Fast Level Set Segmentation of Biomedical Images using Graphics Processing Units

Hormuz Mostofi

April 17, 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 [3]: 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. Global memory has 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 [3], it recommended that each thread in a block firstly loads data from global memory to shared memory, synchronizes with all other threads within the threadblock 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.

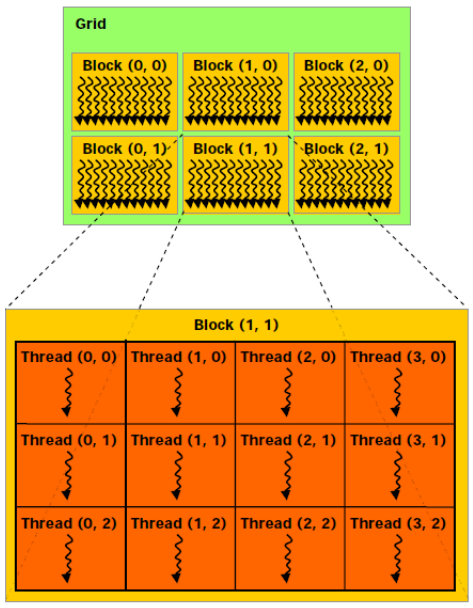


Figure 1.1: A grid of thread blocks. This figure taken from [3]

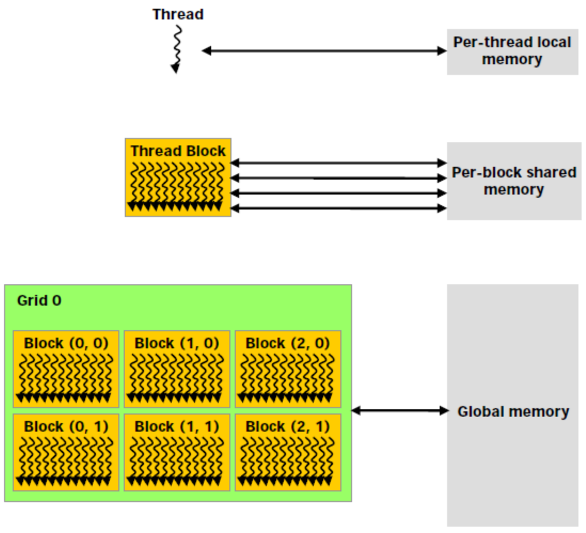


Figure 1.2: The memory heirarchy of CUDA threads and blocks. This figure taken from [3]

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 [5], 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.

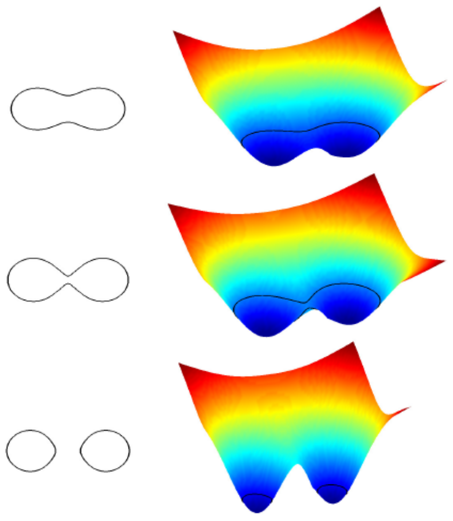


Figure 2.1: The relationship between the level set function (left) and contour (right) can be seen. It can be seen evolving the surface splits the contour.

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 [2] 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 [2, 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.

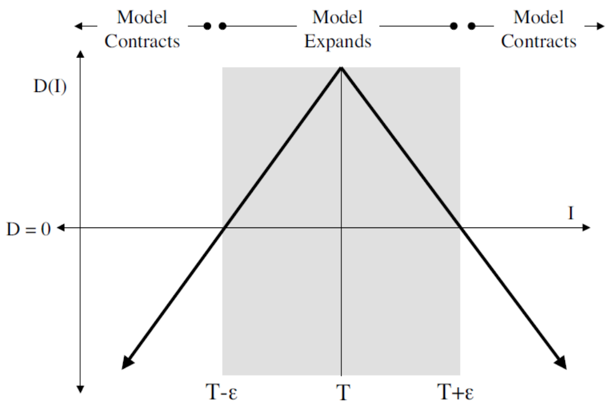


Figure 2.2: The speed term from [1]

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.

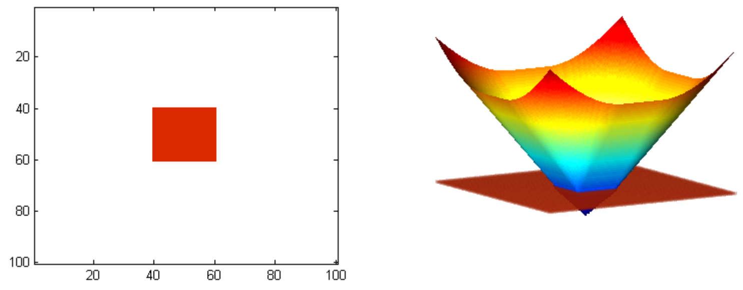
PICTURE OF CURVATURE FAIL

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 $D : \mathbb{R}^3 \to \mathbb{R}$ for a set S, from [5], 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



(a) Arbitrary Initial Mask

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

Figure 2.3: 2D Signed Euclidean Distance Transform

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 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 [4].

A first order accurate method for time discretization of equation (2.1), is given by the forward Euler method, from [4]:

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

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 [2], results in the derivatives below required for the level set equation update.

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

$\nabla\phi$ 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.

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

Using the *difference of normals* method from [2], curvature is computed using the above derivates with the two normals $\mathbf{n}^+$ and $\mathbf{n}^-$.

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

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

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 [4], 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

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

which is usually implemented, through variants of equation (3.13), 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 (3.11) to ensure that **n** does not tend to infinity if the square root is zero.

3.2 Sequential Implementation

Two dimensional implementations of the code in MATLAB, C and then CUDA were firstly written. Once these had been optimized, three dimensional implementations were written.

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 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).

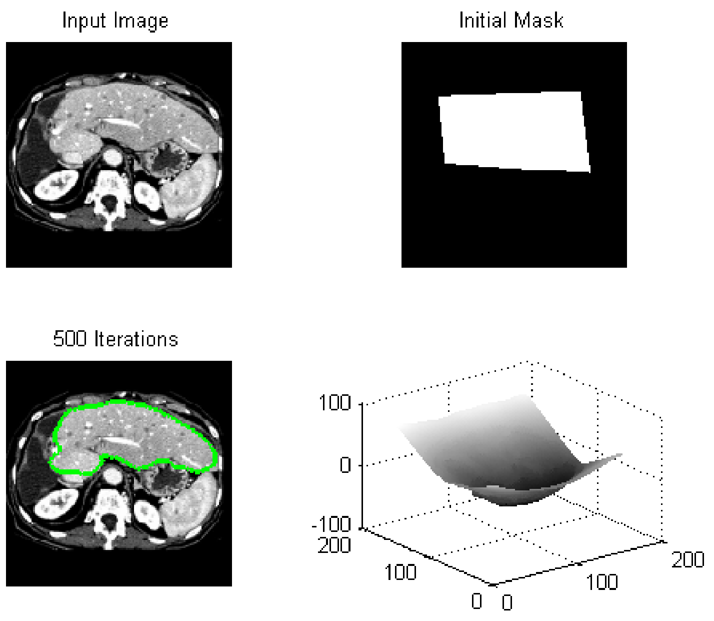


Figure 3.1: MATLAB user interface showing four subfigures with the input image, the initial mask, the current zero level set interface superimposed on the input image and the current level set surface in 3D

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.

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

3.3 Parallel Implemention

3.3.1 Block and Grid Sizes

3.3.2 Shared Memory

Chapter 4

Results

4.1 Speed Tests and Analysis

4.2 Limitations

Chapter 5

Conclusions and Future Work

5.1 Conclusion

5.2 Future Work

Chapter 6

References

Chapter 7

Appendix

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] 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.

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

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

[5] 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.