
      } else if (j < num_points-1) {
        s = 1;
        e = j-1;
        m_s = master[e+1];
        m_e = master[e+1];
        master[s-1] = m_s;
        if (slave1 != NULL) {
          s1_s = slave1[e+1];
          s1_e = slave1[e+1];
          slave1[s-1] = s1_s;
        }
        if (slave2 != NULL) {
          s2_s = slave2[e+1];
          s2_e = slave2[e+1];
          slave2[s-1] = s2_s;
        }
      } else {
        s = 1;
        e = num_points-2;
        m_s = master[s-1];
        m_e = master[e+1];
        if (slave1 != NULL) {
          s1_s = slave1[s-1];
          s1_e = slave1[e+1];
        }
        if (slave2 != NULL) {
          s2_s = slave2[s-1];
          s2_e = slave2[e+1];
        }
      }
      arc_len = 0;
      for (k=s; k<=e+1; k++) {
        d_r = row[k]-row[k-1];
        d_c = col[k]-col[k-1];
        arc_len += sqrt(d_r*d_r+d_c*d_c);
      }
      len = 0;
      for (k=s; k<=e; k++) {
        d_r = row[k]-row[k-1];
        d_c = col[k]-col[k-1];
        len += sqrt(d_r*d_r+d_c*d_c);
        master[k] = (arc_len-len)/arc_len*m_s+len/arc_len*m_e;
        if (slave1 != NULL)
          slave1[k] = (arc_len-len)/arc_len*s1_s+len/arc_len*
              s1_e;
        if (slave2 != NULL)
          slave2[k] = (arc_len-len)/arc_len*s2_s+len/arc_len*
              s2_e;
      }
      i = j;
    }
  }
}



/* First derivative of the Gaussian function */
__device__ double phi2(double x, double sigma)
{
  double t;
  t = x/sigma;
  return -x*SQRT_2_PI_INV/powf(sigma,3.0f)*exp(-0.5f*t*t);
}



/* Return the correct line width w and asymmetry h, and a line
   position
   correction correct for a line with extracted width w_est and
       extracted
   gradient ratio r_est for a given sigma.  Furthermore, return
       the line width
   on the weak and strong side of the line.  These values are
      obtained by
   bilinear interpolation from the table ctable. */
__device__ bool line_corrections(float sigma, float w_est,
   float r_est, float *w, float *h, float *correct, float *
   w_strong, float *w_weak)
{
  long   i_we,i_re;
  bool   is_valid;
  float a,b;

  w_est = w_est/sigma;
  if (w_est < 2 || w_est > 6 || r_est < 0 || r_est > 1) {
    *w = 0;
    *h = 0;
    *correct = 0;
    *w_strong = 0;
    *w_weak = 0;
    return 1;
  }
  i_we = floor((w_est-2)*10);
  i_re = floor(r_est*20);
  if (i_we == 40)
    i_we = 39;
  if (i_re == 20)
    i_re = 19;
  is_valid = ctable[i_re][i_we].is_valid && ctable[i_re][i_we
      +1].is_valid &&
             ctable[i_re+1][i_we].is_valid && ctable[i_re+1][
                 i_we+1].is_valid;
  a = (w_est-2)*10-i_we;
  b = r_est*20-i_re;
  *w = BILINEAR(a,b,w)*sigma;
  *h = BILINEAR(a,b,h);
  *correct = BILINEAR(a,b,correction)*sigma;//TODO
  *w_strong = BILINEAR(a,b,w_strong)*sigma;
  *w_weak = BILINEAR(a,b,w_weak)*sigma;
  return !is_valid;
}



/* Correct the extracted line positions and widths.  The
    algorithm first closes
   gaps in the extracted data width_l, width_r, grad_l, and
       grad_r to provide
   meaningful input over the whole line.  Then the correction
       is calculated.
   After this, gaps that have been introduced by the width
       correction are again
   closed.  Finally, the position correction is applied if
       correct_pos is set.
   The results are returned in width_l, width_r, and cont. */
__global__ void fix_locations( long * offsets,
float *width_l, float *width_r, float *grad_l, float *grad_r,
    float *correction, float *contr, float *asymm,
float sigma, long mode, bool correct_pos, long * cont_num,
    contour_class * cont_classes, float * cont_angle, float *
    cont_response, float * cont_row, float * cont_col)
{
  long    i, l;
  float  px, py;
  float  nx, ny;
  float  w_est, r_est, w_real, h_real, corr, w_strong, w_weak;
  float  correct, asymmetry, response, width, contrast;
  bool    weak_is_r;
  bool    correct_start, correct_end;
  long cont_class, num_points;

  l = blockDim.x * blockIdx.x + threadIdx.x;
  num_points = cont_num[l];
  cont_class = cont_classes[l];
  cont_angle += offsets[l];
  cont_response += offsets[l];
  cont_row += offsets[l];
  cont_col += offsets[l];
  width_r += offsets[l];
  width_l += offsets[l];
  grad_r += offsets[l];
  grad_l += offsets[l];
  correction += offsets[l];
  asymm += offsets[l];
  contr += offsets[l];


  fill_gaps(width_l,grad_l,NULL,num_points, cont_row, cont_col)
      ;
  fill_gaps(width_r,grad_r,NULL,num_points, cont_row, cont_col)
      ;

  /* Calculate true line width, asymmetry, and position
      correction. */
  if (correct_pos) {
    /* Do not correct the position of a junction point if its
        width is found
       by interpolation, i.e., if the position could be
           corrected differently
       for each junction point, thereby destroying the junction
           . */
    correct_start = ((cont_class == cont_no_junc ||
                      cont_class == cont_end_junc ||
                      cont_class == cont_closed) &&
                     (width_r[0] > 0 && width_l[0] > 0));
    correct_end = ((cont_class == cont_no_junc ||
                    cont_class == cont_start_junc ||
                    cont_class == cont_closed) &&
                   (width_r[num_points-1] > 0 && width_l[
                      num_points-1] > 0));
    /* Calculate the true width and assymetry, and its
        corresponding
       correction for each line point. */

    for (i=0; i<num_points; i++) {
      if (width_r[i] > 0 && width_l[i] > 0) {
        w_est = (width_r[i]+width_l[i])*LINE_WIDTH_COMPENSATION
            ;
        if (grad_r[i] <= grad_l[i]) {
          r_est = grad_r[i]/grad_l[i];
          weak_is_r = TRUE;
        } else {
          r_est = grad_l[i]/grad_r[i];
          weak_is_r = FALSE;
        }
        line_corrections(sigma,w_est,r_est,&w_real,&h_real,&
            corr,&w_strong,&w_weak);
        w_real /= LINE_WIDTH_COMPENSATION;
        corr /= LINE_WIDTH_COMPENSATION;
        width_r[i] = w_real;
        width_l[i] = w_real;
        if (weak_is_r) {
          asymm[i] = h_real;
          correction[i] = -corr;
        } else {
          asymm[i] = -h_real;
          correction[i] = corr;
        }
      }
    }


    fill_gaps(width_l,correction,asymm,num_points, cont_row,
        cont_col);
    for (i=0; i<num_points; i++)
      width_r[i] = width_l[i];
    /* Adapt the correction for junction points if necessary.
        */
    if (!correct_start)
      correction[0] = 0.0f;
```

```
    if (!correct_end)
      correction[num_points-1] = 0.0f;

    for (i=0; i<num_points; i++) {
      px = cont_row[i];
      py = cont_col[i];
      nx = cos(cont_angle[i]);
      ny = sin(cont_angle[i]);
      px = px+correction[i]*nx;
      py = py+correction[i]*ny;
      cont_row[i] = px;
      cont_col[i] = py;
    }

    // Calculate the true contrast.
    for (i=0; i<num_points; i++) {
      response = cont_response[i];
      asymmetry = fabs(asymm[i]);
      correct = fabs(correction[i]);
      width = width_l[i];
      if (width < MIN_LINE_WIDTH)
        contrast = 0;
      else
        contrast =
          (response/fabs(phi2(correct+width,sigma)+
                         (asymmetry-1)*phi2(correct-width,sigma
                            )));
      if (contrast > MAX_CONTRAST)
        contrast = 0;
      contr[i] = contrast;
    }
    fill_gaps(contr,NULL,NULL,num_points, cont_row, cont_col);
       //todo
  }
}



/* Extract the line width by using a facet model line detector
   on an image of
   the absolute value of the gradient. */
__global__ void compute_line_width(float *grad, long width,
   long height, float sigma, long mode, bool correct_pos,
long * offsets, long * cont_num, float * cont_row, float *
   cont_col, float * cont_angle,
float * width_r, float * width_l, float * grad_r, float *
   grad_l)
```

```
{
  long    i, j, k;
  long    r, c;
  long    x, y, dir;
  long    num_line;
  long    num_points;
  float  d, dr, dc, drr, drc, dcc;
  float  i1, i2, i3, i4, i5, i6, i7, i8, i9;
  float  t1, t2, t3, t4, t5, t6;
  float2  eigval;
  float4  eigvec;
  float  a, b, t;
  long    num;
  float tf;
  float  nx, ny;
  float  n1, n2;
  float  p1, p2;
  float  val;
  float  px, py;
  extern __shared__ offset sharedData[];
  offset * line = sharedData;

  i = blockDim.x * blockIdx.x + threadIdx.x;
  num_points = cont_num[i];
  cont_row += offsets[i];
  cont_col += offsets[i];
  cont_angle += offsets[i];
  width_r += offsets[i];
  width_l += offsets[i];
  grad_r += offsets[i];
  grad_l += offsets[i];

    for (j=0; j<num_points; j++) {
      px = cont_row[j];
      py = cont_col[j];
      r = floor(px+0.5f);
      c = floor(py+0.5f);
      nx = cos(cont_angle[j]);
      ny = sin(cont_angle[j]);
      // Compute the search line.
      bresenham_width(nx,ny,0.0f,0.0f,MAX_LINE_WIDTH,line,&
         num_line);
      width_r[j] = width_l[j] = 0;
      // Look on both sides of the line.
      for (dir=-1; dir<=1; dir+=2) {
        for (k=0; k<num_line; k++) {
          x = BR(r+dir*line[k].x);
```

```
y = BC(c+dir*line[k].y);
i1 = grad[LINCOOR(BR(x-1),BC(y-1),width)];
i2 = grad[LINCOOR(BR(x-1),y,width)];
i3 = grad[LINCOOR(BR(x-1),BC(y+1),width)];
i4 = grad[LINCOOR(x,BC(y-1),width)];
i5 = grad[LINCOOR(x,y,width)];
i6 = grad[LINCOOR(x,BC(y+1),width)];
i7 = grad[LINCOOR(BR(x+1),BC(y-1),width)];
i8 = grad[LINCOOR(BR(x+1),y,width)];
i9 = grad[LINCOOR(BR(x+1),BC(y+1),width)];
t1 = i1+i2+i3;
t2 = i4+i5+i6;
t3 = i7+i8+i9;
t4 = i1+i4+i7;
t5 = i2+i5+i8;
t6 = i3+i6+i9;
dr = (t3-t1)/6;
dc = (t6-t4)/6;
drr = (t1-2*t2+t3)/6;
dcc = (t4-2*t5+t6)/6;
drc = (i1-i3-i7+i9)/4;
compute_eigenvals(2*drr,drc,2*dcc,&eigval,&eigvec);
val = -eigval.x;
if (val > 0.0f) {
  n1 = eigvec.x;
  n2 = eigvec.y;
  a = 2.0f*(drr*n1*n1+drc*n1*n2+dcc*n2*n2);
  b = dr*n1+dc*n2;
  solve_linear(a,b,&tf,&num);
  t = tf;
  if (num != 0) {
    p1 = t*n1;
    p2 = t*n2;
    if (fabs(p1) <= 0.5f && fabs(p2) <= 0.5f) {
      // Project the maximum point position
          perpendicularly onto the search line.
      a = 1;
      b = nx*(px-(r+dir*line[k].x+p1))+ny*(py-(c+dir*
          line[k].y+p2));
      solve_linear(a,b,&tf,&num);
      t = tf;
      d = (-i1+2*i2-i3+2*i4+5*i5+2*i6-i7+2*i8-i9)/9;
      if (dir == 1) {
        grad_r[j] = d+p1*dr+p2*dc+p1*p1*drr+p1*p2*drc
            +p2*p2*dcc;
        width_r[j] = fabs(t);
      } else {
```

```
                  grad_l[j] = d+p1*dr+p2*dc+p1*p1*drr+p1*p2*drc
                      +p2*p2*dcc;
                  width_l[j] = fabs(t);
                }
                break;
              }
            }
          }
        }
      }
    }
}
```

A.2.2 position.cu

```
/*  Extract line points from an image; part of detect-lines.
    Copyright (C) 1996-1998 Carsten Steger

    This program is free software; you can redistribute it and/
        or modify
    it under the terms of the GNU General Public License as
        published by
    the Free Software Foundation; either version 2, or (at your
        option)
    any later version.

    This program is distributed in the hope that it will be
        useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty
        of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See
        the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public
        License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
        */



#define CUDA_FILE
extern "C" {
#include "lines.h"
#include <time.h>
#include "timer.h"
#include <cuda.h>


/* Constants */

/* The pixel boundaries need to be enlarged slightly since in
   practice it
   frequently happens for neighboring pixels a and b that pixel
       a says a
   maximum lies within pixel b and vice versa.  This presents
      no problem since
   linking algoritm will take care of this. */
#define PIXEL_BOUNDARY 0.6



void checkCUDAError(const char* msg);


void checkCUDAError(const char *msg)
{
    cudaError_t err = cudaGetLastError();
    if( cudaSuccess != err)
    {
        fprintf(stderr, "Cuda error: %s: %s.\n", msg,
                                    cudaGetErrorString( err) );
        exit(EXIT_FAILURE);
    }
}

/* Local function prototypes */

/*static void compute_line_points __P((float *ku[5], byte *
   ismax, float *ev,
                                     float *nx, float *ny,
                                        float *px,
                                     float *py, long width,
                                        long height,
                                     double low, double high,
                                        long mode));*/


}

/* Solve the linear equation a*x+b=0 and return the result in t
    and the number
   of solutions in num. */
```

```
__device__ void solve_linear(float a, float b, float * t, long
   * num)
{
  if (a == 0.0f) {
    *num = 0;
  } else {
    *num = 1;
    *t = -b/a;
  }
}



/* Compute the eigenvalues and eigenvectors of the Hessian
   matrix given by
   dfdrr, dfdrc, and dfdcc, and sort them in descending order
      according to
   their absolute values. */
__device__ void compute_eigenvals(float dfdrr, float dfdrc,
   float dfdcc, float2 * eigval, float4 * eigvec)
{
  float theta, t, c, s, e1, e2, n1, n2; //, phi;

  /* Compute the eigenvalues and eigenvectors of the Hessian
     matrix. */
  if (dfdrc != 0.0f) {
    theta = 0.5*(dfdcc-dfdrr)/dfdrc;
    t = 1.0f/(fabs(theta)+sqrt(theta*theta+1.0f));
    if (theta < 0.0f) t = -t;
    c = 1.0f/sqrt(t*t+1.0f);
    s = t*c;
    e1 = dfdrr-t*dfdrc;
    e2 = dfdcc+t*dfdrc;
  } else {
    c = 1.0f;
    s = 0.0f;
    e1 = dfdrr;
    e2 = dfdcc;
  }
  n1 = c;
  n2 = -s;

  /* If the absolute value of an eigenvalue is larger than the
     other, put that
     eigenvalue into first position.  If both are of equal
        absolute value, put
     the negative one first. */
  if (fabs(e1) > fabs(e2)) {
    eigval->x = e1;
    eigval->y = e2;
    eigvec->x = n1;
    eigvec->y = n2;
    eigvec->z = -n2;
    eigvec->w = n1;
  } else if (fabs(e1) < fabs(e2)) {
    eigval->x = e2;
    eigval->y = e1;
    eigvec->x = -n2;
    eigvec->y = n1;
    eigvec->z = n1;
    eigvec->w = n2;
  } else {
    if (e1 < e2) {
      eigval->x = e1;
      eigval->y = e2;
      eigvec->x = n1;
      eigvec->y = n2;
      eigvec->z = -n2;
      eigvec->w = n1;
    } else {
      eigval->x = e2;
      eigval->y = e1;
      eigvec->x = -n2;
      eigvec->y = n1;
      eigvec->z = n1;
      eigvec->w = n2;
    }
  }
}



/* For each point in the image determine whether there is a
    local maximum of
   the second directional derivative in the direction (nx[l],ny
       [l]) within the
   pixels's boundaries.  If so, set ismax[l] to 2 if the
       eigenvalue ev[l] is
   larger than high, to 1 if ev[l] is larger than low, and to 0
        otherwise.
   Furthermore, put the sub-pixel position of the maximum into
       (px[l],py[l]).
   The parameter mode determines whether maxima (dark lines
       points) or minima
   (bright line points) should be selected.  The partial
      derivatives of the
   image are input as ku[]. */
__global__ static void compute_line_points(float * ku0, float *
    ku1, float * ku2, float * ku3, float * ku4, byte * ismax,
   float * ev, float * nx, float * ny, float * px, float * py,
   long width, long height, float low, float high, long mode)
{
  long   r, c, l;
  float  k0, k1, k2, k3, k4;
  float2  eigval;
  float4  eigvec;
  float  a, b, t;
  long   num;
  float  n1, n2;
  float  p1, p2;
  float  val;
  c = blockIdx.x * blockDim.x + threadIdx.x;
  if (c >= width) return;
  for (r=0; r<height; r++) {
      l = LINCOOR(r,c,width);
      k0 = ku0[l];
      k1 = ku1[l];
      k2 = ku2[l];
      k3 = ku3[l];
      k4 = ku4[l];
      ev[l] = 0.0f;
      nx[l] = 0.0f;
      ny[l] = 0.0f;
      compute_eigenvals(k2,k3,k4,&eigval,&eigvec);
      if (mode == MODE_LIGHT)
        val = -eigval.x;
      else
        val = eigval.x;
      if (val > 0.0f) {
        ev[l] = val;
        n1 = eigvec.x;
        n2 = eigvec.y;
        a = k2*n1*n1+2.0f*k3*n1*n2+k4*n2*n2;
        b = k0*n1+k1*n2;
        solve_linear(a,b,&t,&num);
        if (num != 0) {
          p1 = t*n1;
          p2 = t*n2;
          if (fabs(p1) <= PIXEL_BOUNDARY && fabs(p2) <=
              PIXEL_BOUNDARY) {
            if (val >= low) {
              if (val >= high)
                ismax[l] = 2;
              else
                ismax[l] = 1;
            }
            nx[l] = n1;
            ny[l] = n2;
            px[l] = r+p1;
            py[l] = c+p2;
          }
        }
      }
  }
}



extern "C" {
static float * k[5], * kDevice[5];
static byte *ismax, *ismaxDevice;
static float *ev, *n1, *n2, *p1, *p2;
static float *evDevice, *n1Device, *n2Device, *p1Device, *
   p2Device;
static int *num_resultDevice;

void initCuda_position(long width, long height) {
  long i, size = width * height * sizeof(float);
  for (i = 0; i < 5; ++i) {
    k[i] = (float *) malloc(size);
    cudaMalloc((void **) &kDevice[i], size);
  }
  cudaMalloc((void **) &num_resultDevice, sizeof(int));
  ismax = (byte *) xcalloc(width*height,sizeof(*ismax));
  ev = (float *) xcalloc(width*height,sizeof(*ev));
  n1 = (float *) xcalloc(width*height,sizeof(*n1));
  n2 = (float *) xcalloc(width*height,sizeof(*n2));
  p1 = (float *) xcalloc(width*height,sizeof(*p1));
  p2 = (float *) xcalloc(width*height,sizeof(*p2));
  cudaMalloc((void **) &ismaxDevice, width*height*sizeof(*ismax
     ));
  cudaMalloc((void **) &evDevice, width*height*sizeof(*evDevice
     ));
  cudaMalloc((void **) &n1Device, width*height*sizeof(*n1Device
     ));
  cudaMalloc((void **) &n2Device, width*height*sizeof(*n2Device
     ));
  cudaMalloc((void **) &p1Device, width*height*sizeof(*p1Device
     ));
  cudaMalloc((void **) &p2Device, width*height*sizeof(*p2Device
     ));
}



void cleanupCuda_position() {
  long i;
  cudaFree(p2Device);
  cudaFree(p1Device);
  cudaFree(n2Device);
  cudaFree(n1Device);
  cudaFree(evDevice);
  cudaFree(ismaxDevice);
  free(p2);
  free(p1);
  free(n2);
  free(n1);
  free(ev);
  free(ismax);
  cudaFree(num_resultDevice);
  for (i = 0; i < 5; ++i)
    cudaFree(kDevice[i]);
}



// Computes a gradient field based on the first derivatives.
    The result overwrites the x derivatives.
__global__ void compute_gradient(float *dx, float *dy, long
    width, long height) {
  float x, y;
  long l, r, c = blockDim.x * blockIdx.x + threadIdx.x;
  if (c >= width)
    return;
  for (r=0; r<height; r++) {
    l = LINCOOR(r,c,width);
    x = dx[l];
    y = dy[l];
    dx[l] = sqrt(x*x+y*y);
  }
}
```

```
/* Main routine to detect lines in an image of dimension width
    * height.  The
   extracted lines are returned in result, while num_result is
      the number of
   detected lines.  The parameter sigma is the amount of
      smoothing that the
   Gaussian kernel performs, while low and high are the
      hysteresis thresholds
   used in the linking algorithm.  With mode, either bright or
      dark lines can
   be selected.  The parameter compute_width determines whether
       the line width
   should be extracted, while correct_pos determines whether
      the line width
   and position correction should be applied. */
void detect_lines (byte *image, long width, long height,
   contour ***result, long *num_result, double sigma, double
   low, double high, long mode, bool compute_width, bool
   correct_pos, bool extend_lines)
{
  long   j, i;
  float  *floatImage;
  long floatSize = width * height * sizeof(float);
  cputimer_t timer;
  long numBlocks;

  floatImage = (float *) xcalloc(width * height, sizeof(float))
     ;
  for (i = 0; i < width * height; ++i)
    floatImage[i] = image[i];

//  printTimeAndReset(&timer, "alloc");

  startTimer(&timer);
  convolve_gauss(floatImage,kDevice[0],width,height,sigma,
     DERIV_R);
  convolve_gauss(floatImage,kDevice[1],width,height,sigma,
     DERIV_C);
  convolve_gauss(floatImage,kDevice[2],width,height,sigma,
     DERIV_RR);
  convolve_gauss(floatImage,kDevice[3],width,height,sigma,
     DERIV_RC);
  convolve_gauss(floatImage,kDevice[4],width,height,sigma,
     DERIV_CC);
  printTimeAndReset(&timer, "convolve");
```

```
cudaMemset(ismaxDevice, 0, width*height*sizeof(*ismaxDevice))
    ;
cudaMemset(p1Device, 0, floatSize);
cudaMemset(p2Device, 0, floatSize);

printTimeAndReset(&timer, "memcpy to");

cudaThreadSynchronize();
checkCUDAError("kernel invocation a");

numBlocks = (long)(ceil(width / 32.0f));
compute_line_points<<<numBlocks, 32>>>(kDevice[0],kDevice[1],
    kDevice[2],kDevice[3],kDevice[4],ismaxDevice,evDevice,
    n1Device,n2Device,p1Device,p2Device,width,height,low,high,
    mode);

cudaThreadSynchronize();
checkCUDAError("kernel invocation b");

printTimeAndReset(&timer, "line points");

for (i = 0; i < 2; ++i) { // Note: only the first two are
    used by cpu later
  cudaMemcpy(k[i], kDevice[i], width * height * sizeof(float)
      , cudaMemcpyDeviceToHost);
}
cudaMemcpy(num_result, num_resultDevice, sizeof(int),
    cudaMemcpyDeviceToHost);
cudaMemcpy(ismax, ismaxDevice, width * height * sizeof(byte),
     cudaMemcpyDeviceToHost);
cudaMemcpy(ev, evDevice, width * height * sizeof(float),
    cudaMemcpyDeviceToHost);
cudaMemcpy(n1, n1Device, width * height * sizeof(float),
    cudaMemcpyDeviceToHost);
cudaMemcpy(n2, n2Device, width * height * sizeof(float),
    cudaMemcpyDeviceToHost);
cudaMemcpy(p1, p1Device, width * height * sizeof(float),
    cudaMemcpyDeviceToHost);
cudaMemcpy(p2, p2Device, width * height * sizeof(float),
    cudaMemcpyDeviceToHost);

cudaThreadSynchronize();
checkCUDAError("kernel invocation c");//todo

printTimeAndReset(&timer, "memcpy from");
```

```
compute_contours(ismax,ev,n1,n2,p1,p2,k[0],k[1],result,
    num_result,sigma,
                 extend_lines,mode,low,high,width,height);

printTimeAndReset(&timer, "contours");

if (compute_width) {
  long total_points = 0;
  long * offsets, * offsets_device;
  float * width_r, * width_l, * asymm, * contrast;
  float * width_r_device, * width_l_device, * grad_r_device,
      * grad_l_device, * correct_device, * asymm_device, *
      contrast_device;
  long * cont_num, * cont_num_device;
  contour_class * cont_class, * cont_class_device;
  float * cont_row, * cont_col, * cont_angle, * cont_response
      ;
  float * cont_row_device, * cont_col_device, *
      cont_angle_device, * cont_response_device;
  contour * cont;

  compute_gradient<<<numBlocks, 32>>>(kDevice[0], kDevice[1],
       width, height); // The result goes into kDevice[0]; we
      no longer need the derivatives

  // Count the total number of points, and copy each contour'
      s point count and contour class to common arrays
  cont_num      = (long*)           xcalloc(*num_result,
      sizeof(long));
  cont_class    = (contour_class*)  xcalloc(*num_result,
      sizeof(contour_class));
  for (i = 0; i < *num_result; ++i) {
    total_points += (*result)[i]->num;
    cont_num[i] = (*result)[i]->num;
    cont_class[i] = (*result)[i]->cont_class;
  }

  // Generate offsets into the common arrays
  offsets = (long*)xcalloc(*num_result, sizeof(long));
  offsets[0] = 0;
  for (i = 1; i < *num_result; ++i) {
    offsets[i] = offsets[i - 1] + (*result)[i - 1]->num;
  }

  // Copy the arrays for each contour
  cont_row      = (float*)xcalloc(total_points, sizeof(float)
      );
```

```
cont_col      = (float*)xcalloc(total_points, sizeof(float)
   );
cont_angle    = (float*)xcalloc(total_points, sizeof(float)
   );
cont_response = (float*)xcalloc(total_points, sizeof(float)
   );
for (i = 0; i < *num_result; ++i) {
  cont = (*result)[i];
  memcpy(cont_row + offsets[i], cont->row, cont->num *
     sizeof(float));
  memcpy(cont_col + offsets[i], cont->col, cont->num *
     sizeof(float));
  memcpy(cont_angle + offsets[i], cont->angle, cont->num *
     sizeof(float));
  memcpy(cont_response + offsets[i], cont->response, cont->
     num * sizeof(float));
}


cudaThreadSynchronize();
checkCUDAError("gradient");

cudaMalloc((void**)&offsets_device, *num_result * sizeof(
   long));
cudaMalloc((void**)&cont_num_device,      *num_result *
   sizeof(long));
cudaMalloc((void**)&cont_class_device,    *num_result *
   sizeof(contour_class));
cudaMalloc((void**)&cont_row_device,      total_points *
   sizeof(float));
cudaMalloc((void**)&cont_col_device,      total_points *
   sizeof(float));
cudaMalloc((void**)&cont_angle_device,    total_points *
   sizeof(float));
cudaMalloc((void**)&cont_response_device, total_points *
   sizeof(float));
cudaMalloc((void**)&width_l_device,       total_points *
   sizeof(float));
cudaMalloc((void**)&width_r_device,       total_points *
   sizeof(float));
cudaMalloc((void**)&grad_l_device,        total_points *
   sizeof(float));
cudaMalloc((void**)&grad_r_device,        total_points *
   sizeof(float));
cudaMalloc((void**)&correct_device,       total_points *
   sizeof(float));
```

```
cudaMalloc((void**)&contrast_device,      total_points *
   sizeof(float));
cudaMalloc((void**)&asymm_device,         total_points *
   sizeof(float));

cudaMemset(width_l_device,  0, total_points * sizeof(float)
   );
cudaMemset(width_r_device,  0, total_points * sizeof(float)
   );
cudaMemset(grad_l_device,   0, total_points * sizeof(float)
   );
cudaMemset(grad_r_device,   0, total_points * sizeof(float)
   );
cudaMemset(correct_device,  0, total_points * sizeof(float)
   );
cudaMemset(contrast_device, 0, total_points * sizeof(float)
   );
cudaMemset(asymm_device,    0, total_points * sizeof(float)
   );

cudaMemcpy(offsets_device, offsets, *num_result * sizeof(
   long), cudaMemcpyHostToDevice);
cudaMemcpy(cont_num_device, cont_num, *num_result * sizeof(
   long), cudaMemcpyHostToDevice);
cudaMemcpy(cont_class_device, cont_class, *num_result *
   sizeof(contour_class), cudaMemcpyHostToDevice);
cudaMemcpy(cont_row_device, cont_row, total_points * sizeof
   (float), cudaMemcpyHostToDevice);
cudaMemcpy(cont_col_device, cont_col, total_points * sizeof
   (float), cudaMemcpyHostToDevice);
cudaMemcpy(cont_angle_device, cont_angle, total_points *
   sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(cont_response_device, cont_response,
   total_points * sizeof(float), cudaMemcpyHostToDevice);

printTimeAndReset(&timer, "memcpy etc");

compute_line_width<<<*num_result, 1, (long)ceil(
   MAX_LINE_WIDTH*3) * sizeof(offset)>>>(kDevice[0], width,
    height, sigma, mode, correct_pos,
    offsets_device, cont_num_device, cont_row_device,
       cont_col_device, cont_angle_device,
    width_r_device, width_l_device, grad_r_device,
       grad_l_device);
cudaThreadSynchronize();
printTimeAndReset(&timer, "width");
```

```
fix_locations<<<*num_result, 1>>>(offsets_device,
    width_l_device,width_r_device,grad_l_device,
    grad_r_device,correct_device,contrast_device,
    asymm_device,
     sigma,mode,correct_pos,cont_num_device,
        cont_class_device,cont_angle_device,
        cont_response_device, cont_row_device,
        cont_col_device);
cudaThreadSynchronize();
printTimeAndReset(&timer, "bias");

width_l = (float*)xcalloc(total_points,sizeof(*width_l));
width_r = (float*)xcalloc(total_points,sizeof(*width_r));
contrast = (float*)xcalloc(total_points,sizeof(*contrast));
asymm = (float*)xcalloc(total_points,sizeof(*asymm));
cudaThreadSynchronize();
checkCUDAError("line width");

cudaMemcpy(width_l, width_l_device, total_points * sizeof(
    float), cudaMemcpyDeviceToHost);
cudaMemcpy(width_r, width_r_device, total_points * sizeof(
    float), cudaMemcpyDeviceToHost);
cudaMemcpy(cont_row, cont_row_device, total_points * sizeof
    (float), cudaMemcpyDeviceToHost);
cudaMemcpy(cont_col, cont_col_device, total_points * sizeof
    (float), cudaMemcpyDeviceToHost);
cudaMemcpy(asymm, asymm_device, total_points * sizeof(float
    ), cudaMemcpyDeviceToHost);
cudaMemcpy(contrast, contrast_device, total_points * sizeof
    (float), cudaMemcpyDeviceToHost);

for (i = 0; i < *num_result; ++i) {
  cont = (*result)[i];
  cont->width_l = (float*)xcalloc(cont->num, sizeof(float))
      ;
  cont->width_r = (float*)xcalloc(cont->num, sizeof(float))
      ;
  memcpy(cont->width_l, width_l + offsets[i], cont->num *
      sizeof(float));
  memcpy(cont->width_r, width_r + offsets[i], cont->num *
      sizeof(float));
  memcpy(cont->row, cont_row + offsets[i], cont->num *
      sizeof(float));
  memcpy(cont->col, cont_col + offsets[i], cont->num *
      sizeof(float));
  if (correct_pos) {
        cont->asymmetry = (float*)xcalloc(cont->num, sizeof(
           float));
        cont->contrast = (float*)xcalloc(cont->num, sizeof(
           float));
        memcpy(cont->asymmetry, asymm + offsets[i], cont->num *
            sizeof(float));
        if (mode == MODE_LIGHT) {
          memcpy(cont->contrast, contrast + offsets[i], cont->
              num * sizeof(float));
        }
        else {
          for (j = 0; j < cont->num; ++j) {
            cont->contrast[j] = -contrast[offsets[i] + j];
          }
        }
      }
    }
  }
}
}
```

Appendix B

CD-ROM

The attached CD-ROM contains the full source code of Steger's original implementation (in the `steger` folder) and of our CUDA and OpenMP versions (in the `cuda` and `openmp` folders, respectively). It also contains the not entirely completed code for steps 4 and 5, in the folder `width` — this version works for single images, but the code is untidy and lacks proper memory deallocation. The `cuda` folder contains the "official" version of the code. The test images are located in the `images` folder.

The contents of the CD-ROM can also be downloaded from `http://folk.ntnu.no/asmunde/cd.rar`.

Our CUDA and OpenMP applications are built by invoking `make` (the configure scripts should not be run; we have not had the time to adapt them to our version — so the code might only run under systems similar to ours). The `ffmpeg` library is required for the CUDA application because of the video player. See `large.sh` for an example of how to run the application. If a CUDA device other than device 0 is to be used, the switch `--device 1` can be used when invoking `detect-lines`. The `large.sh` script will use the values of the environment variable `$CUDA_DEVICE_PARAMETER` as extra switches.

For instructions on running Steger's implementation, see the `README` file.