Department of Engineering Science

Final Year Project

Fast Level Set Segmentation of Biomedical Images using Graphics Processing Units

Author:

Hormuz Mostofi

Keble College

Supervisors:

Dr. Julia Schnabel

Dr. Vicente Grau

May 2009

FINAL HONOUR SCHOOL OF
ENG / ECS / EEM (delete as appropriate)

DECLARATION OF AUTHORSHIP

You should complete this certificate. It should be bound into your fourth year project report, immediately after the title page. Three copies of the report should be submitted to the Chairman of Examiners for your Honour School, c/o Clerk of the Schools, Examination Schools, High Street, Oxford.

Name (in capitals):

College (in capitals): **Supervisor:**

Title of project (in capitals):

Page count: _____

Please tick to confirm the following:

I am aware of the University's disciplinary regulations concerning conduct in examinations and, in particular, of the regulations on plagiarism. ☐

The project report I am submitting is entirely my own work except where otherwise indicated. ☐

It has not been submitted, either wholly or substantially, for another Honour School or degree of this University, or for a degree at any other institution. ☐

I have clearly signalled the presence of quoted or paraphrased material and referenced all sources. ☐

I have acknowledged appropriately any assistance I have received in addition to that provided by my supervisor. ☐

I have not sought assistance from any professional agency. ☐

The project report does not exceed 50 pages (including all diagrams, photographs, references and appendices). ☐

I agree to retain an electronic version of the work and to make it available on request from the Chair of Examiners should this be required in order to confirm my word count or to check for plagiarism.

Candidate's signature: …….. Date: ……..

Acknowledgments

I would like to greatly thank my supervisors *Dr Julia Schnabel* and *Dr Vicente Grau* for their excellent guidance in this project. Their ability to point out the subtle technicalities of biomedical image and volume segmentation proved an invaluable resource.

I would also like to thank *Professor Mike Giles* and *Daniel Goodman* for their guidance in CUDA programming. I found the weekly CUDA meetings led by *Professor Mike Giles* a very helpful forum to learn advanced CUDA programming techniques, whilst keeping up with any developments with respect to CUDA.

Finally, I would like to thank all the staff of the Oxford E-Research Centre for providing the facilities and excellent work environment for the development of this project.

Personal Challenges

At the outset of this project, I had very limited knowledge in the areas of biomedical image segmentation and implementations thereof. A vast literature review of level set segmentation techniques was required in order to take the knowledge gained from C6 Medical Image Analysis course to engineer an initial implementation. As my knowledge of C was initially fairly basic, this first implementation was written in MATLAB using the Image Processing Toolbox.

In order to optimize the algorithm, new skills had to be learnt for the development of a faster sequential C-language algorithm and a much faster parallel algorithm. For the sequential C-language algorithm, extensive knowledge and experience with pointers and arrays needed to be acquired alongside learning OpenGL for visualization. Optimizing further, in order to engineer the significantly faster parallel algorithm, a significant amount of time needed to be spent learning and experimenting with the NVIDIA CUDA programming language and architecture with all its intricacies and subtlety.

The final challenge was to extend to three dimensions, increasing the complexity of sequential and parallel algorithms greatly. Therefore throughout this project there has been a very steep learning curve to overcome in terms of software engineering skill and experience with segmentation algorithms.

Contents

Chapter 1

Introduction

1.1 Image Segmentation

Image segmentation is the task of splitting a digital image into one or more regions of interest. It is a fundamental problem in computer vision and many different methods, each with their own advantages and disadvantages, exist for the task. Image segmentation is a particularly difficult task for several reasons. Firstly, the ambiguous nature of splitting up images into objects of interest requires a trade off between making algorithms more generalized or having many user specified parameters. Secondly, image artifacts such as noise, inhomogeneity, acquisition artifacts and poor contrast, are very difficult to account for in segmentation algorithms without a high level of interactivity from the user.

In this report, segmentation is discussed in a medical imaging context however the proposed algorithm could equally be used in general purpose segmentations. Segmented images are typically used as the input for applications such as classification, shape analysis and measurement. In medical image processing, segmented images are used for studying anatomical structures, diagnosis and assisting in surgical planning. Before computational algorithms existed, segmentation of medical images was a tedious process, performed by hand by clinical experts. This was a fairly accurate, yet slow process. These expert segmentations form the gold standard with which to validate algorithmic segmentations.

As there is no single general solution to image segmentation problems, several techniques exist each of which has its own strengths and limitations. Some common techniques are thresholding based segmentation (discussed in Section 2.2), region based segmentation (i.e. *region growing*),

edge based segmentation and deformable active contour models (such the *snakes* [10] and geodesic active contour models).

1.2 Motivation

In biomedical image segmentation, there is great interest segmenting three dimensional anatomical structures from CT, MRI and PET data. General purpose segmentation on the other hand is typically a two dimensional problem. As a result the data sets used to segment biomedical images are much larger ($O(n^3)$) than those used otherwise ($O(n^2)$), resulting in computation times an order of magnitude greater (although this is not true in all cases). In addition to the time complexity there is also significant storage complexity, for example a very large MRI volume could have as many as 512^3 elements or more, which in combination with large element sizes may result in memory errors on current available hardware when seeking a large area of contiguous memory to write this to.

Furthermore, although the level set approach used in this project is very powerful (its advantages are discussed in Section 2.1), the principal disadvantage of using level sets is that they are relatively slow to compute.

As a result, processing times in a clinical setting may be impractical with more computationally demanding segmentations requiring many minutes or even hours. This provides us with the motivation to accelerate these segmentations by using a parallel algorithm executed on the GPU (Graphics Processing Unit).

1.3 Parallel Processing

The algorithms for processing level sets have vast parallelization potential. Section 3.1 details the algorithms used to discretize the level set equation and Section 3.3 discusses how these can be executed on graphics hardware.

1.3.1 GPGPU

General purpose computation on graphics processing units (GPGPU) is the technique of using graphics hardware to compute applications typically handled by the central processing unit (CPU). Graphics cards over the past two decades have been required to become highly efficient at rendering

increasingly complex 3D scenes at high frame rates. This has forced their architecture to be massively parallel in order to compute graphics faster than general purpose CPUs.

Compared to a CPU, a GPU features many more transistors on the control path due to the lower number of control instructions required. Memory is optimized for throughput and not latency, with strict access patterns. GPUs are not optimized for general purpose programs, and they do not feature the complex instruction sets, or branch control of the modern CPU. Finally, although current high-performance CPUs feature multiple cores for limited parallel processing, GPUs are arguably a more attractive option in terms of lower price and power usage.

The advent of GPGPU programming came with programmable shader units that allowed the programmer to write small programs at each pixel or vertex in the rendering pipeline. In order to program the shader units, shading languages had to be used in conjunction with graphics APIs such as DirectX and OpenGL. NVIDIA developed the high-level shading language Cg to assist in programming shaders, however it still required knowledge of graphics APIs. More recently, languages have been developed that allow the programmer to implement algorithms without any knowledge of graphics APIs or architectures. One such language is NVIDIA CUDA, and is the language chosen for the optimizations in this project.

1.3.2 CUDA

Compute Unified Device Architecture, or CUDA, is NVIDIA's GPGPU technology that allows for programming of the GPU without any graphics knowledge. The C language model has at its core three key abstractions, from [16]: a hierarchy of thread groups (to allow for transparent scalability), shared memories (allowing access to low-latency cached memory), and barrier synchronization (to prevent *race conditions*). This breaks the task of parallelization into three sub problems, which allows for language expressivity when threads cooperate, and scalability when extended to multiple processor cores.

Framework

CUDA uses extends C by allowing a programmer to write *kernels* that when invoked execute thousands of lightweight identical threads in parallel. CUDA arranges these threads into a hierarchy of blocks and grids, as can be seen in Figure 1.1 allowing for runtime transparent scaling of code to different GPUs. The threads are identified by their location within the grid and block, making

CUDA perfectly suited for tasks such as image processing where each thread is easily assigned to an individual *pixel* (picture element) or *voxel* (volume element).

When writing and optimizing complex parallel code in CUDA it is often found that threads may need to cooperate, typically by sharing data or temporarily halting execution. The memory hierarchy of CUDA threads is shown in Figure 1.2. Here it can be seen that each thread has access to: a per-thread private local memory, a per-block on-chip shared memory to share data between threads, and finally an off-chip global memory accessible to all threads within all blocks. There are also constant and texture memory spaces accessible to all threads, however these are not featured in our algorithm and so will not be discussed in any further detail.

Performance Guidelines

There are many techniques to optimize a parallel algorithm. Firstly, the optimum block and grid sizes should be used to ensure maximum 'occupancy'. Occupancy is the ratio of the number of active warps (32 parallel threads) to the maximum number of active warps supported by the GPU multiprocessor. To maximize efficiency, there is a trade off between making the occupancy high enough to ensure no multiprocessor is ever idle, and making it low enough to ensure no *bank conflicts* (when two threads attempting to access the same location in shared memory have their accesses serialized).

Secondly, one of the best ways in which to optimize the parallelization is through efficient memory usage. The global memory space is not cached and therefore has a much higher latency and lower bandwidth than on-chip shared memory. Therefore it is the aim of the programmer to minimize global memory accesses. From [16], it recommended that each thread in a block firstly loads data from global memory to shared memory, synchronizes with all other threads within the thread block to ensure shared memory locations have been written to, processes the data, synchronizes again to ensure shared memory has been fully updated with results, and finally writes the results back to global memory coalesced.

Coalescence is an important concept in memory management as it can speed up memory reads and stores significantly. [16] lists the following three conditions for coalescing: "threads must access either 32-bit words, 64-bit words, or 128-bit words", "all 16 words must lie in the same segment of size equal to the memory transaction size" and "threads must access the words in sequence".

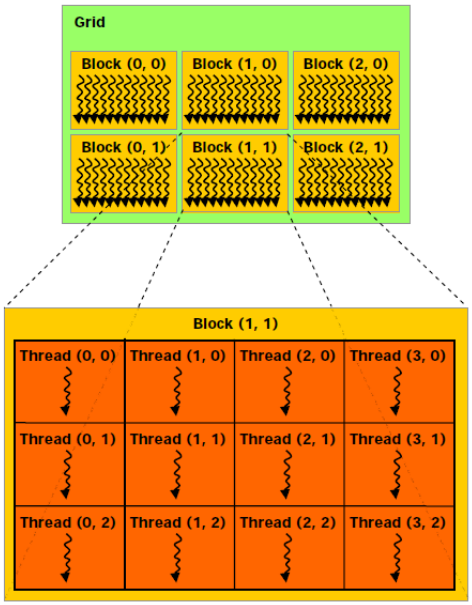


Figure 1.1: A grid of thread blocks. This figure is taken from [16]

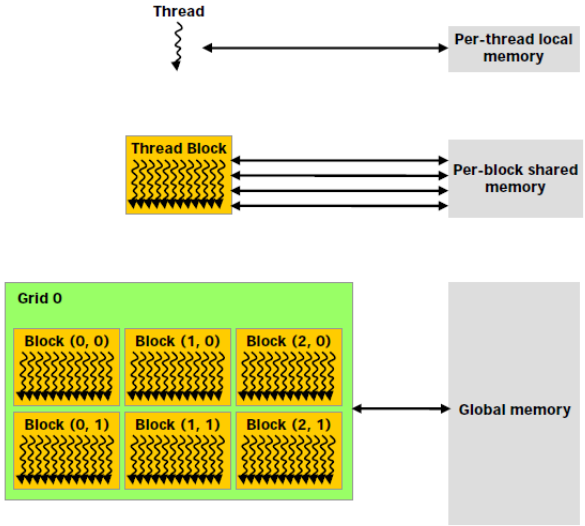


Figure 1.2: The memory hierarchy of CUDA threads and blocks. This figure is taken from [16]

Chapter 2

Method

In this section, we introduce the level set method and dynamic implicit surfaces. Their role in segmentation is discussed having introduced and defined mathematical constructs such as signed distance transforms.

2.1 Level Set Method

The level set method evolves a contour (in two dimensions) or a surface (in three dimensions) implicitly by manipulating a higher dimensional function, called the level set function $\phi(\mathbf{x},\mathbf{t})$. The evolving contour or surface can be extracted from the zero level set $\Gamma(\mathbf{x},\mathbf{t}) = \{\phi(\mathbf{x},\mathbf{t}) = \mathbf{0}\}$. The advantage of using this method is that topological changes such as merging and splitting of the contour or surface are catered for implicitly, as can be seen below in Figure 2.1. The level set method, since its introduction by Osher and Sethian in [19], has seen widespread application in image processing, computer graphics (surface reconstructions) and physical simulation (particularly fluid simulation).

The evolution of the contour or surface is governed by a level set equation. The solution tended to by this partial differential equation is computed iteratively by updating ϕ at each time interval. The general form of the level set equation is shown below.

$$\frac{\partial \phi}{\partial t} = -|\nabla \phi| \cdot F \tag{2.1}$$

In the above level set equation F is the velocity term that describes the level set evolution. By manipulating F, we can guide the level set to different areas or shapes, given a particular

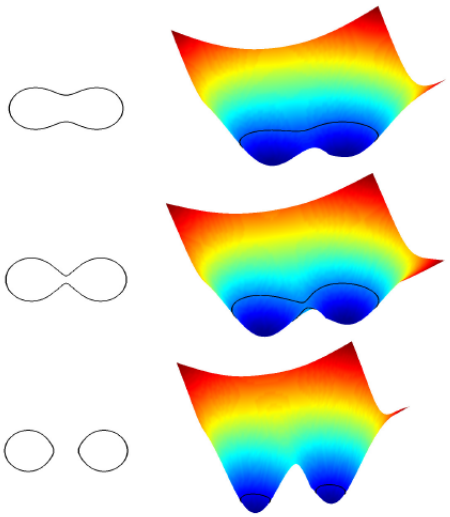


Figure 2.1: The relationship between the level set function (left) and contour (right) can be seen. It can be seen evolving the surface splits the contour.

initialization of the level set function.

2.2 Segmentation using Level Sets

Typically, for applications in image segmentation F is dependent on the pixel intensity or curvature values of the level set. The importance of having a curvature term is shown in Figure 2.2. Here there is no force to smoothen high curvatures, resulting in the contour *leaking*. This is when the level set surface evolves through a anatomical boundary into another anatomical object that was not intended to be segmented. This also makes segmentation difficult for objects which have very high curvature as the curvature weighting term often needs to be set very low in order to allow for these high curvatures, yet doing so may result in such leaking.

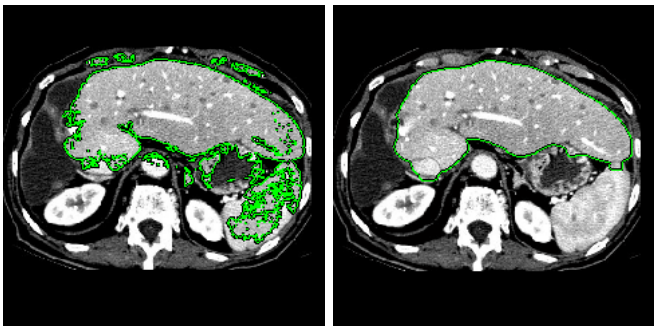


Figure 2.2: Leaking when there is no curvature term (or $\alpha = 1$)

F may also be dependent on an edge indicator function, which is defined as having a value zero

on an edge, and non-zero otherwise. This causes F to slow the level set evolution when on an edge.

In [13] F is dependent on data and curvature functions only (with a weighting parameter between the two) for the purposes of image segmentation. Therefore, we will adopt the same methodology making the level set equation take the form

$$\frac{\partial \phi}{\partial t} = -|\nabla \phi| \left[\alpha D(I) + (1-\alpha) \nabla \cdot \frac{\nabla \phi}{|\nabla \phi|} \right] \tag{2.2}$$

where the data function $D(I)$ tends the solution towards targeted features, and the mean curvature term $\nabla \cdot (\nabla \phi / |\nabla \phi|)$ keeps the level set function smooth. Weighting between these two is $\alpha \in [0, 1]$, a free parameter that is set beforehand to control how smooth the contour or surface should be.

The data function $D(I)$ acts as the principal 'force' that drives the segmentation. By making D positive in desired regions or negative in undesired regions, the model will tend towards the segmentation sought after. A simple speed function that fulfills this purpose, used by Lefohn, Whitaker and Cates in [13, 2], is given by

$$D(I) = \epsilon - |I - T| \tag{2.3}$$

which is plotted in Figure 2.3. Here T describes the central intensity value of the region to be segmented, and ϵ describes the intensity deviation around T that is part of the desired segmentation. Therefore if a pixel or voxel has an intensity value within the $T \pm \epsilon$ range the model will expand, and otherwise it will contract.

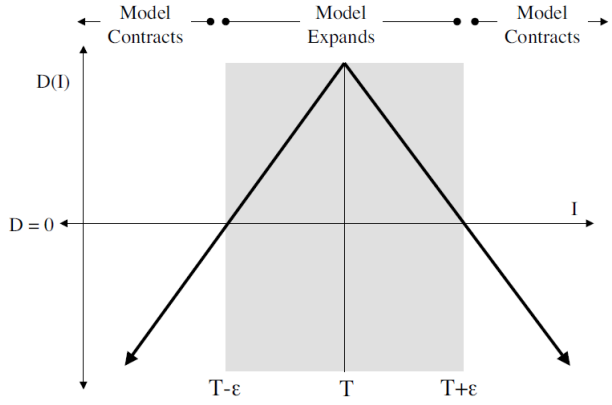


Figure 2.3: The speed term from [2]

Therefore the three user parameters that need to be specified for segmentation are T,ϵ and α. An initial mask (to be transformed to a signed distance function as discussed in Section 2.2.1) for the level set function is also required, which may take the form of a cube in three dimensions or a square in two dimensions, or any other arbitrary closed shape. Typically, the user selects spherical seed points specifying the center in i, j, k space and the radius to guide the level set to the anatomical object of interest.

The level set iteration can be terminated once ϕ has converged, or after a certain number of iterations.

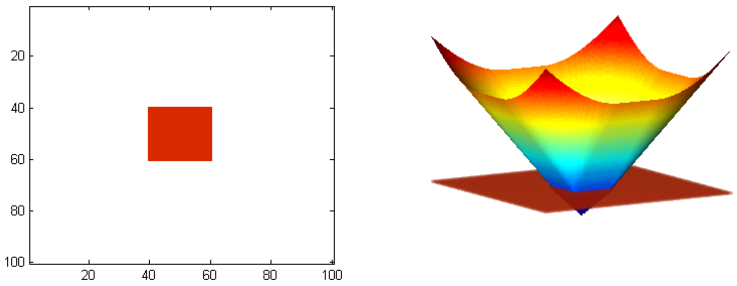
2.2.1 Signed Distance Transforms

A distance transform assigns a value for every pixel (or voxel) within a binary image containing one or more objects a value which represents the minimum distance from that pixel to the closest pixel on the boundary of the object(s). The mathematical definition of a distance function $D : \mathbb{R}^3 \rightarrow \mathbb{R}$ for a set S, from [19], is

$$D(r, S) = \min|r - S| \text{ for all } r \in \mathbb{R}^3 \tag{2.4}$$

A *signed* distance transform assigns the sign of the distance value as positive for those pixels outside the object, and negative for those inside it. This is the sign convention that will be followed in the implementation, however the opposite sign convention could also be used. It should be noted that the distance values depend on the chosen metric for distance: some common distance metrics are Euclidean distance, chessboard distance, and city block distance. Many of the algorithms that compute signed Euclidean distance transforms (SEDT) often trade accuracy for efficiency and feature varying levels of complexity.

Signed distance transforms are required to initialize ϕ and also to reinitialize it every certain number of iterations. Computation of the initialization of ϕ is required before iteration of the level set equation can take place, and this will typically be a signed distance transform of an initial mask. Therefore the level set segmentation filter requires two images: an initial mask (which indicates targeted regions) and a *feature* image (which is the image to be segmented). The choice of how often to reinitialize is an important one: if the number of iterations between reinitialization is too low the level set will simply oscillate, if it is too high the risky of instabilities is elevated.



(a) Arbitrary Initial Mask

(b) Signed Distance Transform of Mask with Zero Level Set Overlaid

Figure 2.4: 2D Signed Euclidean Distance Transform

Alternatively, [8] provides a method of evolving level sets without reinitialization using signed distance transforms by forcing the level set function to be close to a signed distance function.

2.2.2 3D Volume Segmentations

Extending a two dimensional level set segmentation algorithm to three dimensions is a relatively straightforward task, however requires careful consideration of boundary conditions. There are many more derivatives that are required in order to compute the level set update. In addition to the increased number of variables, creating a signed Euclidean distance function is one of the major challenges in developing 3D segmentation code. Unfortunately, neither C code or CUDA code was available to perform distance transform (re)initialization in 3D and therefore MATLAB was used to initialize and reinitialize the level set during execution. There has however been recent work on CUDA accelerated distance transforms from [23],[5]. The storage and computational complexity of 3D volume segmentation must also be appreciated and forms much of the motivation for acceleration with CUDA.

Chapter 3

Implementation

3.1 Level Set Algorithm

3.1.1 Upwinding

Equation (2.1), the level set equation, needs to be discretized for both sequential and parallel computation. This is done using the *up-wind* differencing scheme. The following explanation of *upwinding* is from [18].

A first order accurate method for time discretization of equation (2.1), is given by the forward Euler method, from [18]:

$$\frac{\phi^{t+\Delta t} - \phi^t}{\Delta t} + F^t \cdot \nabla \phi^t = 0 \tag{3.1}$$

where ϕ^t represents the current values of ϕ at time t, F^t represents the velocity field at time t, and $\nabla \phi^t$ represents the values of the gradient of ϕ at time t. When computing the gradient, a great deal of care must be taken with regards to the spatial derivatives of ϕ. This is best exemplified by considering the expanded form of equation (3.1)

$$\frac{\phi^{t+\Delta t} - \phi^t}{\Delta t} + u^t \phi_x^t + v^t \phi_y^t + w^t \phi_z^t = 0 \tag{3.2}$$

where u, v, w are the x, y, z components of F. For simplicity, consider the one dimensional form of equation (3.2) at a specific grid point x_i

$$\frac{\phi^{t+\Delta t} - \phi^t}{\Delta t} + u_i^t (\phi_x)_i^t = 0 \tag{3.3}$$

where $(\phi_x)_i$ is the spatial derivative of ϕ at x_i. The method of characteristics indicates whether to use a forward or backwards difference method for ϕ based on the sign of u_i at the point x_i. If $u_i > 0$, the values of ϕ are moving from left to right, and therefore backwards difference methods (D_x^- in Equations 3.4) should be used. Conversely, if $u_i < 0$, forward difference methods (D_x^+ in Equations 3.4) should be used to approximate ϕ_x. It is this process of choosing which approximation for the spatial derivative of ϕ to use based on the sign of u_i that is known as *upwinding*.

Extending this to three dimensions and assuming an isotropic resolution, from [13], results in the derivatives below required for the level set equation update.

$$\begin{array}{lll} D_x = (\phi_{i+1,j,k} - \phi_{i-1,j,k})/2 & D_x^+ = \phi_{i+1,j,k} - \phi_{i,j,k} & D_x^- = \phi_{i,j,k} - \phi_{i-1,j,k} \\ D_y = (\phi_{i,j+1,k} - \phi_{i,j-1,k})/2 & D_y^+ = \phi_{i,j+1,k} - \phi_{i,j,k} & D_y^- = \phi_{i,j,k} - \phi_{i,j-1,k} \\ D_z = (\phi_{i,j,k+1} - \phi_{i,j,k-1})/2 & D_z^+ = \phi_{i,j,k+1} - \phi_{i,j,k} & D_z^- = \phi_{i,j,k} - \phi_{i,j,k-1} \end{array} \tag{3.4}$$

$\nabla\phi$ is approximated using the upwind scheme.

$$\nabla\phi_{\max} = \begin{bmatrix} \sqrt{\max(D_x^+, 0)^2 + \max(-D_x^-, 0)^2} \\ \sqrt{\max(D_y^+, 0)^2 + \max(-D_y^-, 0)^2} \\ \sqrt{\max(D_z^+, 0)^2 + \max(-D_z^-, 0)^2} \end{bmatrix} \tag{3.5}$$

$$\nabla\phi_{\min} = \begin{bmatrix} \sqrt{\min(D_x^+, 0)^2 + \min(-D_x^-, 0)^2} \\ \sqrt{\min(D_y^+, 0)^2 + \min(-D_y^-, 0)^2} \\ \sqrt{\min(D_z^+, 0)^2 + \min(-D_z^-, 0)^2} \end{bmatrix} \tag{3.6}$$

Finally, depending on whether $F_{i,j,k} > 0$ or $F_{i,j,k} < 0$, $\nabla\phi$ is

$$\nabla\phi = \begin{cases} ||\nabla\phi_{\max}||_2 & \text{if } F_{i,j,k} > 0 \\ ||\nabla\phi_{\min}||_2 & \text{if } F_{i,j,k} < 0 \end{cases} \tag{3.7}$$

$$\phi(t + \Delta t) = \phi(t) + \Delta t F |\nabla\phi| \tag{3.8}$$

The speed term F, as discussed before, is based on the pixel intensity values and curvature values.

This implementation is computationally very demanding (in terms of time taken and storage required) and therefore there have been several developments to optimize the implementation. Some of these include the narrow band method [1] and sparse field method [24]. The narrow band method restricts the computation of level set update to a thin band of 'active' pixel/voxels that are on or near to the level set implicit contour/surface. This speeds up computation as there is very little need to update the level set for pixel/voxels very far from the contour. The sparse field method adopts a similar approach, performing computations on a domain only one cell wide.

3.1.2 Curvature

Curvature is computed based on the values of the current level set using the derivatives below. In two dimensions only $D_x^{+y}, D_x^{-y}, D_y^{+x}, D_y^{-x}$ below are required, alongside the derivatives defined previously. In three dimensions, all the derivatives below are required. Again, we are making isotropic assumptions.

$$\begin{aligned}
D_x^{+y} &= (\phi_{i+1,j+1,k} - \phi_{i-1,j+1,k})/2 & D_x^{-y} &= (\phi_{i+1,j-1,k} - \phi_{i-1,j-1,k})/2 \\
D_x^{+z} &= (\phi_{i+1,j,k+1} - \phi_{i-1,j,k+1})/2 & D_x^{-z} &= (\phi_{i+1,j,k-1} - \phi_{i-1,j,k-1})/2 \\
D_y^{+x} &= (\phi_{i+1,j+1,k} - \phi_{i+1,j-1,k})/2 & D_y^{-x} &= (\phi_{i-1,j+1,k} - \phi_{i-1,j-1,k})/2 \\
D_y^{+z} &= (\phi_{i,j+1,k+1} - \phi_{i,j-1,k+1})/2 & D_y^{-z} &= (\phi_{i,j+1,k-1} - \phi_{i,j-1,k-1})/2 \\
D_z^{+x} &= (\phi_{i+1,j,k+1} - \phi_{i+1,j,k-1})/2 & D_z^{-x} &= (\phi_{i-1,j,k+1} - \phi_{i-1,j,k-1})/2 \\
D_z^{+y} &= (\phi_{i,j+1,k+1} - \phi_{i,j+1,k-1})/2 & D_z^{-y} &= (\phi_{i,j-1,k+1} - \phi_{i,j-1,k-1})/2
\end{aligned} \tag{3.9}$$

Using the *difference of normals* method from [13], curvature is computed using the above derivatives with the two normals $\mathbf{n}^+$ and $\mathbf{n}^-$.

$$\mathbf{n}^+ = \begin{bmatrix} \frac{D_x^+}{\sqrt{(D_x^+)^2+\left(\frac{D_y^{+x}+D_y}{2}\right)^2+\left(\frac{D_z^{+x}+D_z}{2}\right)^2}} \\ \frac{D_y^+}{\sqrt{(D_y^+)^2+\left(\frac{D_x^{+y}+D_x}{2}\right)^2+\left(\frac{D_z^{+y}+D_z}{2}\right)^2}} \\ \frac{D_z^+}{\sqrt{(D_z^+)^2+\left(\frac{D_y^{+z}+D_x}{2}\right)^2+\left(\frac{D_y^{+z}+D_y}{2}\right)^2}} \end{bmatrix} \tag{3.10}$$

$$\mathbf{n}^- = \begin{bmatrix} \frac{D_x^-}{\sqrt{(D_x^-)^2+\left(\frac{D_y^{-x}+D_y}{2}\right)^2+\left(\frac{D_z^{-x}+D_z}{2}\right)^2}} \\ \frac{D_y^-}{\sqrt{(D_y^-)^2+\left(\frac{D_x^{-y}+D_x}{2}\right)^2+\left(\frac{D_z^{-y}+D_z}{2}\right)^2}} \\ \frac{D_z^-}{\sqrt{(D_z^-)^2+\left(\frac{D_y^{-z}+D_x}{2}\right)^2+\left(\frac{D_y^{-z}+D_y}{2}\right)^2}} \end{bmatrix} \tag{3.11}$$

The two normals are used to compute divergence, allowing for mean curvature to be computed as shown below in equation (3.12).

$$H = \frac{1}{2}\nabla \cdot \frac{\nabla\phi}{|\nabla\phi|} = \frac{1}{2}((\mathbf{n}_x^+ - \mathbf{n}_x^-) + (\mathbf{n}_y^+ - \mathbf{n}_y^-) + (\mathbf{n}_z^+ - \mathbf{n}_z^-)) \tag{3.12}$$

3.1.3 Stability

From [18], a finite difference approximation to a linear partial differential equation is convergent if and only if it is both consistent and stable. Stability implies that small errors in the solution are not amplified during iteration. Stability is enforced using the Courant-Friedrichs-Lewy (CFL) condition which states the numerical wave speed must be greater than the physical wave speed, i.e. $\Delta x/\Delta t > |u|$. Rearranging, we have

$$\Delta t < \frac{\Delta x}{\max\{|u|\}} \tag{3.13}$$

which is usually implemented, through variants of equation (3.13), by choosing a *CFL number* that lies between 0 and 1 to further guarantee stability.

Another measure taken to ensure stability is the inclusion of a floating point relative accuracy term in the denominator of any fractions to avoid singularity errors as the denominator tends to

zero. This is done in equations (3.10),(3.11) to ensure that **n** does not tend to infinity if the square root is zero.

3.2 Sequential Implementation

For this project, two dimensional implementations of the code in MATLAB and C were written before progressing to CUDA code. Once these had been written and tested, three dimensional implementations were coded. The following pseudo code outlines the structure of the MATLAB, C and CUDA implementations, with only minor differences between the different versions.

Algorithm 1: Pseudo code for Level Set Segmentation

Input: Feature Image I, Initial Mask m, Threshold T, Range ϵ, Iterations N, Reinitialize Every $RITS$
Output: Segmentation Result

Initialize ϕ_0 to Signed Euclidean Distance Transform (SEDT) from mask m
Calculate Data Speed Term $D(I) = \epsilon - |I - T|$
forall N *Iterations* **do**
  Calculate First Order Derivatives $D_x^{(\pm)}, D_y^{(\pm)}, D_z^{(\pm)}$
  Calculate Second Order Derivatives $D_x^{(\pm y,z)}, D_y^{(\pm x,z)} ... D_z^{(\pm x,y)}$
  Calculate Curvature Terms $\mathbf{n}^+, \mathbf{n}^-$
  Calculate Gradient $\nabla\phi$
  Calculate Speed Term $F = \alpha D(\bar{x}) + (1-\alpha)\nabla \cdot \frac{\nabla\phi}{|\nabla\phi|}$
  Update Level Set Function $\phi(t + \Delta t) = \phi(t) + \Delta t F|\nabla\phi|$
  if *Iterations % RITS == 0* **then**
    Reinitialize ϕ to SEDT
  end
end

3.2.1 MATLAB

The first task was to write code in MATLAB to learn about the inner workings of 2D image segmentation. The MATLAB Image Processing Toolbox provides many functions (such as the ability to load, re-sample and filter images, compute distance transforms and easily visualize the level set evolution) which kept code reasonably concise.

The code is split into two files (a launcher and a kernel), in order to separate the initialization and level set update code. The launcher handles the image loading and re-sampling, with the functions `imread` and `imresize`. The user specifies parameters for threshold values T, range ϵ and

curvature weighting α, runs the launcher and then proceeds to draw a closed polygon that will form the initial mask (providing some basic interactivity).

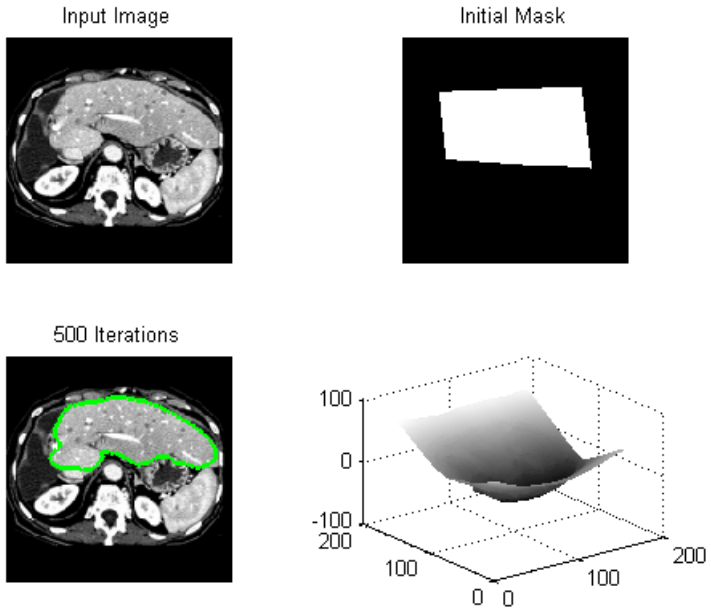


Figure 3.1: MATLAB user interface showing four sub figures with the input image, the initial mask, the current zero level set interface superimposed on the input image and the current level set surface in 3D

The level set function ϕ is then initialized to a signed distance function of this mask, and iteration of the level set equation begins for a fixed number of iterations (also user-definable). Reinitialization of the level set is performed once every 50 iterations, and the current level set contour and surface are displayed every 20 iterations.

The derivatives are calculated by subtracting shifted matrices of the level set function. Note that derivatives are not calculated in an element by element fashion, as this would be less efficient.

The MATLAB code also features the Courant-Friedreichs-Lewy (CFL) condition which was described in section 3.1.3 to enforce stability, instead of arbitrarily defining Δt.

Finally, the user has the option of downsampling the input image in order to speed up the computation.

The MATLAB code was later adapted to 3D volume segmentation. 2D MATLAB code can be found in Appendix A.

3.2.2 C

Initially C code was written to most closely mirror the MATLAB code. The C code would serve as check for the parallel implementations to ensure correct values were calculated and, more importantly, as a benchmark to measure speed ups against.

Whereas the MATLAB code uses shifts of the matrix ϕ to calculate derivatives, in C the level set function ϕ is stepped through in an element by element fashion. The code is structured such that for a given i, j, k all derivatives are calculated; this is opposed to structuring the code such that for a given derivative all i, j, k for that derivative are calculated. The latter method could not form the structure for a CUDA algorithm, and in fact this did briefly form the initial C implementation which was later restructured. Many boundary conditions had to be placed in order to ensure that derivatives took the value zero at certain boundaries. For example the forward difference derivative $D_x^+ = \phi_{i+1,j} - \phi_{i,j}$ must equal zero when $i = \text{imageW}$ as there is no $\phi_{i+1,j}$ term. The complexity of this task increases in three dimensions as there are six boundaries instead of four boundaries to condition for.

In the C implementation, the feature image and current level set function are stored in memory as one dimensional arrays. Therefore they are stepped through by nested for loops, which depending on the dimensionality of the problem would either loop i, j or i, j, k. To go from the two dimensional i, j indices to a one dimensional index *ind* when stepping through the array, the equation $\text{ind} = i + (j \times \text{imageW})$ is used and in three dimensions the equation $\text{ind} = i + (j \times \text{imageW}) + (k \times \text{imageW} \times \text{imageH})$ is used.

This implementation uses some external code for computing signed distance transforms and loading 2D bitmap images. In 3D raw (header-less) files were used. The `sedt2d` (Signed Euclidean Distance Transform in 2D) function written by Timothy Terriberry was used for the distance transforms and `bmploader.cpp` from the NVIDIA CUDA SDK library was used for image loading.

Finally, in order to visualize the level set evolution the combination of OpenGL and GLUT (OpenGL Utility Toolkit) was used to render the current zero level set. Rendering code was kept as compact and efficient as possible in order to have as minimal an effect as possible on performance whilst also making the program more comparable with the MATLAB code. Its principal purpose was of course to visualize how the level set was evolving (checking for instabilities, incorrect parameters for thresholding, range and curvature) and also view the final segmentation. When testing

and timing this algorithm, these visualizations were of course not used. A small amount of time was spent designing a GUI interface with QT however this never become fully integrated due to time constraints.

3.3 Parallel Implementation

3.3.1 Unoptimized Version

The first parallel implementation followed the structure shown in the pseudo code below. In CUDA, it is assumed that both the host and device maintain their own DRAM [16]. Host memory is allocated as before using `malloc` and device memory is allocated using `cudaMalloc`. As memory bandwidth between the host memory and device memory is low (it is much lower than the bandwidth between the device and the device memory), it is recommended to keep the number of transfers to a minimum. In order to minimize the latency of accessing the shared memory it is recommended to make the block size a multiple of 16 and use the `cudaMallocPitch` routine to allocate memory with padding if the images's x dimension is not a multiple of 16. Therefore most CUDA programs follow a standard structure of initialization, host to device data transfer, compute, and finally memory transfer of compute results from device to host.

Unfortunately the algorithm for computing signed distance transforms in 2D is not executed in CUDA and creating one from scratch would have been beyond the scope of this project. Therefore device to host memory transfers were required every time reinitialization was necessary. Of course, when timing it is possible to stop and start timers during this process. As stated previously, for 3D signed Euclidean distance transforms neither a sequential or CUDA version was available.

CUDA threads are assigned a unique thread ID that identifies its location within the thread block and grid. This provides a natural way to invoke computation across the image and level set domain, by using the thread IDs for addressing. This is best explained with the tables below. Assume our image has dimensions 4×4 and the block size is 2×2. Invoking the kernel with a grid size of 2 blocks $\times$ 2 blocks results in the 16 threads shown in table 3.3.1, in the form (`threadIdx.y`,`threadIdx.x`). These threads are grouped into blocks of four, as shown in table 3.3.1, in the form (`blockIdx.y`,`blockIdx.x`).

As each thread has access to its own `threadIdx` and `blockIdx`, global indices (i, j) can be determined using the equations

Algorithm 2: Parallel Implementation Pseudo code

Initialize $\phi^0_{i,j,k}, D$ on host memory
Allocate memory for ϕ^n, ϕ^{n+1}, D on device
Copy ϕ^0, D from host to device
forall *N Iterations* **do**
  Execute Level Set Update CUDA Kernel $\phi^{n+1}_{i,j,k} = \phi^n_{i,j,k} + \Delta t F |\nabla \phi^n_{i,j,k}|$
  Swap pointers of $\phi^n_{i,j,k}, \phi^{n+1}_{i,j,k}$
  if *Iterations % RITS == 0* **then**
    Copy ϕ from device to host
    Reinitialize ϕ to Signed Euclidean Distance Transform
    Copy ϕ from host to device
  end
end
Copy ϕ from device to host

(0,0)	(0,1)	(0,1)	(0,1)
(1,0)	(1,1)	(1,0)	(1,1)
(0,0)	(0,1)	(0,0)	(0,1)
(1,0)	(1,1)	(1,0)	(1,1)

Table 3.1: Thread IDs of 16 threads grouped into 4 blocks

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
```

where `blockDim.x` and `blockDim.y` represent the dimensions of the block (which in this case are both equal to 2). Of course, much larger block sizes are used, keeping the block x dimension (BX) a multiple of 16 for maximum speed. The effect of different block sizes on performance is analysed in Section 4.1.2.

Once these indices were set up, it was relatively straightforward to transfer the level set update code to a CUDA kernel. Although this code exhibited speedups over the single threaded implementation, there was still significant optimization to perform as a great deal of computation time was being wasted on access global memory continuously. Therefore this is a *naive* implementation.

Some features such as the CFL condition, could not be implemented in this parallel version

(0,0)	(0,1)
(1,0)	(1,1)

Table 3.2: Block IDs of 4 blocks grouped into a grid

without slowing down computation time significantly. This is because such a condition requires the determination of the largest element of $\nabla\phi$ which is computed roughly half way through the update procedure. Therefore integrating this condition would require transferring $\nabla\phi$ and curvature terms back to host memory to determine $\max\{F|\nabla\phi|\}$, or perhaps more efficiently calling a CUDA kernel to determine the largest element. The cost of this added complexity and slowdown outweighed the benefits, and therefore Δt was chosen to be a fixed parameter.

3.3.2 2D Shared Memory Optimization

In order to keep the number of costly accesses to device memory at a minimum, effective use of the on-chip shared memory is essential. This along with maximizing parallel execution and optimization of instruction usage form the three main performance optimization strategies for CUDA [16].

Integrating use of the shared memory into the CUDA kernel requires partitioning the level set domain into tiles. For first order finite difference problems such as ours each tile must also contain values for neighborhood nodes (often known as *halo* nodes) for the $i \pm 1$ and $j \pm 1$ elements, which would be stored in separate tiles, so these must also be read into shared memory. As the size of the shared memory is only 16 KB, the sizes of the tiles and corresponding halo are limited. [14] outlines a framework for such a process that may serve as a good model for a multi GPU implementation, however the kernel will need to be modified as it is optimized for higher order stencils (without cross-derivative terms). Instead, tiling code was adapted from Giles' (2008) 'Jacobi iteration for Laplace discretization' algorithm [6] which supports cross-derivatives well. The shared memory management technique in this finite difference algorithm accelerated the global memory implementation by over an order of magnitude.

The two dimensional segmentation algorithm does not require any $k \pm 1$ terms, making the shared memory management more straightforward. For a block (and tile) size of $BX \times BY$ there are $2 \times (BX + BY + 2)$ halo elements, as can be seen in Figure 3.2. In this figure the darker elements represent the thread block (the active tile) and the lighter elements represent the halo. It is in this manner that the domain of the computation is partitioned and this results in overlapping of the halo nodes.

Each thread loads ϕ_n values from global memory to the active tile stored in shared memory. However, depending on the location of the thread within the thread block it may also load a single halo node into the shared memory. Therefore in order to load all halo nodes, this technique assumes

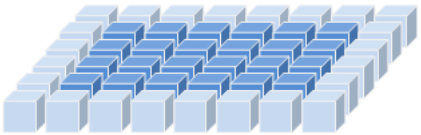


Figure 3.2: 2D Shared Memory Arrangement

that there are at least as many interior nodes as there are halo nodes. Before data can be loaded into the halos, the thread ID needs to be mapped to the location of a halo node both within the halo and within the global indices. The segment of code that sets up the halo indices (both local and global) for loading into shared memory is shown below, code is from [6].

```
k    = threadIdx.x + threadIdx.y*BLOCK_X;
  halo = k < 2*(BLOCK_X+BLOCK_Y+2);

  if (halo) {
    if (threadIdx.y<2) {                // y-halos (coalesced)
      i = threadIdx.x;
      j = threadIdx.y*(BLOCK_Y+1) - 1;
    }
    else {                              // x-halos (not coalesced)
      i = (k%2)*(BLOCK_X+1) - 1;
      j =  k/2 - BLOCK_X - 1;
    }
```

The first $2 \times (BX + BY + 2)$ threads are assigned to load values into the halo in this manner. This is best visualised with the example of a 6×6 thread block as shown below in Figure 3.3.

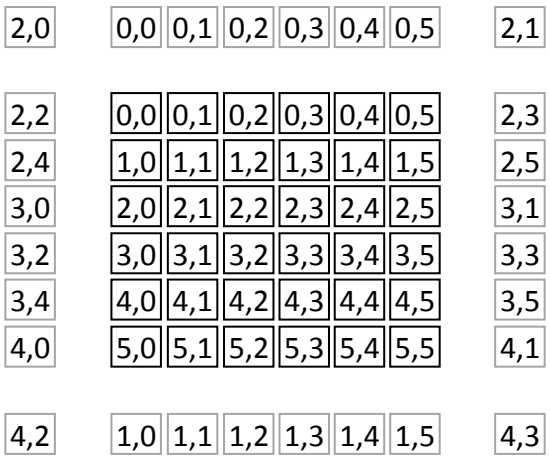


Figure 3.3: Tile and halo showing for a 6×6 block the mapping of thread IDs to halo nodes

This method of loading elements has been chosen in order to maximize *coalescence*. Not only are the interior tile nodes loaded coalesced, but as can be seen above, the first 12 elements of the thread block load the y halos (above and below the the interior tile excluding corners) in a coalesced manner. The side halos (x halos) loads are non-coalesced. When writing back results to global memory, as only the interior nodes have updated values they are written to global memory coalesced.

3.3.3 3D Shared Memory Optimization

In three dimensions, $k \pm 1$ terms are required and therefore these values need to also be stored in shared memory. CUDA only allows for two dimensional grid sizes (even though blocks can be three dimensional), implying that the number of blocks in the z dimension cannot exceed 1. The algorithm by Giles [6] uses three k-planes of data for this purpose as shown in Figure 3.4.

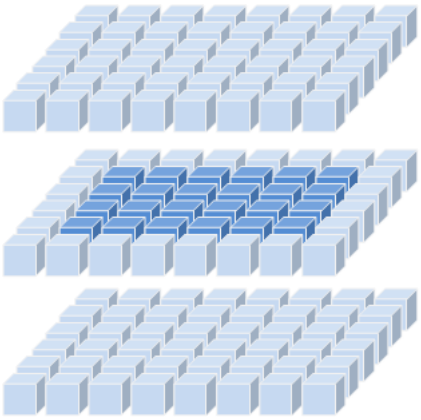


Figure 3.4: 3D Shared Memory Arrangement

Before looping over the k-planes begins, the ϕ_k plane is loaded into the $k+1$ plane of shared memory. Upon entering the loop this plane is shifted down one plane to the k plane and the ϕ^n_{k+1} plane is loaded into the $k+1$ plane. Level set function values for ϕ^{n+1}_k are calculated and written coalesced back to global memory. The k plane is then shifted to the $k-1$ plane, the $k+1$ plane is shifted to the k plane, and new values are loaded from ϕ^n_{k+1} to the $k+1$ plane. This looping over the z dimension continues for all $z <$ imageD. In this manner, each block actually processes a $BX \times BY \times$ imageD sub domain.

Chapter 4

Results

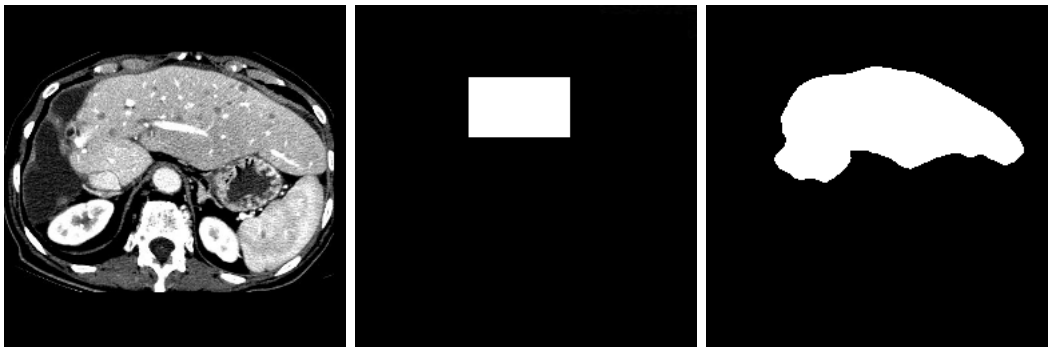
In the following, the results of the speed ups attained by optimizing using CUDA will be shown. However, before this can be done, some preliminaries need to be listed. Firstly, all hardware testing was done on a single PC with an Intel Core 2 Duo T8100 Processor with a clock speed of 2.1 GHz and 4 GB of RAM. The graphics hardware used was the NVIDIA GeForce 8600M GT, with CUDA 2.1 software installed. It should be noted that at the time of writing, CUDA 2.2 was available although in beta form and was not chosen due to potential instabilities. Timing code used was from the `cutil` library provided in the CUDA toolkit.

Although 8600M GT is adequate for CUDA development, it has rather limited performance in comparison to other graphics chips. This implies that the performance speed ups below could potentially be up to a further order of magnitude faster on latest hardware. For example, although the shader processing rate of 8600M GT is quoted as 91.2 Gigaflops the recently released GeForce GTX 295 boasts an impressive 1788.48 Gigaflops potentially allowing for another order of magnitude speed up from the 8600M GT hardware. This is mainly due to the increased number of on chip multiprocessors, however to lesser extent is also due to the device being of higher *compute capability*: there are fewer limitations (such as support for double precision arithmetic) and relaxed requirements for coalescing memory transfers. Due to the transparent runtime scalability of CUDA kernels, very few adjustments would need to be made to tailor code for new hardware. Nonetheless, the most obvious adjustment would be to increase the size of the thread blocks for increased occupancy.

4.1 Speed Tests and Analysis

4.1.1 2D Segmentations

In Figure 4.1 the example of a liver segmentation is shown. The liver data is of good contrast and dimension 256×256 (which is a multiple of 16 implying no memory padding is required in CUDA). The liver has been entirely segmented, with the initial mask as the input. The time taken for 5000 iterations in MATLAB, C, CUDA (Unoptimized) and CUDA (Optimized) are shown in Table 4.1.



(a) Feature Image Input - Liver (b) Initial Mask Input (c) Output of Segmentation

Figure 4.1: 2D Liver Segmentation with parameters $T = 180, \epsilon = 45, \alpha = 0.003$

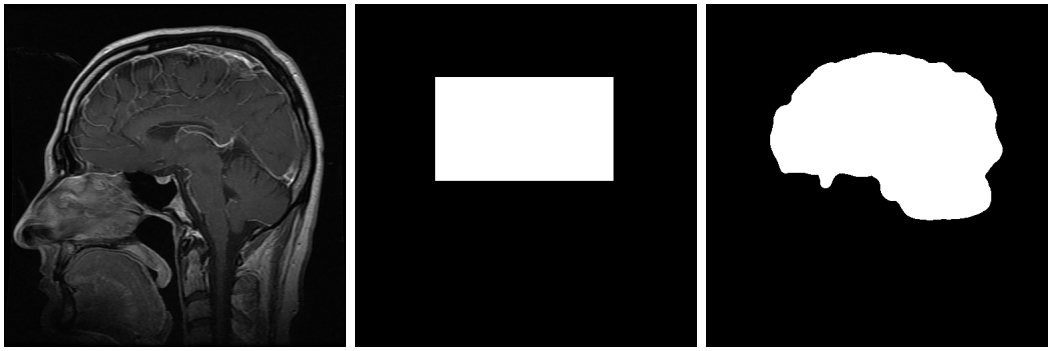
Algorithm Version	Time (s)
MATLAB	425.95
C	55.44
CUDA (Unoptimized)	8.38
CUDA (Optimized)	1.73

Table 4.1: Comparison of runtime for different algorithm versions - 2D liver segmentation

The runtime speed up attained from sequential code in C to CUDA optimized code is approximately $32\times$. The block size used for 2D CUDA compute was 32×8.

In Figure 4.2 we can see the brain in sagittal view. This image is of relatively poor contrast and has dimensions 512×512. This makes the image both a computationally demanding segmentation (as it has relatively large dimensions) and challenging in terms of accuracy. The segmentation inputs and output can be seen in Figure 4.2. It can be seen that the sequential algorithm has performed reasonably in segmenting the white and gray matter and some of the brain stem. Considerable weighting had to be given to curvature in order to prevent leaks due to the poor contrast, resulting in a very rounded segmentation.

The performance speedup attained on this larger image is therefore $46\times$, which is greater than



(a) Feature Image Input - brain (b) Initial Mask Input (c) Output of Segmentation

Figure 4.2: 2D Brain Segmentation with parameters $T = 45, \epsilon = 30, \alpha = 0.003$

Algorithm Version	Time (s)
MATLAB	2737.84
C	322.81
CUDA (Unoptimized)	44.68
CUDA (Optimized)	6.99

Table 4.2: Comparison of runtime for different algorithm versions - 2D brain segmentation

the speed up attained for the smaller 256×256 image. This motivates exploration into the effect of different image sizes on CUDA speed up, which is discussed in Section 4.1.1.

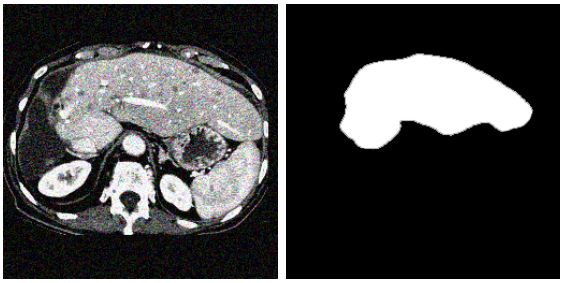
Effect of Noise

Denoising filters already exist as part of the CUDA SDK (i.e. `imageDenoising`). Our algorithm does not feature any image pre-processing algorithms such as denoising or blurring so its performance on noisy images is expected to be poor. In order to test this, artificial Gaussian noise of 20% and 40% was added to the liver image as shown in Figure 4.3.

It can be seen that through manipulating the parameter ϵ segmentations are still approximately valid. The effect of noise on performance of the algorithm was negligible.

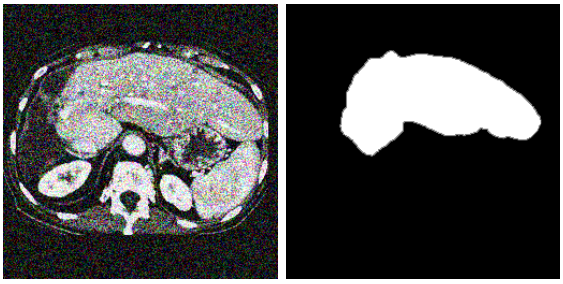
Effect of Different Image Sizes

It is found, as expected, that for all versions of the algorithm (CUDA and single threaded) compute time scales linearly with the number of elements. For square 2D images, this implies that increasing image size by a factor of two in each dimension increases compute time by a factor of four. Figure 4.4(a) displays compute times to 5000 iterations across the different algorithm versions. Both the same input image and mask were used for all tests, again with a block dimension of 32×8.



(a) Liver with 20% Noise

(b) Segmentation with $T = 180, \epsilon = 55, \alpha = 0.003$



(c) Liver with 40% Noise

(d) Segmentation with $T = 180, \epsilon = 75, \alpha = 0.003$

Figure 4.3: Segmentation of the liver with artificially added noise

It can be seen that the difference in compute time between the parallel and sequential versions is greatest for the largest images. This is due to the number of elements computed per second being much greater for the parallel algorithms than for the sequential algorithms, across all image sizes. Interestingly, Figure 4.4(b) shows that the number of elements computed per second is approximately constant for both the sequential and unoptimized CUDA implementations, but not for the optimized shared memory CUDA algorithm. This is due to very low occupancy of the GPU at these small image sizes, resulting in reduced masking of the high latency between device memory and shared memory.

4.1.2 3D Segmentations

Figure 4.5 illustrates a level set surface evolving in 3 dimensions. In order to visualize the level set evolving every certain number of iterations the CUDA SDK example `volumeRender` was modified. This version of the code with visualization of the level set evolution is approximately a factor of 2 slower. This volume rendering engine uses ray tracing which is not advised for segmentation validation, and therefore *Paraview 3.4.0* (www.paraview.org) was used.

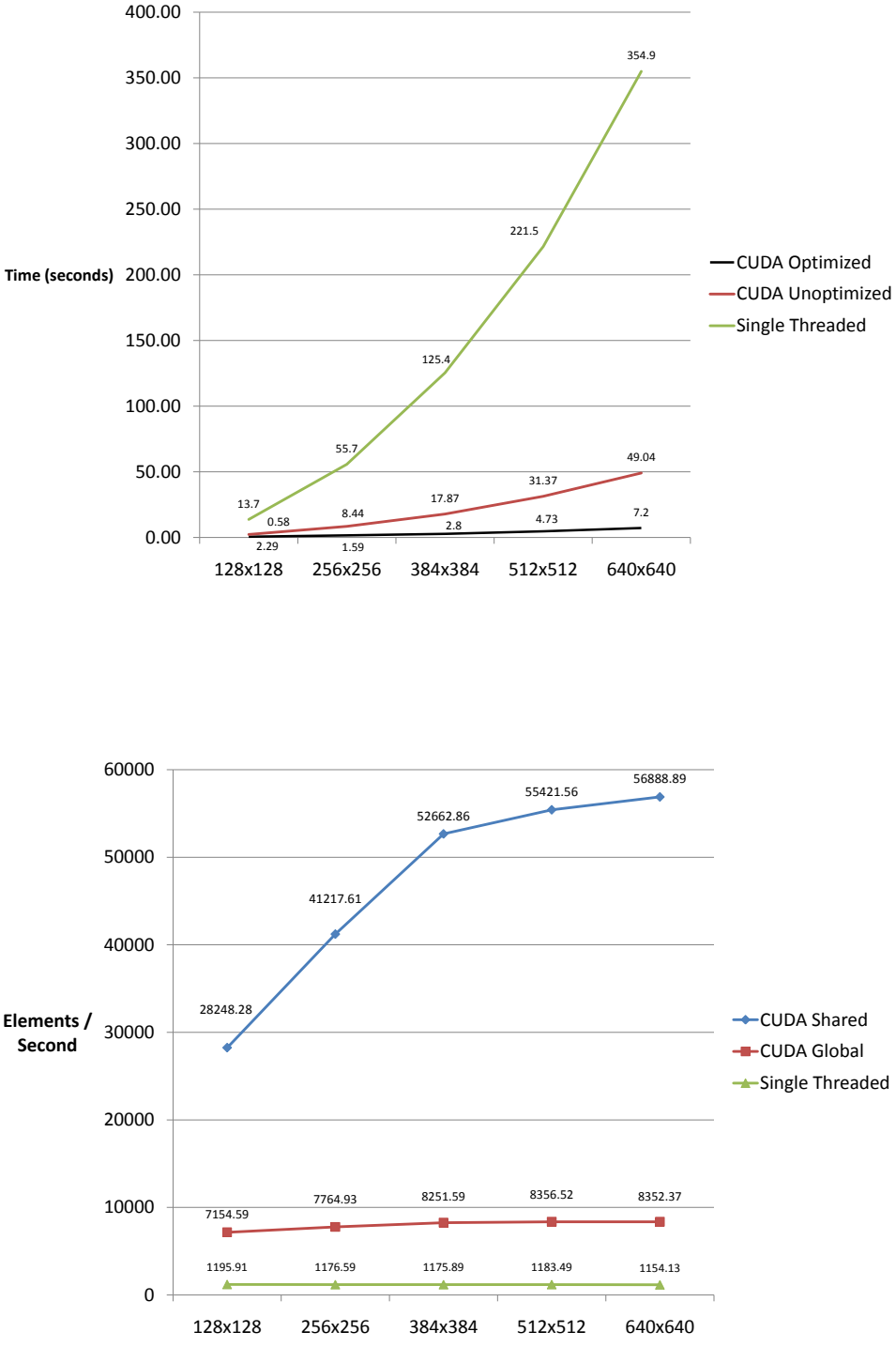


Figure 4.4: Effect of different image sizes (a) On compute time (b) On number of elements computed per second]

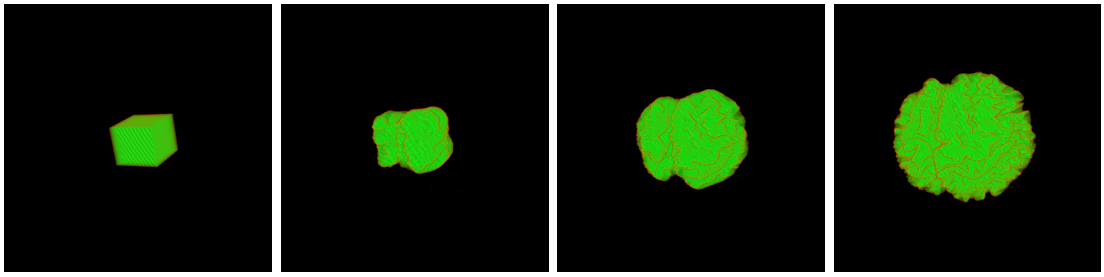


Figure 4.5: Level Set Surface Evolution in 3D at 0, 50, 150 and 200 Iterations

The 3D segmentation CUDA code uses a block size of 32×4. A block size of 32×8 causes the kernel invocation to fail due to the registers used per threads multiplied by the thread block size being greater N (where for G80 NVIDIA hardware $N = 8192$ 32-bit registers per multiprocessor. This limits the extent to which occupancy can be increased to mask latencies due to global memory loads. Section 4.1.2 explores the effect of varying block sizes on performance.

The results from segmenting the cerebral hemispheres, cerebellum and brain stem can be seen in Figure 4.6. The high quality of this segmentation is due in part to the excellent *BrainWeb* MRI data used (data is available from [4]). This data is of size $181 \times 217 \times 181$ and is therefore a good test of the algorithm for reasonably large volumes. As the CFL condition had not been implemented in the 3D level set solver, it was found to converge at 1000 iterations and that DT values greater than 0.1 resulted in instability.

The times taken to segment are shown in Table 4.3. MATLAB performance is not shown as out of memory errors were encountered when loading such large arrays (and even if these had not been encountered, the segmentation would have taken an infeasible amount of time).

Algorithm Version	Time (s)
MATLAB	N/A
C	4697.5
CUDA (Unoptimized)	392.8
CUDA (Optimized)	141.2

Table 4.3: Comparison of runtime for different algorithm versions - 3D Brain segmentation of BrainWeb data [4]

An impressive speed up of 33× is observed. To further demonstrate the power of this algorithm testing was briefly done on a more mid-range 8800 GTX card, observing a speed up of 117× compared to the sequential algorithm.

In Figure 4.7 segmentation of both the right and left ventricles can be seen. This segmentation

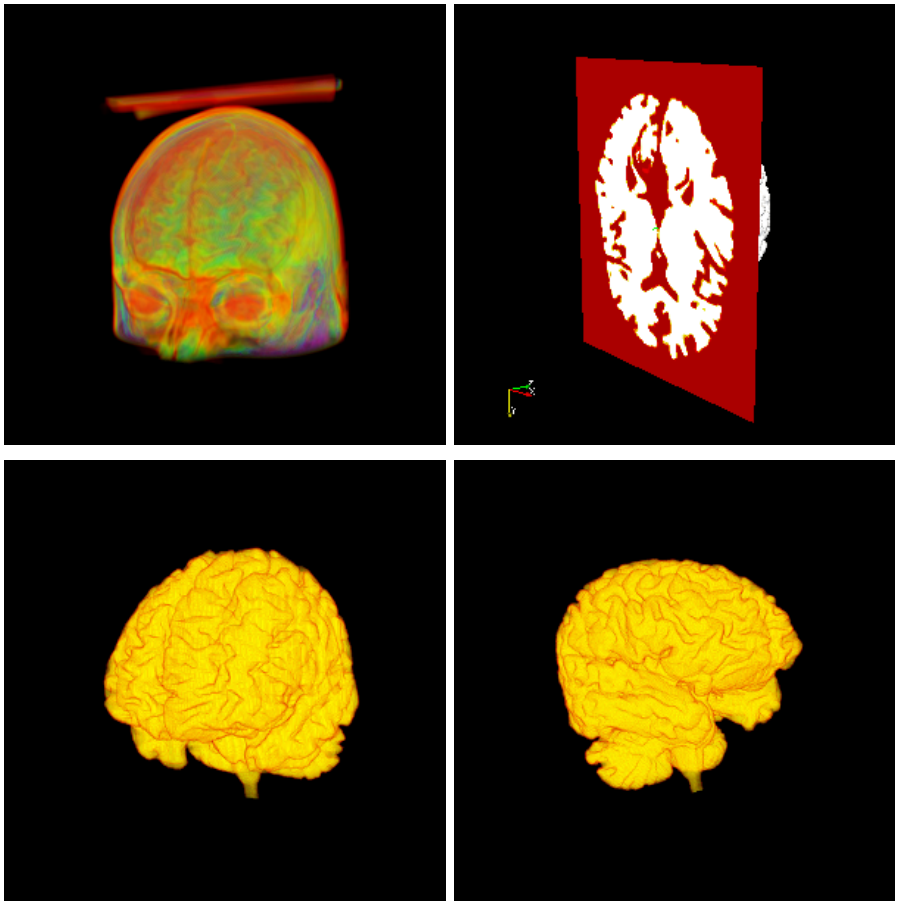
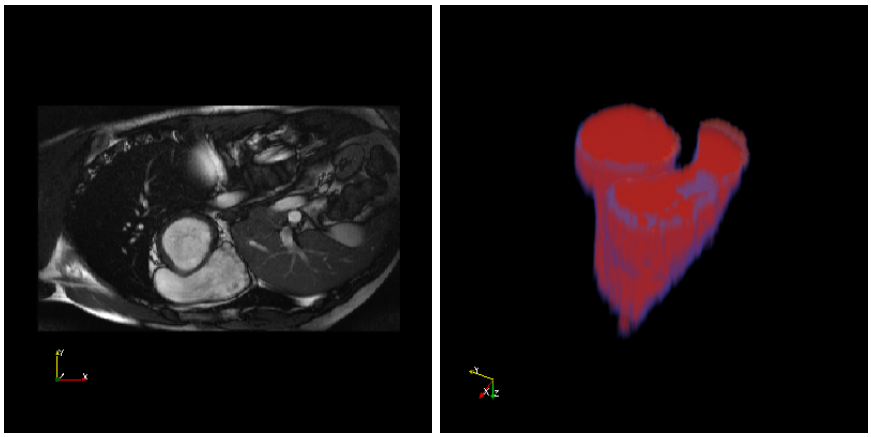


Figure 4.6: Segmentation of a brain MRI dataset with parameters $T = 150, \epsilon = 50, \alpha = 0.03$ MRI data from [4]



(a) (b)

Figure 4.7: Segmentation of the right and left ventricles from a heart MRI dataset with parameters $T = 180, \epsilon = 60, \alpha = 0.02$ (a) Input data slice (b) Segmented heart clipped through z plane

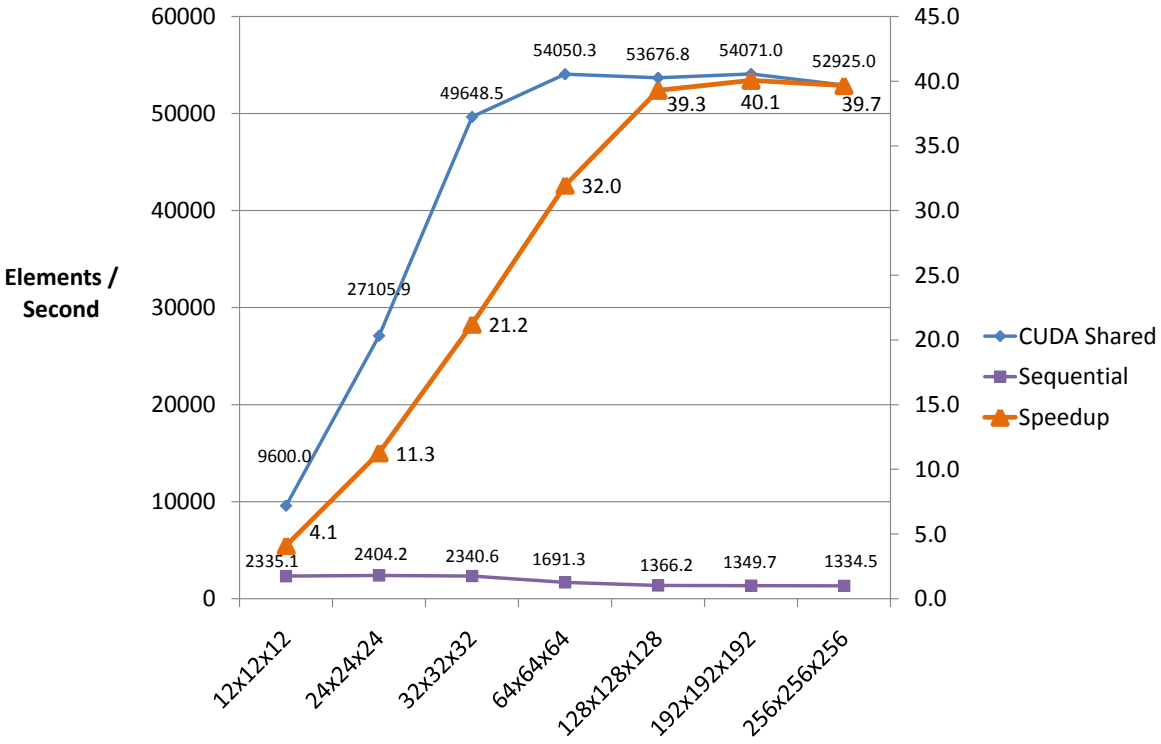


Figure 4.8: Number of elements computed per second for different volume sizes

data only had 17 z plane slices (total resolution $256 \times 160 \times 17$).

Effect of Different Volume Sizes

Figure 4.8 shows the effect of multiple volume sizes on the compute time to 1000 iterations on the optimized CUDA algorithm. Tests were not run on unoptimized CUDA code as this is not of particular interest.

It can be seen from this figure that the speed up at low volume sizes is much smaller than the speed up at larger volume sizes. The sequential algorithm performs almost half as slowly as itself for volume sizes larger than 64^3. This is most likely due to the fast on board CPU cache being used only for volume sizes smaller than this, for volume sizes larger it cannot fit on the CPU cache and so is stored on the slower DRAM. Conversely, the CUDA code performs relatively poorly for small volume sizes and much more quickly for larger ones. This is essentially due to

low numbers of processors being used for such small images and many more being used for larger images. Therefore the speedup line essentially shows that the algorithm follows Amdahl's and Gustafson's laws of parallel computation.

Volume sizes much larger than 256^3 could not be tested as the maximum amount of global memory available for the 8600M GT is 256 MB. For example, a 320^3 volume would take up $320^3 \times$ `sizeof(float)` and there are three of these arrays (for the feature image, previous level set iteration and current level set iteration) which would take up 375 MB of graphics memory. It is however expected, for these even larger volumes, that the speedup of the algorithm will remain approximately plateaued.

Effect of Different CUDA Block/Grid Sizes

Finding the optimum block size for CUDA code is one of the most important ways to optimize performance. Block sizes should not be a user parameter (other than for testing purposes most notably across different GPUs) as it assumed the developer would have chosen the optimum block size for maximum performance for a particular GPU. Figure 4.4 shows the compute times to 1000 iterations for the CUDA optimized code with different block sizes, all other parameters were held constant.

$BX \times BY$	Threads/Block	Time (s)
32×4	128	141.7
16×8	128	141.9
16×12	192	108.4
32×6	192	107.4
48×4	192	106.8

Table 4.4: Comparison of runtime for different block sizes

It can be seen that for blocks with 192 threads performance is approximately constant across the different block arrangements. This is due to the fact that BX has been chosen to be a multiple of 16 to maximize performance, the parameter BY has much less of an effect on performance and should always be set secondary to BX.

The *CUDA Occupancy Calculator* allows the computation of the multiprocessor occupancy that a particular CUDA kernel has on a particular GPU. The resource usage of the 3D shared memory kernel is shown below in Table 4.5. Some of these values were attained by compiling with the `-cubin` option to `nvcc` (NVIDIA CUDA compiler).

Resource	Usage
Threads Per Block	192
Registers Per Thread	39
Shared Memory Per Block (bytes)	5760

Table 4.5: 3D shared memory kernel resource usage - CUDA Occupancy Calculator inputs

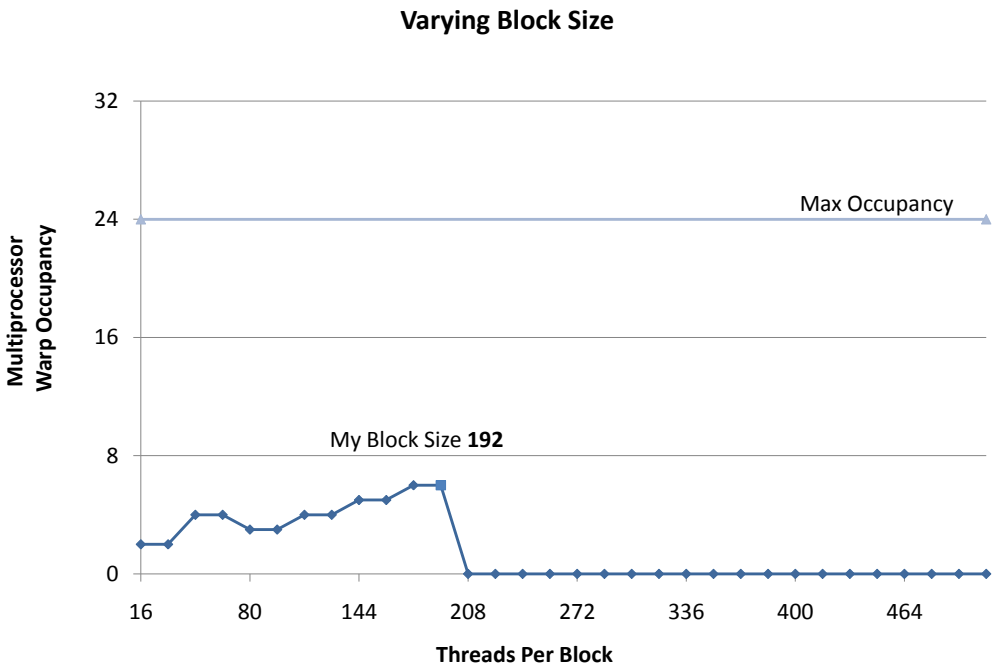


Figure 4.9: The effect of varying the number of threads per block on multiprocessor occupancy

The results from the CUDA Occupancy Calculator are shown below in Figure 4.9. The kernel uses a large number of registers due to its complexity, and this is what is limiting the maximum number of threads per block to 192. Having said that, adjusting the kernel to use fewer registers, and thus attain higher occupancy, may not necessarily yield higher performance due to the potential introduction of other effects such as additional instructions, spills to device memory and divergent branches.

4.2 Discussion and Limitations

4.2.1 Speed

This algorithm does not currently use a narrow band, introduced in Section 3.1.1, to update the level set in either the sequential or parallel versions. If it were introduced into the CUDA kernel, the complexity of the kernel would increase further, increasing the number of registers used. Therefore the performance gains on the parallel algorithm would not be as great in comparison to the gains on the sequential algorithm.

Comparison of CPU and GPU code was done with algorithms that most closely mirrored each other. Although this standardizes the code, it does distort the results slightly as there is potential for optimization on the CPU by making effective use of the CPU cache, and multiple cores (if present).

Goodman [7] shows that CPU code may be slowed if a GPU kernel is executed and therefore suggests that CPU and GPU code run in separate independent environments. To this effect, this has been catered for, increasing the accuracy of the speed up figures attained.

Furthermore, making comparisons with MATLAB code is inadvisable given the environment that MATLAB code runs. MATLAB code is JIT compiled, creating many problems when directly comparing compute times between the two versions. Speed ups were for this reason not measured against MATLAB code. In fact, the main purpose of the MATLAB code was to learn about the inner workings of level set segmentation.

4.2.2 Accuracy

Firstly, the nature of segmenting images using thresholding and curvature terms favours segmentations of anatomical objects with a relatively homogeneous gray value range. Therefore, this algorithm performs best when delineating more homogeneous anatomical structures as can be seen with the detailed brain segmentation shown in Figure 4.6.

Secondly, as discussed in Section 2.2.2 the current 3D level set segmentation solver does not integrate a reinitialization algorithm. This is the largest limitation of this algorithm as it may result in instabilities in the level set function if $\nabla\phi$ values get too large. That said, almost no instabilities were found during testing. It should be noted that when implementing a 3D distance transform there is a major trade off between accuracy and speed.

Chapter 5

Conclusions and Future Work

5.1 Conclusion

In this project report a fast segmentation algorithm has been presented and analyzed. The implementation on the graphics device is very fast, with large two dimensional images and three dimensional volumes segmented 30 to 40 times faster (on a relatively low performance GPU) than sequential algorithms. The method of using level sets to segment images, and how to accelerate this process using GPUs has been discussed in great detail. The numerical methods used for the implementation were listed in Section 3.1. Giles' CUDA kernel for Laplace discretization in 3D [6] has been adapted for level set iteration. In Section 4.1 it was seen that the power of GPU acceleration was demonstrated for very large data sets.

Given the wide range of applications level sets have in computing (image processing, computer graphics and physical simulation) this algorithm serves as an excellent framework to solve a diverse array of problems.

CUDA itself has been shown to be an excellent framework to accelerate computational problems in engineering, and is gaining more features and fewer limitations every few months. The principal disadvantages of CUDA are that it is only effective for very data parallel problems, and that it is not an industry standard. Recently, to counter the latter, it is very likely that it will in fact be replaced by *OpenCL* (Open Computing Language). The syntax and architecture between CUDA and OpenCL will be very similar, allowing this code to be easily ported to OpenCL.

Nonetheless the impressive speedups attained using such low end hardware demonstrate the power of this parallel segmentation algorithm, and this makes segmentation with large 3D volumes

much more practical in a clinical setting.

5.2 Future Work

There are several areas in which this algorithm could be extended. These revolve around three central themes of speed, accuracy and usability.

In terms of speed, integrating the narrow band method into the algorithm will provide some further speed up, this however increases the complexity of the kernel potentially resulting in higher register usage and less occupancy. This, in combination to adding support for multiple GPUs and testing on very high performance hardware would be of significant interest.

In terms of accuracy, it would be interesting, and reasonably straightforward, to integrate some of the already coded CUDA image processing examples (such as denoising, blurring, sharpening examples) to form a modular CUDA image processing and segmentation library that clinicians could work with. This could be included with a modular graphical user interface to make a very robust, usable and fast image processing library.

Bibliography

[1] David Adalsteinsson and James A. Sethian. A fast level set method for propagating interfaces. *Journal of Computational Physics*, 118:269–277, 1994.

[2] J.E. Cates, A.E. Lefohn, and R.T. Whitaker. GIST: an interactive, GPU-based level set segmentation tool for 3D medical images. *Medical Image Analysis*, 8(3):217–231, 2004.

[3] T.F. Chan and L.A. Vese. Active contours without edges. *IEEE Transactions on image processing*, 10(2):266–277, 2001.

[4] Chris A. Cocosco, Vasken Kollokian, Remi K.-S. Kwan, G. Bruce Pike, and Alan C. Evans. Brainweb: Online interface to a 3D MRI Simulated Brain Database. *NeuroImage*, 5:425, 1997.

[5] Kent Roger B. Fagerjord and Tatyana V. Lochehina. *GPGPU: Fast and easy Distance Field computation on GPU.* 2008.

[6] Mike Giles. Jacobi iteration for a laplace discretisation on a 3d structured grid. 2008.

[7] Daniel Goodman. Measuring speed ups on graphics cards. 2009.

[8] C.L.C.X.C. Gui and MD Fox. Level set evolution without re-initialization: a new variational formulation. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2005. CVPR 2005*, volume 1, 2005.

[9] L. Ibanez, W. Schroeder, L. Ng, and J. Cates. The ITK Software Guide. Kitware, Inc.

[10] M. Kass, A. Witkin, and D. Terzopoulos. Snakes: Active contour models. *International journal of computer vision*, 1(4):321–331, 1988.

[11] A. Kharlamov and V. Podlozhnyuk. Image Denoising.

[12] O. Klar. Interactive GPU-based Segmentation of Large Medical Volume Data with Level-Sets.

[13] Aaron E. Lefohn, Joe M. Kniss, Charles D. Hansen, and Ross T. Whitaker. A streaming narrow-band algorithm: Interactive computation and visualization of level sets. *IEEE Transactions on Visualization and Computer Graphics*, 10:422–433, 2004.

[14] Paulius Micikevicius. 3D finite difference computation on GPUs using CUDA. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84, New York, NY, USA, 2009. ACM.

[15] H. Nguyen. GPU gems 3. 2007.

[16] NVIDIA. Compute Unified Device Architecture–Programming Guide, 2008.

[17] NVIDIA. Compute Unified Device Architecture–Reference Manual, 2008.

[18] S. Osher and R.P. Fedkiw. *Level set methods and dynamic implicit surfaces*. Springer, 2003.

[19] Stanley Osher and James A. Sethian. Fronts propagating with curvature dependent speed: algorithms based on hamilton-jacobi formulations. *Journal of Computational Physics*, 79:12–49, 1988.

[20] M. Rumpf and R. Strzodka. Level set segmentation in graphics hardware. In *Image Processing, 2001. Proceedings. 2001 International Conference on*, volume 3, 2001.

[21] J.A. Sethian et al. Level set methods and fast marching methods. *Journal of Computing and Information Technology*, 11(1):1–2, 2003.

[22] O. Sharma and F. Anton. CUDA based Level Set Method for 3D Reconstruction of Fishes from Large Acoustic Data. In *International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 17*.

[23] A. Sud, M.A. Otaduy, and D. Manocha. DiFi: Fast 3D distance field computation using graphics hardware. In *Computer Graphics Forum*, volume 23, pages 557–566. Blackwell Publishing, 2004.

[24] Ross T. Whitaker. A Level-Set Approach to 3D Reconstruction from Range Data. *Int. J. Comput. Vision*, 29(3):203–231, 1998.

Appendix A

MATLAB 2D Code

Note: Full versions of all source code, and their revisions, can be found at `cudaseg.googlecode.com`

```
function seg = simpleseg(I,init_mask,max_its,E,T,alpha)

%-- Create a signed distance map (SDF) from mask
phi=bwdist(init_mask)-bwdist(1-init_mask)-.5;

%main loop
for its = 1:max_its

    D = E - abs(I - T);
    K = get_curvature(phi);
    F = -alpha*D + (1-alpha)*K;

    dxplus=shiftR(phi)-phi;
    dyplus=shiftU(phi)-phi;
    dxminus=phi-shiftL(phi);
    dyminus=phi-shiftD(phi);

    gradphimax_x = sqrt(max(dxplus,0).^2+max(-dxminus,0).^2);
    gradphimin_x = sqrt(min(dxplus,0).^2+min(-dxminus,0).^2);
    gradphimax_y = sqrt(max(dyplus,0).^2+max(-dyminus,0).^2);
    gradphimin_y = sqrt(min(dyplus,0).^2+min(-dyminus,0).^2);
    gradphimax = sqrt((gradphimax_x.^2)+(gradphimax_y.^2));
    gradphimin = sqrt((gradphimin_x.^2)+(gradphimin_y.^2));
    gradphi=(F>0).*(gradphimax) + (F<0).*(gradphimin);

    %stability CFL
    dt = .5/max(max(abs(F.*gradphi)));

    %evolve the curve
    phi = phi + dt.*(F).*gradphi;

    %reinitialise distance funciton every 50 iterations
    if(mod(its,50) == 0)
```

```
        phi=bwdist(phi<0)-bwdist(phi>0);
    end

    %intermediate output
    if(mod(its,20) == 0)
        showcontour(I,phi,its);
        subplot(2,2,4);  surf(phi); shading flat;
    end
end

%make mask from SDF
seg = phi<=0; %-- Get mask from levelset

%-- whole matrix derivatives
function shift = shiftD(M)
shift = shiftR(M')';

function shift = shiftL(M)
shift = [ M(:,2:size(M,2)) M(:,size(M,2)) ];

function shift = shiftR(M)
shift = [ M(:,1) M(:,1:size(M,2)-1) ];

function shift = shiftU(M)
shift = shiftL(M')';

function curvature=get_curvature(phi)
dx=(shiftR(phi)-shiftL(phi))/2;
dy=(shiftU(phi)-shiftD(phi))/2;
dxplus=shiftR(phi)-phi;
dyplus=shiftU(phi)-phi;
dxminus=phi-shiftL(phi);
dyminus=phi-shiftD(phi);
dxplusy =(shiftU(shiftR(phi))-shiftU(shiftL(phi)))/2;
dyplusx =(shiftR(shiftU(phi))-shiftR(shiftD(phi)))/2;
dxminusy=(shiftD(shiftR(phi))-shiftD(shiftL(phi)))/2;
dyminusx=(shiftL(shiftU(phi))-shiftL(shiftD(phi)))/2;

nplusx = dxplus./sqrt(eps+(dxplus.^2 )+((dyplusx+dy )/2).^2);
nplusy = dyplus./sqrt(eps+(dyplus.^2 )+((dxplusy+dx )/2).^2);
nminusx= dxminus./sqrt(eps+(dxminus.^2)+((dyminusx+dy)/2).^2);
nminusy= dyminus./sqrt(eps+(dyminus.^2)+((dxminusy+dx)/2).^2);

curvature=((nplusx-nminusx)+(nplusy-nminusy)/2);

%-- Displays the image with curve superimposed
function showcontour(I, phi, i)
subplot(2,2,3); title('Evolution');
imshow(I,'initialmagnification',200,'displayrange',[0 255]);
hold on;
contour(phi, [0 0], 'g','LineWidth',2);
hold off; title([num2str(i) ' Iterations']); drawnow;
```

Appendix B

CUDA 3D Kernel Source Code

```
#define ALPHA           0.03 //Weighting term between data and curvature
#define DT              0.1  //Time step

#define max(x,y)    ((x>y) ? x : y )
#define min(x,y)    ((x<y) ? x : y )

#define INDEX(i,j,j_off)  (i +__mul24(j,j_off))

#define BLOCKDIM_X      32
#define BLOCKDIM_Y      4
#define BLOCKDIM_Z      1

__global__ void updatephi( float *d_phi, float *d_phi1, float *d_D,
                int imageW, int imageH, int imageD, int pitch)
{

float dx,dy,dz;
float dxplus, dyplus, dzplus, dxminus, dyminus, dzminus;
float dxplusy, dxminusy, dxplusz, dxminusz, dyplusx, dyminusx,
      dyplusz, dyminusz, dzplusx, dzminusx, dzplusy, dzminusy;

float gradphimax, gradphimin;
float nplusx, nplusy, nplusz, nminusx, nminusy, nminusz, curvature;
float F, gradphi;

//M. GILES CUDA TILING CODE

int   indg, indg_h, indg0;
int   i, j, k, ind, ind_h, halo, active;

int IOFF  = 1;
int JOFF  = (BLOCKDIM_X+2);
int KOFF  = (BLOCKDIM_X+2)*(BLOCKDIM_Y+2);

__shared__ float s_data[3*(BLOCKDIM_X+2)*(BLOCKDIM_Y+2)]; //Allocate Shared Memory

k    =  threadIdx.y*BLOCKDIM_X + threadIdx.x;
```

```
halo = k < 2*(BLOCKDIM_X+BLOCKDIM_Y+2);

if (halo) {
        if (threadIdx.y<2) {                    // y-halos (coalesced)
                i = threadIdx.x;
                j = threadIdx.y*(BLOCKDIM_Y+1) - 1;
        }
        else {                                  // x-halos (not coalesced)
                i = (k%2)*(BLOCKDIM_X+1) - 1;
                j =  k/2 - BLOCKDIM_X - 1;
        }

        ind_h  = INDEX(i+1,j+1,BLOCKDIM_X+2)+KOFF;

        i      = INDEX(i,blockIdx.x,BLOCKDIM_X);    // global indices
        j      = INDEX(j,blockIdx.y,BLOCKDIM_Y);
        indg_h = INDEX(i,j,pitch);

        halo   =  (i>=0) && (i<imageW) && (j>=0) && (j<imageH);
}

//
// then set up indices for main block
//

i    = threadIdx.x;
j    = threadIdx.y;
ind  = INDEX(i+1,j+1,BLOCKDIM_X+2) ;

i    = INDEX(i,blockIdx.x,BLOCKDIM_X);      // global indices
j    = INDEX(j,blockIdx.y,BLOCKDIM_Y);
indg = INDEX(i,j,pitch);

active = (i<imageW) && (j<imageH);

//
// read initial plane of u1 array
//

if (active) s_data[ind+KOFF+KOFF] = d_phi1[indg];
if (halo) s_data[ind_h+KOFF+KOFF] = d_phi1[indg_h];


for(int k=0;k<imageD;k++){ //Loop over z dimension

if (active) {
        indg0 = indg;
        indg  = INDEX(indg,imageH,pitch);
        s_data[ind-KOFF] = s_data[ind];
        s_data[ind]      = s_data[ind+KOFF];
        if (k<imageD-1)
                s_data[ind+KOFF] = d_phi1[indg];
```

```
}

if (halo) {
        indg_h = INDEX(indg_h,imageH,pitch);
        s_data[ind_h-KOFF] = s_data[ind_h];
        s_data[ind_h]      = s_data[ind_h+KOFF];
        if (k<imageD-1)
                s_data[ind_h+KOFF] = d_phi1[indg_h];
}
//M. GILES CUDA TILING CODE END

if (active) {

if(i==0||i==imageW-1){dx=0;} //Calculate First Order Derivatives
else {dx=(s_data[ind+IOFF]-s_data[ind-IOFF])/2;}
if(j==0||j==imageH-1){dy=0;}
else {dy=(s_data[ind-JOFF]-s_data[ind+JOFF])/2;}
if(k==0||k==imageD-1){dz=0;}
else {dz=(s_data[ind+KOFF]-s_data[ind-KOFF])/2;}

if(i==imageW-1){dxplus=0;}
else {dxplus =(s_data[ind+IOFF]-s_data[ind     ]);}
if(j==0){dyplus=0;}
else {dyplus =(s_data[ind-JOFF]-s_data[ind     ]);}
if(k==imageD-1){dzplus=0;}
else {dzplus =(s_data[ind+KOFF]-s_data[ind     ]);}
if(i==0){dxminus=0;}
else {dxminus=(s_data[ind     ]-s_data[ind-IOFF]);}
if(j==imageH-1){dyminus=0;}
else {dyminus=(s_data[ind     ]-s_data[ind+JOFF]);}
if(k==0){dzminus=0;}
else {dzminus=(s_data[ind     ]-s_data[ind-KOFF]);}

if(i==0||i==imageW-1||j==0){dxplusy=0;} //Calculate Cross Derivatives
else {dxplusy =(s_data[ind-JOFF+IOFF]-s_data[ind-JOFF-IOFF])/2;}
if(i==0||i==imageW-1||j==imageH-1){dxminusy=0;}
else {dxminusy=(s_data[ind+JOFF+IOFF]-s_data[ind+JOFF-IOFF])/2;}
if(i==0||i==imageW-1||k==imageD-1) {dxplusz=0;}
else {dxplusz =(s_data[ind+KOFF+IOFF]-s_data[ind+KOFF-IOFF])/2;}
if(i==0||i==imageW-1||k==0) {dxminusz=0;}
else {dxminusz=(s_data[ind-KOFF+IOFF]-s_data[ind-KOFF-IOFF])/2;}
if(j==0||j==imageH-1||i==imageW-1){dyplusx=0;}
else {dyplusx =(s_data[ind-JOFF+IOFF]-s_data[ind+JOFF+IOFF])/2;}
if(j==0||j==imageH-1||i==0){dyminusx=0;}
else {dyminusx=(s_data[ind-JOFF-IOFF]-s_data[ind+JOFF-IOFF])/2;}
if(j==0||j==imageH-1||k==imageD-1) {dyplusz=0;}
else {dyplusz =(s_data[ind+KOFF-JOFF]-s_data[ind+KOFF+JOFF])/2;}
if(j==0||j==imageH-1||k==0) {dyminusz=0;}
else {dyminusz=(s_data[ind-KOFF-JOFF]-s_data[ind-KOFF+JOFF])/2;}
if(k==0||k==imageD-1||i==imageW-1) {dzplusx=0;}
else {dzplusx =(s_data[ind+IOFF+KOFF]-s_data[ind+IOFF-KOFF])/2;}
if(k==0||k==imageD-1||i==0) {dzminusx=0;}
```

```
else {dzminusx=(s_data[ind-IOFF+KOFF]-s_data[ind-IOFF-KOFF])/2;}
if(k==0||k==imageD-1||j==0) {dzplusy=0;}
else {dzplusy =(s_data[ind-JOFF+KOFF]-s_data[ind-JOFF-KOFF])/2;}
if(k==0||k==imageD-1||j==imageH-1) {dzminusy=0;}
else {dzminusy=(s_data[ind+JOFF+KOFF]-s_data[ind+JOFF-KOFF])/2;}

//Calculate grad phi max/min using macro defined above
gradphimax=sqrt((sqrt(max(dxplus,0)*max(dxplus,0)+max(-dxminus,0)*max(-dxminus,0)))
        *(sqrt(max(dxplus,0)*max(dxplus,0)+max(-dxminus,0)*max(-dxminus,0)))
        +(sqrt(max(dyplus,0)*max(dyplus,0)+max(-dyminus,0)*max(-dyminus,0)))
        *(sqrt(max(dyplus,0)*max(dyplus,0)+max(-dyminus,0)*max(-dyminus,0)))
        +(sqrt(max(dzplus,0)*max(dzplus,0)+max(-dzminus,0)*max(-dzminus,0)))
        *(sqrt(max(dzplus,0)*max(dzplus,0)+max(-dzminus,0)*max(-dzminus,0))));

gradphimin=sqrt((sqrt(min(dxplus,0)*min(dxplus,0)+min(-dxminus,0)*min(-dxminus,0)))
        *(sqrt(min(dxplus,0)*min(dxplus,0)+min(-dxminus,0)*min(-dxminus,0)))
        +(sqrt(min(dyplus,0)*min(dyplus,0)+min(-dyminus,0)*min(-dyminus,0)))
        *(sqrt(min(dyplus,0)*min(dyplus,0)+min(-dyminus,0)*min(-dyminus,0)))
        +(sqrt(min(dzplus,0)*min(dzplus,0)+min(-dzminus,0)*min(-dzminus,0)))
        *(sqrt(min(dzplus,0)*min(dzplus,0)+min(-dzminus,0)*min(-dzminus,0))));

//Calculate Curvature Terms
nplusx = dxplus / sqrt(1.192092896e-07F + (dxplus*dxplus)
        + ((dyplusx + dy)*(dyplusx + dy)*0.25)
        + ((dzplusx + dz)*(dzplusx + dz)*0.25));
nplusy = dyplus / sqrt(1.192092896e-07F + (dyplus*dyplus)
        + ((dxplusy + dx)*(dxplusy + dx)*0.25)
        + ((dzplusy + dz)*(dzplusy + dz)*0.25));
nplusz = dzplus / sqrt(1.192092896e-07F + (dzplus*dzplus)
        + ((dxplusz + dz)*(dxplusz + dz)*0.25)
        + ((dyplusz + dy)*(dyplusz + dy)*0.25));

nminusx=dxminus / sqrt(1.192092896e-07F + (dxminus*dxminus)
        + ((dyminusx + dy)*(dyminusx + dy)*0.25)
        + ((dzminusx + dz)*(dzminusx + dz)*0.25));
nminusy=dyminus / sqrt(1.192092896e-07F + (dyminus*dyminus)
        + ((dxminusy + dx)*(dxminusy + dx)*0.25)
        + ((dzminusy + dz)*(dzminusy + dz)*0.25));
nminusz=dzminus / sqrt(1.192092896e-07F + (dzminus*dzminus)
        + ((dxminusz + dz)*(dxminusz + dz)*0.25)
        + ((dyminusz + dy)*(dyminusz + dy)*0.25));

curvature = ((nplusx-nminusx)+(nplusy-nminusy)+(nplusz-nminusz))/2;

F = (-ALPHA * d_D[indg0]) + ((1-ALPHA) * curvature); //Calculate Speed Term F
if(F>0) {gradphi=gradphimax;} else {gradphi=gradphimin;} //Construct Grad Phi
d_phi[indg0]=s_data[ind] + (DT * F * gradphi); //Update Level Set Function

}
__syncthreads(); //Thread Barrier Synchronization
}
}
```

Appendix C

CUDA 3D Main Source Code

```
//
// CUDA 3D Level Set Segmentation
// Coded by: Hormuz Mostofi, University of Oxford
//
// Example Usage:
//       Seg -volume=input.raw -mask=mask.raw -xsize=256 -ysize=256 -zsize=256
//       -iterations=1000 -threshold=150 -epsilon=50 -alpha=0.01
//

#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cutil.h>

#define BLOCKDIM_X      32
#define BLOCKDIM_Y      6
#define BLOCKDIM_Z      1

char *volumeFilename, *maskFilename;
int     ITERATIONS, THRESHOLD, EPSILON;
float alpha;
int imageW, imageH, imageD, N, pitch;

float *phi, *D;
size_t size, pitchbytes;
unsigned char *input,*output;

float *d_phi, *d_phi1, *d_D;

int its=0;
unsigned int Timer = 0;

int i,j,k;

__global__ void updatephi( float *d_phi, float *d_phi1, float *d_D,
        int imageW, int imageH, int imageD, float alpha, int pitch);
```

```
// loadRawUchar, loadMask, and writeoutput functions not of interest

void cuda_update(){

dim3 dimGrid( ((imageW-1)/BLOCKDIM_X) + 1, ((imageH-1)/BLOCKDIM_Y) +1);
dim3 dimBlock(BLOCKDIM_X, BLOCKDIM_Y, BLOCKDIM_Z);

updatephi<<< dimGrid, dimBlock>>>(d_phi, d_phi1, d_D,
        imageW, imageH, imageD, alpha, pitch);
d_phi1=d_phi;

CUT_CHECK_ERROR("Kernel execution failed\n");
CUDA_SAFE_CALL(cudaThreadSynchronize());
}

int main(int argc, char** argv){

// Ensure all parameters are set
if(argc<9){
        printf("Too few command line arguments specified\n");
        exit(0);
}

// Parse Command Line Arguments
cutGetCmdLineArgumentstr( argc, (const char**) argv, "volume", &volumeFilename);
cutGetCmdLineArgumentstr( argc, (const char**) argv, "mask", &maskFilename);
cutGetCmdLineArgumenti( argc, (const char**) argv, "xsize", &imageW);
cutGetCmdLineArgumenti( argc, (const char**) argv, "ysize", &imageH);
cutGetCmdLineArgumenti( argc, (const char**) argv, "zsize", &imageD);
cutGetCmdLineArgumenti( argc, (const char**) argv, "iterations", &ITERATIONS);
cutGetCmdLineArgumenti( argc, (const char**) argv, "threshold", &THRESHOLD);
cutGetCmdLineArgumenti( argc, (const char**) argv, "epsilon", &EPSILON);
cutGetCmdLineArgumentf( argc, (const char**) argv, "alpha", &alpha);

// Initialise Feature Image and Mask on Host
N=imageW*imageH*imageD;
input = loadRawUchar(volumeFilename, N);
phi = loadMask(maskFilename, N);

// Calculate D(I) = E - | I - T |
if((D = (float *)malloc(imageW*imageH*imageD*sizeof(float)))==NULL)printf("ME_D\n");
for(i=0;i<N;i++){
        D[i] = EPSILON - abs((unsigned char)input[i] - THRESHOLD);
}

// Set up CUDA Timer
cutCreateTimer(&Timer);

// Allocate Memory on Device
CUDA_SAFE_CALL( cudaMallocPitch((void**)&d_D, &pitchbytes,
        sizeof(float)*imageW, imageH*imageD));
CUDA_SAFE_CALL( cudaMallocPitch((void**)&d_phi,&pitchbytes,
```

```
        sizeof(float)*imageW, imageH*imageD));
CUDA_SAFE_CALL( cudaMallocPitch((void**)&d_phi1, &pitchbytes,
        sizeof(float)*imageW, imageH*imageD));

pitch=pitchbytes/sizeof(float);

// Copy Host Data to Device Memory
CUDA_SAFE_CALL( cudaMemcpy2D(d_D,pitchbytes, D, sizeof(float)*imageW,
        sizeof(float)*imageW, imageH*imageD, cudaMemcpyHostToDevice));
CUDA_SAFE_CALL( cudaMemcpy2D(d_phi1, pitchbytes, phi, sizeof(float)*imageW,
        sizeof(float)*imageW, imageH*imageD, cudaMemcpyHostToDevice));

// Start Timer
cutStartTimer(Timer);

// Iterate Level Set Solver
for(its=0;its<=ITERATIONS;its++){
        cuda_update();
        if(its%50==0){
        printf("Iteration %3d Time: %3.2f\n",
                its, 0.001*cutGetTimerValue(Timer),);}
}

// Stop Timer
cutStopTimer(Timer);

// Write Result Back to Host Memory
cudaMemcpy2D(phi, sizeof(float)*imageW, d_phi1, pitchbytes,

writeoutput(phi, N);

// Free Memory
CUDA_SAFE_CALL( cudaFree(d_phi) );
CUDA_SAFE_CALL( cudaFree(d_phi1) );
CUDA_SAFE_CALL( cudaFree(d_D) );
free(D);
free(phi);
free(input);

}
```

Department of Eng Science

4th Year Project Risk Assessment

Form RA01 16/10/03

Risk Assessment	4th Year Project – Fast Level Set Segmentation of Biomedical Images using GPUs			Page 1 of 1
In Building	Oxford E-Research Centre			
Assessment undertaken	Hormuz Mostofi	Signed	Date :	18th November 2008
Assessment supervisor	Julia Schnabel	Signed	Date :	18th November 2008

Hazard	Persons at Risk	Risk Controls In Place	Further Action Necessary To Control Risk
Headache and eyestrain	User	• Taking regular breaks of approx 5 minutes per hour • Calibrating screen properly (brightness and flicker prevention) • Ensuring there is adequate ambient room lighting	Consult Supervisor and advise Departmental Safety Officer if problems persist
Back pain	User	• Adjusting chair properly and ensuring correct seating posture	Consult Supervisor
Upper limb pain	User	• Ensure correct seat height • Use wrist rest • Forearms horizontal and level with desk surface	Consult Supervisor
Lower limb pain	User	• Allow room for legs under desk and/or place footrest	Consult Supervisor
Electrical shock from computing equipment	User	• Ensure all electrical equipment is PAT tested by Electronics	Consult Supervisor to check validity of PAT test and label

Your E-mail Address ……Checked by ..... date ……..

(D J Reed)