Active Contours without Edges on the GPU

Morten Bojsen-Hansen
20053496
mortenbh@cs.au.dk

***Abstract*—Image segmentation is the process of partitioning an input image into regions with certain common visual characteristics. It has numerous applications ranging from biometrics to weapon systems to medical imaging, which is the primary focus of this paper. Specifically we will look at a model by Chan-Vese that detects objects not defined by gradient. We will then implement this algorithm on the GPU and see what kind of speedup we can get compared to serial and multi-threaded CPU implementations. Finally we will quantity our results as well as make a qualitative evaluation of the method with respect to how it performs for segmenting medical images.**

***Index Terms*—Chan-Vese, snakes, active contours, image segmentation, medical imaging level set method, energy functional, finite differences, CUDA, OpenMP, Thrust.**

This project was developed in association with
Dung My Hoa and Finn Jepsen Schmidt.

The application we developed in this project is available as a Windows executable and as source code at http://daimi.au.dk/~mortenbh/ChanVese/. Here you will also find a README describing how to use the application.

I. Introduction

SEGMENTATION is the process of partitioning a data set into disjoint sets or segments of elements with common cohesive properties. The idea is obtain a simplified or somehow altered representation of the input data set, which is easier to analyze and work with in general. *Image segmentation* is the process of locating and quantifying objects in an input image. The common cohesive properties of the segments mentioned above are typically certain visual characteristics. In other words we seek to label each image element (*e.g.* pixel, voxel) such that it belongs to a set of similarly labelled image elements with common visual characteristics.

Image segmentation has numerous real-world applications. Any form of object recognition such as fingerprint recognition, face recognition, locating objects (roads, mines etc.) in satellite images uses image segmentation. Machine vision is another dicipline where image segmentation is used. Specifically it is used to reject defective components coming out of an assembly line instead of needing visual inspection by a human being. In this paper we will focus on image segmentation as applied to medical imaging. Applications herein include, but are not limited to measuring tissue volumes, the study of anatomical structure, diagnostics and treatment planning.

Manual segmentation, which consists of manually drawing contours on printed slices or drawing pixels on a computer, is still the standard in fields such as oncology. This process is time-consuming, especially if contours need to be drawn on many slices, so anything that can be done to shorten the time spent by specialists here is desired, since it frees up their time for more important tasks. Even in fields where computer-assisted image segmentation is being used though, the process can be time-consuming due to the huge computational burden imposed by large data sets. Ideally we want near real-time segmentation, so we can track objects live, as the images are being retrieved from *e.g.* CT or MRI scanners

The main way to parallelize an application has usually been what is known as *task parallism*, where an application is manually partitioned into tasks, which can then be run in parallel to reduce execution time. Unfortunately this manual partioning is both cumbersome, error-prone and time-consuming. Worse yet, it doesn't scale. Most applications don't have eight or even four tasks that can run concurrently at all times. With the gigahertz wars finally over, transister count and thus processing power will increase mostly as a result of more processing cores, at least for the foreseeable future. This means programmers can no longer rely on new hardware to make their applications faster if they use task parallelism, since the number of cores will soon be greater than the number of tasks.

An attractive alternative to task parallelism is *data parallelism*. Here we parallelize at a lower level of granularity, namely at the level of data. If we need to process a data set, we may be able to process different parts of the data set concurrently. As long as the data set is sufficiently large, this form of parallelism scales much better.

Still, the number of cores is relatively small in practice. At the time of this writing Intel's top-end CPU had only eight cores and more mainstream CPUs have a maximum of four cores. This means that even if our application scales perfectly with data parallelism, we can only expect a four-fold speedup through parallelization on the CPU in practice.

Since about 2000, general purpose programming on the graphical processing unit (GPGPU) has been a hot research topic. GPU manufactorers have realized this, and have accommodated the demand by removing restrictions on the type of computations possible, greatly increasing the compute power and making toolkits that significantly ease the development of parallel algorithms on the GPU. A few such toolkits are Brook[1], CUDA[2] and OpenCL[3].

GPUs are obviously primarily designed for processing and rendering 3D graphics. This, however, turns out to be an inherently data parallel problem. With the advent of GPGPU, this means that most highly data parallel algorithms will benefit

[1]BrookGPU – www.graphics.stanford.edu/projects/brookgpu/
[2]NVIDIA Compute Unified Device Architecture – www.nvidia.com/object/cuda_home.html
[3]OpenCL – www.khronos.org/opencl/

from being implemented on the GPU over a traditional multi-threaded CPU implementation. This is true since GPUs can run several orders og magnitude more threads concurrently than your typical CPU. As we willl show, some image segmentation schemes also benefit greatly from being implemented on the GPU.

Autonomous image segmentation can be done in a variety of ways. One of the most widespread methods is the idea of so-called active contours – curves that evolve subject to constraints given by the input image to find objects in the image. The use of active contours for image segmentation was made popular by Kass, Witkin and Terzopoulos [1] in their seminal paper about the snake model. Their model evolved a spline by minimizing the energy functional:

$$E^{\star}_{snake} = \alpha \int_0^1 \left|C'(s)\right|^2 ds + \beta \int_0^1 \left|C''(s)\right| ds - \lambda \int_0^1 \left|\nabla u_0 C(s)\right| ds$$

Here, α, β and λ are positive parameters. $u_0 : \bar{\Omega} \to \mathbb{R}$ is our image, where Ω denotes a bounded open subset of $\mathbb{R}^2$ with boundary $\partial\Omega$. Finally $C(s) : [0,1] \to \mathbb{R}^2$ is our parameterized curve. Notice how the first two terms keeps the curve smooth, since it's penalized for having non-zero first and second-order derivatives. Also notice that the third term attracts the curve towards objects, since ∇u_0 works like an edge-detector: large gradient magnitudes imply edges in u_0. Since we don't want the snakes to stop due to small-scale noise, we usually convolve u_0 with a Gaussian function before applying the gradient.

One big problem with active contours models that use the gradient as an edge detector, such as the snake model mentioned above, is that they can only detect objects with edges defined by gradient. In practice, the gradient is bounded and thus the stopping function is never zero at the edges, which means the snake may not stop at the edges. In noisy images, we have to smooth u_0 significantly, which means the edges are smoothed as well. With regards to parallelism, the snake model is poor too. Since we use a spline to represent the contour, we only have a small number of points to evolve, which won't scale well on e.g. the GPU.

In this paper we will introduce an active contours model without edges by Chan and Vese [2]. This method overcomes the above-mentioned problems by locating objects, not by gradient, but by finding a boundary in the image that best approximates the image by two piecewise-constant areas. This model is also much better with respect to parallelism on the GPU, since the constraints in this model depend on the entire input image.

Our contributions are as follows:

- An implementation on the GPU using the CUDA toolkit of the model proposed by Chan and Vese in their Active Contours Without Edges [2] paper. We also include a technical description and discussion of how we made this implementation to achieve maximum performance.
- A multi-threaded reference implementation on the CPU for comparison with the GPU implementation.
- A quantitative comparison of the GPU and CPU implementations, primarily with respect to performance. Also a qualitative evalutation of our Chan-Vese implementation for segmentation of medical images.

From hereon the paper will be structured as follows. In section II we will briefly gloss over some related work. In section III we describe and motive the model by Chan-Vese, give a level set formulation of the model and discretisize it. In section IV we describe and discuss our GPU implementation of the model. In section V we describe the methods we have used to characterize our implementation and the model, as well as present our results based on these methods. In section VI we discuss the results we obtained in the previous section. Finally we conclude and talk about future work in section VII.

II. Related Work

In this section we will briefly mention some related work, but not go into details.

As we shall see later, one of the limitations of the Chan-Vese model proposed in [2] is that it can only partition an image into two piecewise-constant regions. The same authors later devised a method that could partition an image into a number of piecewise-constant regions in [3]. The method is a generalization of their previously published single-phase method to a multi-phase method. Futhermore the method is quite space-efficient, since it only needs $\log n$ level set functions for n phases. Needless to say, this generalization makes the method much more useful than the original two-segment method.

A completely different technique is the 'balloon method' of Bowden [4]. While this model is aimed at volumetric data sets it can certainly be limited to the two-dimensional case also. The idea behind the model is that of small simulated 'balloons' that inflate to evolve a curve to it's maximum capacity within the volume. The idea is based partly on the original snakes model.

III. Chan-Vese Model

In this section we will describe and motivate the model by Chan-Vese [2]. We will describe a level set formulation of the method and show how to discretisize the model to obtain an algorithm that evolves a curve in time.

As in the case of the snake model described in section I, we consider an open subset Ω of $\mathbb{R}^2$. We will describe the evolving curve as an open subset of $\omega \subset \Omega$ with boundary $\partial\omega$. To make thing clearer, we let $inside(C)$ denote the region ω and $outsite(C)$ denote the region $\Omega \backslash \bar{\omega}$. In other words $inside(C)$ and $outside(C)$ are two disjoint regions that partition Ω with C being the boundary between these two regions.

The main idea of the model is the assumption that the image u_0 can be partitioned into two regions of approximately piecewise-constant intensities. Consider now the energy func-

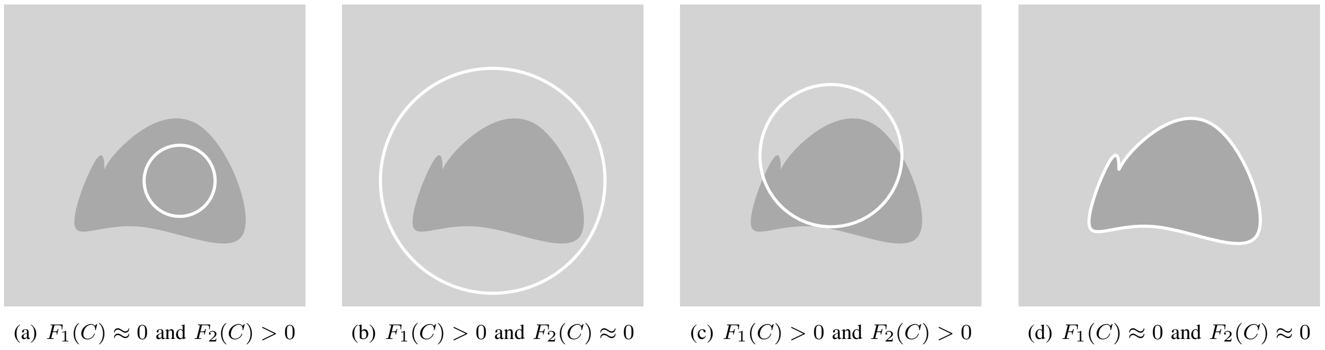

(a) $F_1(C) \approx 0$ and $F_2(C) > 0$ (b) $F_1(C) > 0$ and $F_2(C) \approx 0$ (c) $F_1(C) > 0$ and $F_2(C) > 0$ (d) $F_1(C) \approx 0$ and $F_2(C) \approx 0$

Fig. 1. Illustrating the terms $F_1(C)$ and $F_2(C)$ of the energy functional $E_{fitting}(C)$ with a simple example.

tional

$$\begin{aligned}E_{fitting}(C) &= F_1(C) + F_2(C)\\ &= \int_{inside(C)} |u_0(x,y) - c_1|^2 \, dx\, dy\\ &\quad + \int_{outside(C)} |u_0(x,y) - c_2|^2 \, dx\, dy\end{aligned}$$

where c_1 and c_2 denote the average intensity of $inside(C)$ and $outside(C)$ respectively. It is easy to see that boundary of the object in u_0 minimizes $E_{fitting}(C)$. Consider the example in figure 1. If C is inside the object then $F_1(C) \approx 0$ and $F_2(C) > 0$. Similarly if C is outside the object we have $F_1(C) > 0$ while $F_2(C) \approx 0$. If C is both inside and outside the object both F_1 and F_2 are above 0, and finally, if C is at the boundary of the object, $E_{fitting}(C)$ is minimized.

In the final model we add two additional regularization terms to obtain the energy functional $F(c_1, c_2, C)$:

$$\begin{aligned}F(c_1, c_2, C) &= \mu \cdot \text{Length}(C) + \nu \cdot \text{Area}(inside(C))\\ &\quad + \lambda_1 \int_{inside(C)} |u_0(x,y) - c_1|^2 \, dx\, dy\\ &\quad + \lambda_2 \int_{outside(C)} |u_0(x,y) - c_2|^2 \, dx\, dy\end{aligned}$$

where $\mu, \nu \geq 0$ and $\lambda_1, \lambda_2 > 0$ are user-defined parameters. The regularization terms punish contours that are either too long or have too large of an internal area. Obviously we cannot pick μ and ν too large, or the contour won't be able to expand.

A. Level Set Formulation

The *level set method* by Osher and Sethian [5] is a popular way of tracking an interface or curve and has been used in many forms of image segmentation too. The idea is to implicitly embed the curve in a space one dimension higher than that of it's boundary. Thus a two-dimensional curve is represented in three dimensions as a function where each point represents the distance to the closest point on the curve. The curve then becomes the zero level set of the three-dimensional function, hence the name *level set*. Additionally the sign of the function tells us whether we are inside or outside the curve. We adopt the convention that we have positive values inside the curve and negative outside.

Chan and Vese also formulated a level set formulation of their model. Thusly they represent C as the zero level set of a Lipschitz function $\phi : \Omega \to \mathbb{R}$. If we let

$$H(z) = \begin{cases} 1, & \text{if } z \geq 0\\ 0, & \text{if } z < 0,\end{cases}$$

and $\delta(z) = \frac{d}{dz} H(z)$, we can define the energy functional as

$$\begin{aligned}F(c_1, c_2, \phi) &= \mu \int_\Omega \delta\left(\phi\left(x,y\right)\right) \left|\nabla \phi\left(x,y\right)\right| \, dx\, dy\\ &\quad + \nu \int_\Omega H\left(\phi\left(x,y\right)\right) \, dx\, dy\\ &\quad + \lambda_1 \int_\Omega \left|u_0\left(x,y\right) - c_1\right|^2 H\left(\phi\left(x,y\right)\right) \, dx\, dy\\ &\quad + \lambda_2 \int_\Omega \left|u_0\left(x,y\right) - c_2\right|^2 \left(1 - H\left(\phi\left(x,y\right)\right)\right) \, dx\, dy\end{aligned}$$

where

$$c_1 = \frac{\int_\Omega u_0(x,y) H(\phi(x,y)) \, dx\, dy}{\int_\Omega H(\phi(x,y)) \, dx\, dy} \tag{1}$$

$$c_2 = \frac{\int_\Omega u_0(x,y)(1 - H(\phi(x,y))) \, dx\, dy}{\int_\Omega (1 - H(\phi(x,y))) \, dx\, dy} \tag{2}$$

In order to compute the Euler-Lagrange equation for the unknown function ϕ, we have to replace H and δ with regularized versions. We denote the regularized versions as H_ε and $\delta_\varepsilon = H'_\varepsilon$. By introducing the artificial time variable t, we finally get the following PDE for evolving ϕ in time:

$$\begin{aligned}\frac{\partial \phi}{\partial t} &= \delta_\varepsilon(\phi) \left[\mu \nabla \cdot \frac{\nabla \phi}{|\nabla \phi|} - \nu - \lambda_1 \left(u_0 - c_1\right)^2 + \lambda_2 \left(u_0 - c_2\right)^2\right]\\ &= 0 \text{ in } \Omega\end{aligned} \tag{3}$$

with boundary condition

$$\frac{\delta_\varepsilon(\phi)}{|\nabla \phi|} \frac{\partial \phi}{\partial \vec{n}} = 0 \text{ on } \partial\Omega.$$

B. Numerical Approximation

To be able to actually solve equation (3) numerically, we need to discretisize it. We do this on a regular two-dimensional M-by-N grid. We let h denote the spacing between cells, Δt denote the time step and $(x_i, y_j) = (ih, jh)$ be the grid points with $0 \leq i < M$ and $0 \leq j < N$.

The original paper proposes a forward Euler implicit scheme. This scheme has the advantage that it is unconditionally stable, meaning we can choose Δt arbitrarily large without fear of the simulation exploding. It does, however, require that we solve a system of linear equations. This can be done iteratively with simple Jacobi iteration or more complicated schemes such as the preconditioned conjugate gradient method. Both methods have been implemented with some success on the GPU, but are still relatively slow. We will instead use the faster explicit forward Euler scheme. This does mean we will lose our unconditional stability, but since we don't have any linear systems to solve, this method is much faster and also simpler. We will deal with the stability in a later section.

Discretisizing c_1 and c_2 is easy, so long as we fix a regularized Heaviside function. For our discretization we have chosen

$$H(z) = \begin{cases} 1, & \text{if } z > \varepsilon \\ 0, & \text{if } z < -\varepsilon \\ \frac{1}{2} + \frac{z}{2\varepsilon} + \frac{1}{2\pi}\sin\left(\frac{\pi z}{\varepsilon}\right) & \text{if } |z| \le \varepsilon. \end{cases}$$

The only thing a bit difficult to discretisize is $\nabla \cdot \frac{\nabla\phi}{|\nabla\phi|}$ – that is, the divergence of the normalized gradient of ϕ. If we let ϕ_x, ϕ_y denote the partial derivatives of ϕ and similarly let ϕ_{xx}, ϕ_{yy} and ϕ_{xy} denote the second-order partial derivatives, we obtain (after some tedious calculations):

$$\nabla \cdot \frac{\nabla\phi}{|\nabla\phi|} = \frac{\phi_{xx}\phi_y^2 - 2\phi_{xy}\phi_x\phi_y + \phi_{yy}\phi_x^2}{(\phi_x + \phi_y)^{3/2}},$$

which we discretisize with simple finite differences:

$$\begin{aligned} \phi_x(i,j) &= (x_{i-1}, x_j) - (x_{i+1}, x_j) \\ \phi_y(i,j) &= (x_i, x_{j-1}) - (x_i, x_{j+1}) \\ \phi_{xx}(i,j) &= (x_{i-1}, x_j) - (x_{i+1}, x_j) + 2(x_i, x_j) \\ \phi_{yy}(i,j) &= (x_i, x_{j-1}) - (x_i, x_{j+1}) + 2(x_i, x_j) \\ \phi_{xy}(i,j) &= \frac{1}{4}\left((x_{i+1}, x_{j+1}) + (x_{i-1}, x_{j+1})\right. \\ &\quad \left. + (x_{i+1}, x_{j-1}) + (x_{i-1}, x_{j-1})\right). \end{aligned}$$

Now we have all the ingredients for an algorithm that iteratively evolves ϕ. We start by initializing ϕ with some initial contour and set $n = 0$. We have chosen a perfect circle, since it's easy to create ϕ this way. We then compute $c_1(\phi^n)$ and $c_2(\phi^n)$ from equation (1) and equation (2). Next we compute equation (3) with the discretization above to obtain $\phi^{n+1} = \phi^n + \Delta t \cdot \frac{\partial\phi}{\partial t}$ with explicit forward Euler. Finally we update $n = n + 1$ and repeat.

IV. GPU Implementation

In this section we will describe how we implemented a GPU version of the model described in section III. We will also describe and discuss the choices we made in the implementation.

As mentioned in section I, we have implemented the Chan-Vese model of section III using the NVIDIA CUDA toolkit. This toolkit allows programmers to program the GPU through the C programming language with a few simple extensions. Of course you need to take care to design algorithms so they make full use of the special way GPUs are designed. There are two main differences between programming for the GPU and the CPU. First of all, the GPU is designed for maximum memory throughput at the cost of latency. Second, the GPU is optimized for data processing, and hence has very few transistors dedicated to flow control as opposed to the CPU.

What the above means is that you should generally avoid threads that diverge due to if-statements and similar, since these will significantly impact the speedup you will be able to get by parallelizing your algorithms on the GPU. Second, you should generally send large data sets to the GPU so it's able to effectively hide the relatively large latency associated with computation on the GPU.

We start with an input image and a initial contour ϕ, both discretized on equally sized 2D grids. The image is loaded from file using the Nokia Qt library[4]. Our initial contour is, as mentioned, a perfect circle, also generated on the CPU. After this, we copy both the image and ϕ to the GPU. Here we use what CUDA calls a *pitched array*, which means an array where each row is padded for maximum memory read and write performance. Note, in reality it is still only a one-dimensional array, but it's padded for 2D memory access patterns. From this point on, we don't need to copy any more data to the GPU, since it has everything it needs, and thus we have effectively optimized memory transfers between the CPU and GPU.

After this we start the algorithm mentioned in section III-B. Thus we need to compute the curvature force – that is, the divergence of the normalized gradient. We use use the discretization we obtained previously. Note how we need to compute this force for every point on our grid, and how we can do this completely independently in each grid cell. This is a good time to introduce the term *coalescence*. Since we use finite differences to compute the curvature force in a grid cell, we need to access the value of ϕ from the neighbouring cells. Unfortunately the GPU is not optimized for this sort of memory access pattern, and memory accesses can suffer a significant performance penalty if nothing is done to rectify this. Optimal performance is obtained if threads access memory in sequence, which is definitely not the case here. Unfortunately there is no real way to fix this directly, since we need the values of neighbouring cells. What we can do, however, is use textures to access ϕ, and the GPU will automatically make sure the memory access is coalesced behind the scenes. Textures effectively work as a cache, so even though we access each cell of ϕ many times (since we access eight neighbouring cells in each cell), memory access is optimized. We save the force computed here in a temporary grid the size of ϕ, so we can later combine it to obtain ϕ at the next time step.

Next we need to compute c_1 and c_2. We start by computing the value of c_1 and c_2 in each grid cell, storing it in a temporary array like above. Since this computation is completely coalesced both regarding reads and writes, we don't actually need to use textures, but do so anyway for consistency. Since c_1 and c_2 are scalars, we need to sum

[4]Nokia Qt – http://qt.nokia.com

over the arrays we just computed. The naive way of doing this uses $O(n)$ time for a grid of size n and doesn't utilize parallelization at all. Instead we can use what is known as a reduction, cutting the time spent to $O(\log n)$ time assuming we have an infinite number of threads. Note that the curvature computation in the previous paragraph is effectively $O(1)$ time assuming an infinite number of threads. In a reduction, you want to execute some binary function on all elements in an array. You can do this in a tree-like fashion, where you perform the given operation on different parts of the array concurrently and combine the result at the end. It is not too hard to implement, but it is harder to get right and even harder to implement efficiently. Thus you usually want to use a library to do reductions. Luckily NVIDIA has created an STL-like C++ library called *Thrust*[5] that, among other things, can do reductions efficiently on the GPU. We use this library with the binary function $plus$ to get the sum of the c_1 and c_2 arrays. Similarly we use a reduction to get the area of $inside(C)$ and $outside(C)$. Recall that c_1 and c_2 is the *average* intensity inside and outside the contour respectively, so we need to divide by the area.

Hereafter we use the values for c_1 and c_2 computed above to compute the image forces. By image forces, we mean the terms $\lambda_1 (u_0 - c_1)^2 + \lambda_2 (u_0 - c_2)^2$ from equation (3). This is straight forward and coalesced. Last we need to update ϕ for the next iteration, which is again straight forward now that we have the two grids containing the curvature force and the image force. The final step is to copy the new phi back to the CPU. We don't do this every iteration, since that would dwarf most of the speedup we get by running the algorithm on the GPU. In our concrete implementation, you can actually select how often you want updates of phi to be copied back to the CPU and thus displayed to the user.

We mentioned in section III-B that we deal with the instability of our forward Euler scheme. The way we do this, is by normalizing the forces added to ϕ when we update it for the next iteration. In the strict mathematical sense, this is not correct, but we haven't been able to detect a significant difference in the obtained result. Since the forces are normalized they obviously cannot explode (grow exponentially), so we get our stability back. Now, a normalization is also a reduction. First we need to find the absolute maximum value in our grid. This is a transformation (unary function absolute value) followed by a reduction (binary function maximum), which Thrust handles with ease. Next we need to divide by the found value, which is again a transformation (unary function that divides by the magnitude).

In the beginning we said we use so-called pitched arrays. Furthermore we need to support images that have arbitrary dimensions and thus also images that don't necessarily have dimensions that are a multiple of the *block size*. Therefore we also pad our arrays in the y-direction, so the number of columns too is a multiple of the block size (in the y-direction). We somehow need to take care of this padding in our kernels and in our reductions. It turns out none of our kernels actually *read* outside the actual grid dimensions (and thus in the padding) since we use textures for all reads, and the textures are clamped at the correct grid dimensions. Thus we only need to take care of the padding in the reductions. We have handled this within the Thrust library, by creating special transformations and reductions that discard values obtained from the padded region. Specifically we have created functions `padded_transform_reduce`, `padded_count_if` and so on that replace the equivalent functions provided by Thrust (`thrust::transform_reduce`, `thrust::count_if` etc.). This way we can write our transformations and reductions like always and almost transparently handle the padded arrays – we only need to provide the actual and padded dimensions respectively.

An easier way of handling images of arbitrary dimensions would have been to simply pad or even stretch or cut the input image to size. While we considered this option, we deemed it too ugly and condemned it as the hack that it is.

TABLE I
PERFORMANCE COMPARISON BETWEEN OUR REFERENCE SERIAL CPU IMPLEMENTATION, OUR MULTI-THREADED OPTIMIZED CPU IMPLEMENTATION AND OUR CUDA GPU IMPLEMENTATION.

Test case	Iterations	CPU (serial)	CPU (OpenMP)	GPU
I	2000	8.5s	4.82s	7.96s
II	500	28.33s	16.49s	4.93s
III	500	74.37s	43.63s	8.91s
IV	50	115.75s	68.23s	10.44s

V. Methods and Results

In this section we present the methods we have used to evaluate the model of Chan-Vese as well as the results we obtained from these methods. A discussion of our findings follows in section VI.

Obviously we wanted to test the performance of our GPU implementation of the Chan-Vese model. Thus we needed a CPU implementation to test against as comparison. Ideally we would like both a serial reference CPU implementation and a highly optimized multi-threaded version to see whether the GPU can compete on even terms. *OpenMP*[6] helped us achieve both with relative ease. Once we had a reference CPU implementation up and running in C++, we could easily add full multi-threading support with OpenMP without having to change the actual algorithm. All we had to do was mark which parts of the code could run in parallel, which parts were reductions and which parts needed mutual exclusion. Thus we ended up with an optimized multi-threaded CPU implementation that could be switched off with a simple compile-time flag. This made it really easy for us to compare our serial and multi-threaded CPU implementations with our GPU implementation.

The performance timings we obtained can be seen in table I. We tested with four different medical data sets. They were 131-by-131, 500-by-375, 888-by-495 and 2653-by-2336 pixels respectively. All tests were performed on a 2,0 GHz Intel Core2 Duo T6400 CPU with 2 MB cache and 4 GB DDR2

[5]Thrust – http://code.google.com/p/thrust/

[6]OpenMP – http://openmp.org/

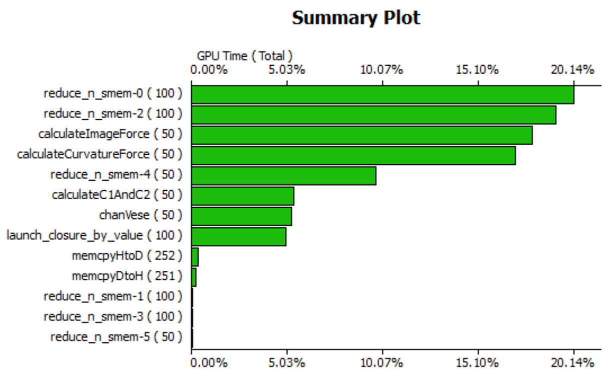


Fig. 2. Profile obtained from `cudaprof` that shows the percentwise time consumption for all kernels. This profile comes from data set IV – that is, the 2653-by-2336 image. Notice that most of the time is spent on the reductions followed by the calculation of image and curvature forces.

RAM. The graphics card was a NVIDIA Geforce 9600M GT. The operation system was Windows Vista SP2 32bit and the compiler was Visual Studio 2008. As can be seen from the table, the multi-threaded version is almost double the speed of the serial version. The GPU version starts out half as fast as the threaded CPU version for data set I, but is faster for all other data sets. For data set II it is over three times faster. For data set III it is almost five times faster, and for data set IV it is over 6.5 times faster. In comparison, the GPU version is over 11 times faster than the serial implementation. Also see figure 2 for a profile of the time-consumption of the different kernels.

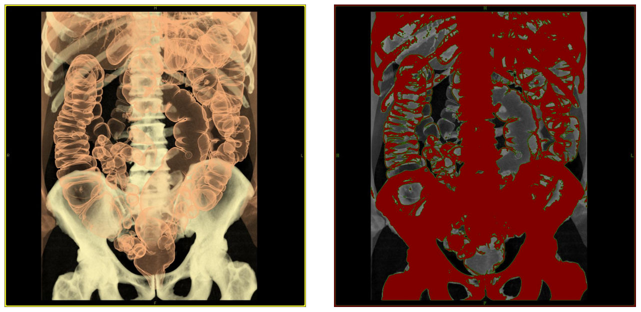
We also wanted to do a qualitative evaluation of the method. Specifically we wanted to see how well or how practical the method was at finding objects in medical images such as CT or MRI scans. To this end we chose three different medical images as can be seen in figure 3, figure 4 and figure 5. In the following we have $\lambda_1 = \lambda_2 = 1$ unless otherwise noted.

In figure 3 we see an image of a brain scan with an initial contour as a perfect circle. We see the contour evolve with the number of iterations to arrive at a pretty satisfactory segmentation of the image. Still we get more than just the brain with this image. In figure 4 we were actually trying to segment the pelvis and spine only – that is, we were trying to segment the bones. Unfortunately we got some of the intestins also. In figure 5 we were trying to segment the bones similarly to the case just mentioned. Unfortunately the segmentation 'bleeds' into the space between the toes. Also we don't get all of the bone, even after this many iterations.

VI. Discussion

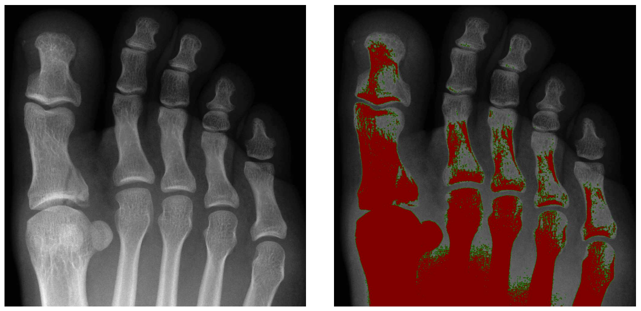
In this section we discuss the results we presented in section V.

If we start by discussing the performance of our different implementations of the Chan-Vese model, we see that the GPU solver beats even the optimized CPU implementations for only moderately-sized data sets. We also notice that size of the data set is the most important factor of how much faster the GPU implementation is compared to the CPU implementations. Since data sets are more than likely to grow in size in the



(a) Original color image (b) Segmentation 2000 iterations

Fig. 4. 1024-by-1024 color image of pelvis and spine. In this segmentation we were looking for the bones only, but got some of the colon as well. Again we have $\mu = 0.2, \Delta t = 5$.



(a) Original four color image (b) Segmentation 2000 iterations

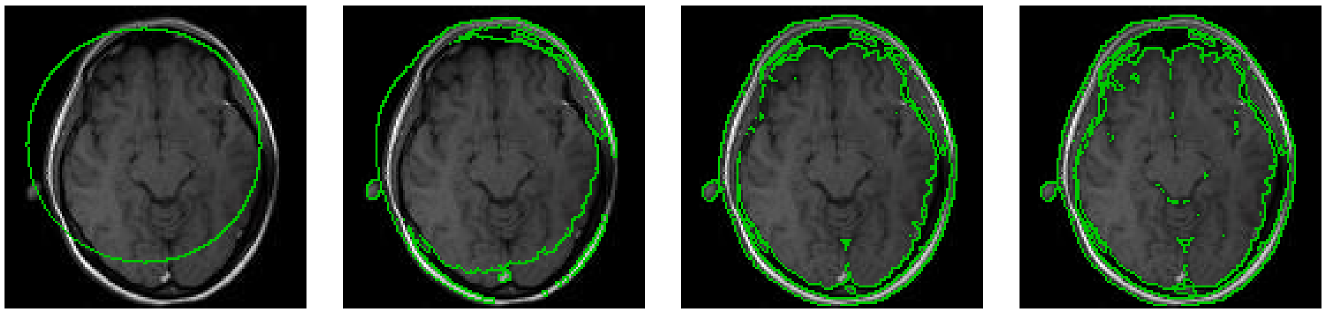
Fig. 5. 1000-by-998 black and white image of the upper part of a foot. The segmentation doesn't stop at the bones, but 'bleeds' out into the space between them. $\mu = 0.2, \Delta t = 5$.

future, we must conclude that a GPU implementation is well worth it. Even if the multi-threaded CPU implementation scales linearly with the number of cores, which it appears to do – at least for two cores, it will be a very long time until the CPU will have enough cores to catch up with the GPU for larger data sets, and by that time the GPU will have become much faster also.

As noted, figure 3 is an example of an image where the Chan-Vese model works well. The image is nicely segmented into a black background, the dark grey brain and the white skull. Since we can only distinguish two segments with the model, we get both the brain and the skull in the segmentation, but that isn't a big problem here.

If we instead look at figure 4, which is a color image, we immediately see that we get too much in the segmentation for it to be really useful. The problem here, is that while the image is in color, the model can only handle single-tone (basically black and white) images, and when the image is converted to grey-scale, we suddenly cannot distinguish bones from intestines any longer. This is clearly a big limitation of the model, which severely limits it's use in medical imaging.

Lastly let us look at figure 5. This is a grey-scale image, so no conversion to grey-scale needs to be done. The segmentation is both incomplete and bleeds, which we noted in section V. This is even worse than the case above, since the segmentation we actually wanted isn't even part of the final segmentation. The number of iterations is also quite high



(a) Intial contour (b) Contour after 50 iterations (c) Contour after 400 iterations (d) Contour after 2000 iterations

Fig. 3. Small 313-by-313 image of a brain. We see the contour evolve with the number of iterations to converge on a pretty good segmentation of the image. Here we picked $\mu = 0.2$ and $\Delta t = 5$. Simulation time was about 8 seconds on the GPU for all 2000 iterations.

compared to how close we are to obtaining the segmentation we want, but even if we continue, the segmentation will most likely just bleed even more, making the whole thing useless. The problem here, is that there are simply too many shades of grey for the model to work satisfactory.

In conclusion, the GPU implementation of the model was well worth it, since it is many times faster than the CPU version, even on the very moderate mobile graphics card we tested with. As for the use of the model for medical imaging, it's use in it's current form is a bit limited, since medical images are seldom 'perfect', and thus we conclude the method is too sensitive.

VII. Conclusion and Future Work

In this paper we have presented the Chan-Vese model for active contours without edges that can be used to segment an image into two regions with piecewise-constant intensities. We also gave a level set formulation of the method as well as derived a PDE to evolve our contour in time. Futhermore we showed how to discretisize this PDE on a regular grid, so the model could easily be implemented on a computer.

We also gave a description of the concrete GPU implementation we have made as well as a discussion of relevant choices we made in the implementation.

We proceeded to talk about our methods and results. Here we benchmarked our GPU implementation against a serial CPU implementation as well as an optimized multi-threaded CPU implementation. The GPU implementation was faster in all but the example with the smallest data set. Additionally the multi-threaded version was almost two times as fast as the serial version – we were running on a dual core CPU.

We also looked at different medical images and performed segmentations on these using our application. Results were mixed. We decided to show three examples in this paper. The first example gave a segmentation that was about as good as you could expect. The second example showed quite a bit of under-segmentation. The third example showed both under-segmentation, but also that the model might not segment the way you want because of too high differences in image intensity.

We concluded that the main problem with the model was that it could only segment into two regions, which was the reason for most under-segmentation. In practice medical images are far from perfect, so we concluded that this simple model needs more work to be truely applicable in medical imaging.

Bearing this in mind, we would really like to implement the multi-phase generalization mentioned in section II. We hope this model, which is able to segment into more than two regions, will aleviate most of the problems we encountered with the simpler model.

Appendix A
Relevant kernels and CPU code

```
#define PHI_IN(X,Y) tex2D(phiTex, X, Y)
#define IMAGE_IN(X,Y) tex2D(imageTex, X, Y)

//
// All of our kernels
//

__global__
void calculateCurvatureForce(float* phi_in, float* force_out, int imgw) {
        int x = blockIdx.x*blockDim.x + threadIdx.x;
        int y = blockIdx.y*blockDim.y + threadIdx.y;

        float fy = PHI_IN(x-1,y) - PHI_IN(x+1,y);
        float fx = PHI_IN(x,y-1) - PHI_IN(x,y+1);
        float fyy = PHI_IN(x-1,y) + PHI_IN(x+1,y) - 2 * PHI_IN(x,y);
        float fxx = PHI_IN(x,y-1) - PHI_IN(x,y+1) - 2 * PHI_IN(x,y);
        float fxy = 0.25f * (PHI_IN(x+1,y+1) - PHI_IN(x-1,y+1)
                      + PHI_IN(x+1,y-1) - PHI_IN(x-1,y-1));
        float force = (fxx*fy*fy - 2.0f * fxy*fx*fy + fyy*fx*fx) /
                    (pow(fx*fx + fy*fy, 1.5f) + eps);

        force_out[y*imgw+x] = force;
}

__device__
float heaviside(float z) {
        if(z > eps) return 1.0f;
        else if(z < -eps) return 0.0f;
        else return 0.5f * (1.0f + z/eps + sin(M_PI * (z/eps))/M_PI);
}

__global__
void calculateC1AndC2(float* image, float* phi, float* force, float* c1,
float* c2, int imgw) {
        int x = blockIdx.x*blockDim.x + threadIdx.x;
        int y = blockIdx.y*blockDim.y + threadIdx.y;

        float h = heaviside(PHI_IN(x,y));
        c1[y*imgw+x] = h * IMAGE_IN(x,y);
        c2[y*imgw+x] = (1.0f - h) * IMAGE_IN(x,y);
}

__global__
void calculateImageForce(float* image, float* force, float sum_c1,
float sum_c2, int imgw) {
        int x = blockIdx.x*blockDim.x + threadIdx.x;
        int y = blockIdx.y*blockDim.y + threadIdx.y;

        force[y*imgw+x] = - pow(IMAGE_IN(x,y) - sum_c1, 2.0f)
                        + pow(IMAGE_IN(x,y) - sum_c2, 2.0f);
}

__global__
void chanVese(float* image, float* phi, float* force, float* imageForce,
int imgw, float mu, float dt) {
        int x = blockIdx.x*blockDim.x + threadIdx.x;
        int y = blockIdx.y*blockDim.y + threadIdx.y;

        int index = y*imgw+x;
        phi[index] += dt * (mu * force[index] + imageForce[index]);
}
```

```
//
// Summing over the c_2 array and calculating the area inside phi while taking
// care of padding using Thrust
//

float sum_c2 = padded_reduce(
        thrust::device_ptr<float>(_c2Ptr),
        thrust::device_ptr<float>(_c2Ptr) + _pwidth*_pheight,
        0.0f,
        thrust::plus<float>(),
        _width, _height, _pwidth, _pheight
);

int in_area = padded_count_if(
        thrust::device_ptr<float>(_phiPtr),
        thrust::device_ptr<float>(_phiPtr) + _pwidth*_pheight,
        std::bind2nd(thrust::greater<float>(), 0.0f),
        _width, _height, _pwidth, _pheight
);

//
// Normalizing the image force array
//

max_force = padded_transform_reduce(
        thrust::device_ptr<float>(_imageForcePtr),
        thrust::device_ptr<float>(_imageForcePtr) + _pwidth*_pheight,
        thrust::absolute_value<float>(),
        0.0f,
        thrust::maximum<float>(),
        _width, _height, _pwidth, _pheight
);

thrust::transform(
                thrust::device_ptr<float>(_imageForcePtr),
                thrust::device_ptr<float>(_imageForcePtr) + _pwidth*_pheight,
                thrust::device_ptr<float>(_imageForcePtr),
                std::bind2nd(thrust::divides<float>(), max_force)
);
```

REFERENCES

[1] M. Kass, A. Witkin, and D. Terzopoulos, “Snakes: Active contour models,” *International Journal of Computer Vision*, vol. 1, no. 4, pp. 321–331, January 1988. [Online]. Available: http://dx.doi.org/10.1007/BF00133570

[2] T. F. Chan and L. A. Vese, “Active contours without edges,” *Image Processing, IEEE Transactions on*, vol. 10, no. 2, pp. 266–277, 2001. [Online]. Available: http://dx.doi.org/10.1109/83.902291

[3] L. A. Vese and T. F. Chan, “A multiphase level set framework for image segmentation using the mumford and shah model,” *International Journal of Computer Vision*, vol. 50, no. 3, pp. 271–293, December 2002. [Online]. Available: http://dx.doi.org/10.1023/A:1020874308076

[4] R. Bowden, T. A. Mitchell, and S. M, “Real-time dynamic deformable meshes for volumetric segmentation and visualisation,” in *Proceedings of the British Machine Vision Conference*, A. F. Clark, Ed., vol. 2, Essex, UK, September 1997, pp. 310–319.

[5] S. Osher and J. A. Sethian, “Fronts propagating with curvature-dependent speed: Algorithms based on hamilton-Jacobi formulations,” *Journal of Computational Physics*, vol. 79, pp. 12–49, 1988. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.46.1266