Fast Level Set Segmentation of Biomedical Images using Graphics Processing Units

Hormuz Mostofi

April 21, 2009

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 provides a trade off between making algorithms more generalized and having many user specified parameters. Secondly, imaging artificats such as noise, inhomogeneity, acquisition artifacts and low 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.

Image segmentation also encompasses three dimensional volume segmentations, which are slower to compute by several orders of magnitude. It should be noted that before such algorithms existed, segmentation of medical images was done by hand by experts. This was a very accurate, yet slow, process. These segmentations will form the gold standard with which to validate algorithmic segmentations.

In this report, the level set method is used for the purposes of segmentation. Their principal disadvantage is that they are relatively slow to compute, which provides the motivation for optimizing and accelerating such algorithms using graphics

processing units (GPUs). Section 2.1 discusses them in great detail.

1.2 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.

1.2.1 GPGPU

General purpose computation on graphics processing units (GPGPU) is the technique of using graphics hardware to compute on applications typically handled by the central processing unit (CPU). Graphics cards over the past two decards have been required to render increasingly complex 3D scenes at high frame rates, which is in itself a highly parallelizable task computationally.

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. It is not optimized for general purpose programs, and does not have the complex instruction sets, or branch control of the modern CPU. It should be noted however that CPUs are slowly being parallelized by featuring multiple cores on a single chip.

The advent of GPGPU programming came with programmable shader units that allowed

1.2.2 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 [5]: a heirachy of thread groups. shared memories, and barrier synchronization. This breaks the task of parallelization into three sub problems, which allows for languaging expressivity when threads cooperate, and scalability when extended to multiple processor cores.

Framework

CUDA uses extends C by allowing a programming to write *kernels* that when invoked execute a 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 threads is easily assigned to an individual pixel or voxel.

When writing and optimizing complex parallel code in CUDA it is often found that threads may need to cooperate. 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. There are also constant and texture memory spaces accessible to all threads, however these are not utilised in this reports 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 maximise efficiency, there is a trade off between making the occupancy very high, ensuring no multiprocessor is ever idle, and making it low enough to ensure no bank conflicts.

Secondly, one of the best ways in which to optimize the parallelization is through efficient shared 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 minimise global memory accesses. From [5], 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.

Coalescence

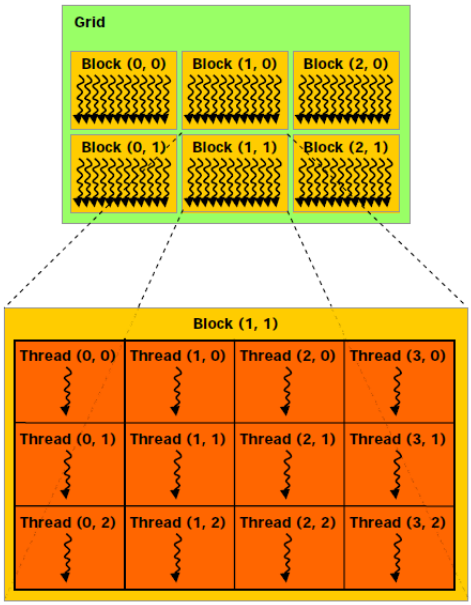


Figure 1.1: A grid of thread blocks. This figure taken from [5]

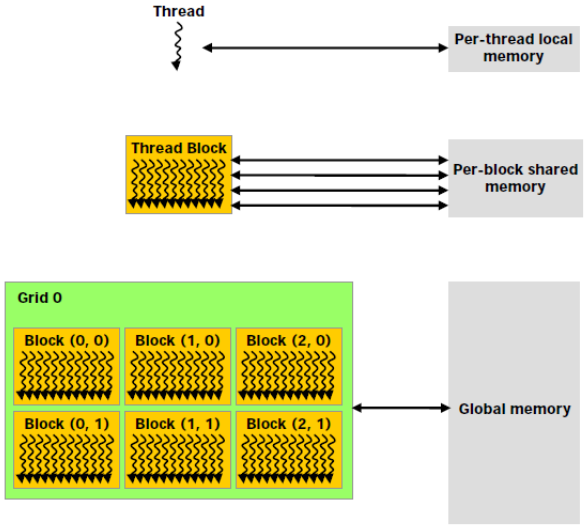


Figure 1.2: The memory heirarchy of CUDA threads and blocks. This figure taken from [5]

Chapter 2

Method

In this section, we introduce the level set method and dynamic implicit surfaces. Their role in segmentation is discussed having introduced having 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 [7], 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 converge the level set to different areas or shapes, given an initialisation of the level set function.

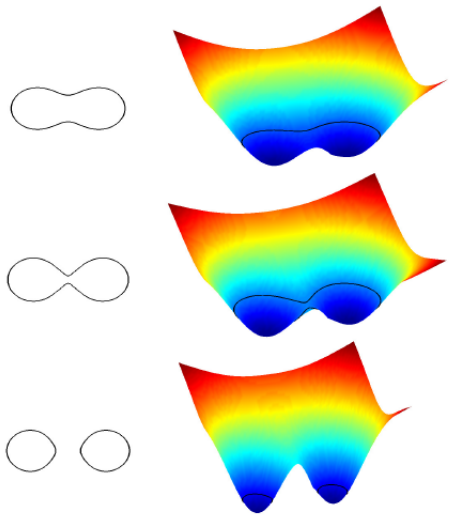


Figure 2.1: The relationship between the level set function (left) and contour (right) can be seen. It can be seen evolving the surface splits the contour.

2.2 Segmentation using Level Sets

Typically, for applications in image segmentation F is dependent on the pixel intensity or curvature values. It may also be dependent on an edge indicator function, which is defined as having a value zero on an edge, and zero otherwise. This causes F to slow the level set evolution when on an edge.

In [3] F is dependent on a data term and a curvature term (with a weighting term between the two) for the purposes of image segmentation. Therefore, the level set equation takes the form

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

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 [3, 1], is given by

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

which is plotted in Figure 2.2. 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.

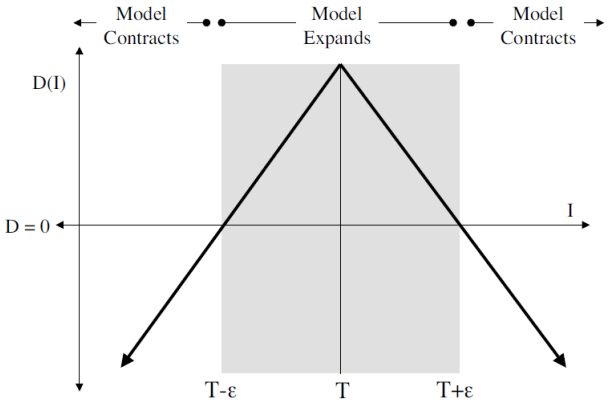


Figure 2.2: The speed term from [1]

Therefore the three user parameters that need to be specified for segmentation are T,ϵ and α. An initialization 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.

PICTURE OF CURVATURE FAIL

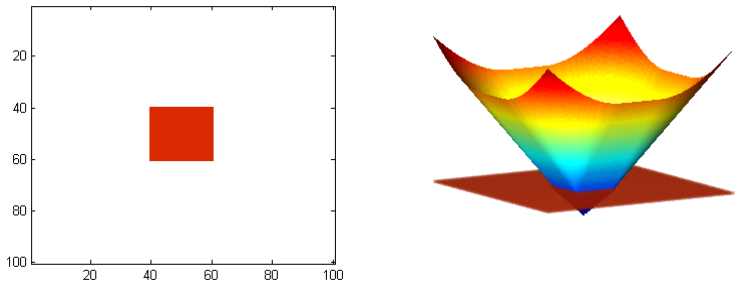
2.2.1 Signed Distance Transform

A distance transform assigns a value for every pixel (or voxel) within a binary image containing one or more objects the value of 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 [7], 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. 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 distance transforms often trade accuracy for efficiency and feature varying levels of complexity.

Signed distance transforms are required to initialize ϕ and also to reinitialize it



(a) Arbitrary Initial Mask

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

Figure 2.3: 2D Signed Euclidean Distance Transform

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).

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 [6].

A first order accurate method for time discretization of equation (2.1), is given by the forward Euler method, from [6]:

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

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).

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

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 difference or backwards difference 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^-) should be used. Conversely, if $u_i < 0$, forward difference methods (D_x^+) 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, from [3], results in the derivatives below required for the level set equation update.

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

$\nabla\phi$ 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.

3.1.2 Curvature

Curvature is computed based on the values of the current level set using the derivatives below. In two dimensions only the first two derivatives are required, alongside the derivatives defined previously. In three dimensions, all the derivatives below are required.

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

Using the *difference of normals* method from [3], curvature is computed using the above derivates with the two normals $\mathbf{n}^+$ and $\mathbf{n}^-$.

$$
\mathbf{n}^+ = \begin{bmatrix}
\frac{D_x^+}{\sqrt{(D_x^+)^2 + \left(\frac{D_y^{+x} + D_y}{2}\right)^2 + \left(\frac{D_z^{+x} + D_z}{2}\right)^2}} \\
\frac{D_y^+}{\sqrt{(D_y^+)^2 + \left(\frac{D_x^{+y} + D_x}{2}\right)^2 + \left(\frac{D_z^{+y} + D_z}{2}\right)^2}} \\
\frac{D_z^+}{\sqrt{(D_z^+)^2 + \left(\frac{D_y^{+z} + D_x}{2}\right)^2 + \left(\frac{D_y^{+z} + D_y}{2}\right)^2}}
\end{bmatrix} \tag{3.9}
$$

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

The two normals are used to compute divergence, allowing for mean curvature to

be computed as shown below in equation (3.11).

$$H = \frac{1}{2}\nabla \cdot \frac{\nabla\phi}{|\nabla\phi|} = \frac{1}{2}((\mathbf{n}_x^+ - \mathbf{n}_x^-) + (\mathbf{n}_y^+ - \mathbf{n}_y^-) + (\mathbf{n}_z^+ - \mathbf{n}_z^-)) \tag{3.11}$$

3.1.3 Stability

From [6], 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-Friedreichs-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

$$\Delta t < \frac{\Delta x}{max\,\{|u|\}} \tag{3.12}$$

which is usually implemented, through variants of equation (3.12), by choosing a *CFL number* that lies between 0 and 1 to further guarentee 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 (3.9),(3.10) to ensure that $\mathbf{n}$ does not tend to infinity if the square root is zero.

3.2 2D Sequential Implementation

Two dimensional implementations of the code in MATLAB, C and then CUDA were the first to be written. Once these had been optimized, three dimensional implementations were coded. The following pseudocode outlines the structure of the MATLAB, C and CUDA implementations, with only minor differences between the different versions.

3.2.1 Matlab

The first task was to write code in MATLAB to segment two dimensional greyscale images. The MATLAB Image Processing Toolbox provides many functions (such as the ability to load, resample and filter images, compute distance transforms and easily visualise the level set evolution) which kept the code reasonably concise.

The code is split into two files (a launcher and a kernel), in order to seperate the initialisation and level set update code. The launcher handles the image loading and

Algorithm 1: Pseudocode for Level Set Segmentation

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

Initialise ϕ_0 to S.D.F 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 % 50 == 0* **then**
    Reinitialise ϕ to S.D.F
  end
end

resampling, 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).

The level set function ϕ is then initialised 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). Reinitialisation 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 ϕ from ϕ (or vice-versa). Note how derivatives are not calculated in an element by element fashion.

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.

3.2.2 C

Initially C code was written to most closely mirror the MATLAB code. The feature image, level set function, and derivatives were stored in memory as one dimensional arrays. These arrays were stepped through by nested for loops, looping i looping

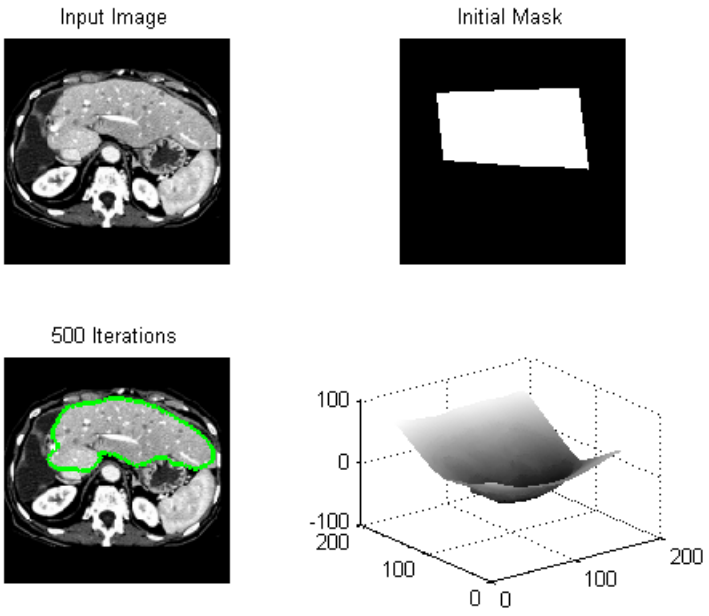


Figure 3.1: MATLAB user interface showing four subfigures with the input image, the initial mask, the current zero level set interface superimposed on the input image and the current level set surface in 3D

j. To go from the two dimensional i, j indices to a one dimensional index *ind*, the equation $\text{ind} = j \times \text{imageW} + i$ was used. These for loops computed the derivatives serially, as shown in the pseudocode for algorithm 2. This approach favoured itself well to being optimized at a low level using pointers to quickly step through the data. It was inefficient from a memory management perspective, with the derivative arrays taking up large amounts of memory for large image sizes, and also continuosly being allocated and freed at each iteration.

Algorithm 2: Pseudocode for Version 1 of Sequential C Code

forall i, j **do**
  Calculate D_x
forall i, j **do**
  Calculate D_y
forall i, j **do**
  Calculate D_z
forall i, j **do**
  Calculate D_x^+
...
Update Level Set Function $\phi(t + \Delta t) = \phi(t) + \Delta t F |\nabla \phi|$

This C implementation did not feature the versatility of the MATLAB implementation as it only accepted bitmap images as the input for the feature image.

The loader function for the bitmap images (`bmploader.cpp`) is from the NVIDIA CUDA SDK image processing examples. It also did not feature any image resizing functions, as the focus was on optimization of the code and not versatility of inputs.

Wheras the MATLAB code used shifts of the matrix ϕ to calculate derivatives, in C the level set function ϕ was being stepped through in an element by element fashion. Therefore 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 $D_x^+ = u_{i+1,j} - u_{i,j}$ must equal zero when $i =$ imageW as there is no $u_{i+1,j}$ term. The complexity of this task increases in three dimensions as there are six boundaries instead of four boundaries to condition for.

Restructuring the code for parallel computation of derivatives (using only one for loop) laid the framework for parallelization in CUDA, and only had a minor impact on performance. In this case all the derivatives are calculated at the same time at a given pixel, as shown below in algorithm 3. Also, the derivatives no longer needed to be stored in memory as arrays, and were simply floating point variables declared at each iteration. These two changes condensed the C code greatly.

Algorithm 3: Pseudocode for Version 2 of Sequential C Code

forall i, j **do**
- Calculate D_x
- Calculate D_y
- Calculate D_z
- Calculate D_x^+
- ...

Update Level Set Function $\phi(t + \Delta t) = \phi(t) + \Delta t F |\nabla \phi|$

For the distance transform initialization and reinitialization proceedures a seperate function was called. This is the `sedt2d` (Signed Euclidean Distance Transform in 2D) function written by Timothy Terriberry.

In order to visualise 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 visualise how the level set was evolving (checking for instabilities, incorrect parameters for thresholding, range and curvature) and view the final segmentation.

3.3 2D Parallel Implemention

3.3.1 Unoptimized Version

The first parallel implementation followed the structure shown in the pseudocode below. In CUDA, it is assumed that both the host and device maintain their own DRAM [5]. 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 minimise 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 array's x dimension is not a multiple of 16. Therefore most CUDA programs follow a standard strucutre 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 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.

Algorithm 4: Parallel Implementation Pseudocode

```
Initialise φ(i,j0), 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 φ(i,jn+1) = φ(i,jn) + ΔtF|∇φ_n|
    Swap pointers of φ(i,jn), φ(i,jn+1)
    if Iterations % 50 == 0 then
        Copy φ from device to host
        Reinitialise φ to S.D.F
        Copy φ from host to device
    end
end
Copy φ from device to host
```

CUDA threads are assigned a unique thread ID that identifies its location within the threadblock 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

(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

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

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

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 2×2 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

```
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). It should be noted that the size of these block sizes affect kernel performance greatly and that much larger block sizes are used (in the proceeding algorthms typical block sizes are 32×4 or 16×8). The effect of different block sizes on performance is analysed in Section 4.1.

Once these indices were set up, only minor tweaks were needed until the code was performing well. 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.

Some features such as the CFL condition, could not be implemented in this parallel version without slowing down computation time significantly. This is because such a condition requires the determinination 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 $max\{F|\nabla\phi|\}$. The cost of this added complexity and slowdown outweighed the benefits, and therefore Δt was chosen to be a free 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 [5].

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 this each tile must also contain values for neighbourhood nodes (often known as *halo* nodes) for the $i \pm 1$ and $j \pm 1$ elements, 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. [4] 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 discretisation' algorithm [2] 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.

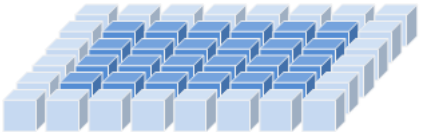


Figure 3.2: 2D Shared Memory Arrangement

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 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 [2].

```
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.

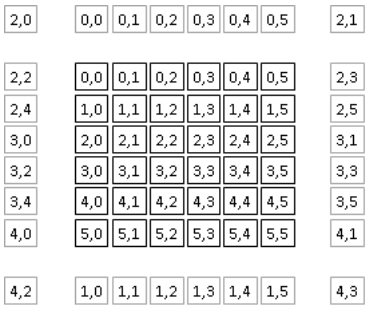


Figure 3.3: Tile and halo showing for a 6×6 block the mapping of thread IDs to halo nodes

This method of loading elements has been chosen in order to maximise *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 [2] uses three k-planes of data for this purpose as shown below in Figure 3.4.

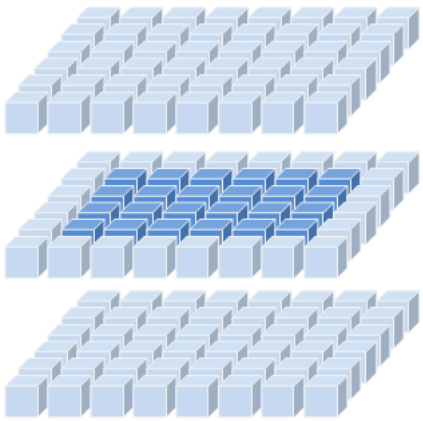


Figure 3.4: 3D Shared Memory Arrangement

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_{k+1} plane is loaded into the $k+1$ plane. Level set function values are updated, and then the k plane is shifted to the $k-1$ plane, the $k+1$ plane is shifted to the k plane, and new values are loaded from ϕ_{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 GPU hardware 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.

Although 8600M GT is adequate for CUDA development, it has rather limited performance in comparison to other graphics chips. 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.

4.1 Speed Tests and Analysis

4.1.1 2D Segmentations

4.2 Limitations

Chapter 5

Conclusions and Future Work

5.1 Conclusion

5.2 Future Work

Chapter 6

References

Chapter 7

Appendix

7.1 MATLAB Level Set Update Code

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

%-- ensures image is 2D double matrix
I = image2graydouble(I);

%-- 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;


%-- Converts image to one channel (grayscale) double
function img = image2graydouble(img)
[dimy, dimx, c] = size(img);
if(isfloat(img)) % image is a double
    if(c==3)
        img = rgb2gray(uint8(img));
    end
else            % image is a int
    if(c==3)
        img = rgb2gray(img);
    end
    img = double(img);
end
```

Bibliography

[1] 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.

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

[3] 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.

[4] 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.

[5] NVIDIA. Compute unified device architecture programming guide. *Nvidia, June*, 2007.

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

[7] 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.