Fast, Streaming 3D Levelset on the GPU for Smooth Multi-phase Segmentation

Ojaswa Sharma\textsuperscript{1}, Qin Zhang\textsuperscript{2}, François Anton\textsuperscript{1}, and Chandrajit Bajaj\textsuperscript{2}

\textsuperscript{1} DTU Informatics
The Technical University of Denmark, Lyngby, 2800, Denmark
\{os,fa\}@imm.dtu.dk
\textsuperscript{2} Computational Visualization Center
The University of Texas at Austin, Austin, Texas, 78712-0027, USA
\{zqyork@ices,bajaj@cs\}.utexas.edu

Abstract. Level set method based segmentation provides an efficient tool for topological and geometrical shape handling, but is slow due to high computational burden. In this work, we provide a framework for streaming computations on large volumetric images on the GPU. A streaming computational model allows processing large amount of data with small memory footprint. Efficient transfer of data to and from the graphics hardware is performed via a memory manager. We show volumetric segmentation using higher order, multi-phase level set method with speedups of the order of 20x.

Key words: Segmentation, graphics hardware, GPU, streaming computation, level set, multi-phase, higher-order

1 Introduction

Volume segmentation is a computationally demanding task. We address this problem by employing a fast solution to the involved partial differential equations (PDEs) using the graphics processing unit (GPU). A recent trend in solving computationally expensive problems is to redesign the solution of the problem so that it can take advantage of the high arithmetic capability of the GPUs. We propose a novel GPU based framework for level set segmentation of large volumes.

A segmentation problem is to subdivide a three dimensional image $I(x, y, z): \Omega \mapsto \mathbb{R}$ ($\Omega \subset \mathbb{R}^3$) into non-overlapping partitions $\Omega_i (i = 1..n_c)$ such that $\bigcup \Omega_i = \Omega$, where each partition is homogeneous in the sense that it minimizes a certain quantity. Each region is said to produce a class representing the partition.

Implicit surfaces naturally capture the topology of the underlying surface in contrast to explicit or parameterized surfaces. Therefore level set based methods are very useful in this context. Deformable level set surfaces under mean curvature flow provide an intuitive means of segmentation. The pioneering work by Osher and Sethian [10] presents an effective implicit representation for evolving
curves. Later on, the work was developed in context of the Mumford-Shah functional by Chan and Vese [6] for 2D images that do not contain prominent edges. In a more advanced paper, these authors suggest a multi-domain segmentation [14] using the same level set framework. Other variants of the same method exist for applications like image de-noising based on total variation minimization [12]. Conventional level set methods solve the interface evolution equation with linear interpolation of the implicit function and its derivatives (at sampled grid points). The resulting level set surface is $C^0$. Bajaj et al. [2, 3] present a cubic spline based level set method that produces a $C^2$ level set surface.

Due to the high computational intensity of the level set method and inherent parallelism in the solution of the involved PDEs, a parallel compute environment is best suited. Schemes for fast evaluation of PDEs are suggested by Weickert et al. [15]. Multigrid methods are also suitable for a fast solution of differential equations. An active contour model using multigrid methods is suggested by Papandreou and Maragos [11]. Of particular interest is a solution to the level set equations for segmenting large volumes. GPU based implementations have been proposed by [13, 8], among which Lefohn et al. [8] demonstrate an efficient sparse GPU segmentation using level set methods.

In this work we propose a streaming solver framework suited to large volume segmentation. With 3D textures available to the commodity graphics hardware, we show that a 2D slicing is no longer required for a solution. This is also in contrast to [8] where the authors use a compact representation of the active volume packed into 2D textures. We solve the governing partial differential equations (PDEs) for a general case of any number of segmentation classes. The number of classes is determined as the level set evolves, creating new classes while merging some of the existing ones. Every single class then gives rise to a partition of the volume. The result of the streaming solver is demonstrated with multi-domain segmentation along with speedup benchmarks for tri-linear and tri-cubic level set computations.

2 Level Set Segmentation

The main idea behind a level set based segmentation method is to minimize an energy term over a domain by numerically solving the corresponding time varying form of the variational equation. Let us represent a volume by a scalar field $I(x, y, z) : \Omega \mapsto K$, where $\Omega$ is a bounded open subset of $\mathbb{R}^3$, and $K \subset \mathbb{R}$ is a bounded set of discrete intensity values sampled over a regular grid. In this setup, motion by mean curvature provides a deformable level set formulation where the surface of interest moves in the direction of normal at any point with velocity proportional to the curvature [10].

The deformable surface is represented by a level set of an implicit function $\phi(x, y, z) : \Omega \mapsto \mathbb{R}$. In level set methods, $\phi$ is generally chosen to be a signed distance function since it allows mean curvature flow with unit speed normal to the level set interface [10, chap. 6]. Toward a segmentation approach, various energy formulations are possible. The energy functional is further penalized by
a regularizing term that introduces smoothness in the resulting surface. The Mumford-Shah energy functional has a distinct advantage of producing better segmentation regions in absence of sharp edges as compared to an edge based energy functional. Consider an evolving interface \( \Gamma = \{(x, y, z) : \phi(x, y, z) = 0\} \) in \( \Omega \), denoting \( \Gamma^+ = \{(x, y, z) : \phi > 0\} \) as the interior of the volume bounded by \( \Gamma \) and \( \Gamma^- = \{(x, y, z) : \phi < 0\} \) as the exterior of the volume bounded by \( \Gamma \). A modified Mumford-Shah energy functional with regularization can be written as:

\[
F(c_1, c_2, \Gamma) = \mu \cdot \text{Area}(\Gamma) + \nu \cdot \text{Volume}(\Gamma^+) + \lambda_1 \int_{\Gamma^+} |I - c_1|^2 \, dx \, dy \, dz + \lambda_2 \int_{\Gamma^-} |I - c_2|^2 \, dx \, dy \, dz, \tag{1}
\]

where \( \mu \geq 0, \nu \geq 0, \lambda_1 > 0, \) and \( \lambda_2 > 0 \) are fixed scalar control parameters. A time varying variational form of (1) is [6]:

\[
\frac{\partial \phi}{\partial t} = \delta_\epsilon(\phi) \left[ \mu \nabla \cdot \left( \frac{\nabla \phi}{|\nabla \phi|} \right) - \nu - \lambda_1 (I_0 - c_1)^2 + \lambda_2 (I_0 - c_2)^2 \right], \tag{2}
\]

where, \( c_1 \) and \( c_2 \) are averages in \( \Gamma^+ \) and \( \Gamma^- \) respectively. Bajaj et al. [2] propose to solve the higher order regularizing term in (2) by cubic spline interpolation to compute accurate higher order derivatives of \( \phi \).

### 3 Multi-phase, Higher-order Level Set Method

Equation (2) defines two regions with respect to the zero level set surface of \( \phi \), i.e., \( \phi > 0 \) and \( \phi < 0 \). Often in segmentation, we need more than two partitions of the input signal. Vese and Chan [14] show that multiple level set evolutions can be used to keep track of multiple regions in the signal. In a Multi-domain setup a single implicit function \( \phi \) is replaced by a vector valued \( \Phi = \{\phi_0, \phi_1, \ldots, \phi_{m-1}\} \) function where \( m \) is the total number of implicit functions that are combined to give a maximum of \( n = 2^m \) partitions of \( \Omega \). Equation (2) is replaced by a system of \( m \) PDEs. We compactly write this system as:

\[
\frac{\partial \phi_i}{\partial t} = \delta_\epsilon(\phi_i) \left[ \mu \nabla \cdot \left( \frac{\nabla \phi_i}{|\nabla \phi_i|} \right) - \nu - \sum_{k=0}^{2^{m-1}-1} \left\{ (I - c_{i,k})^2 ight. \\
- (I - c_{i,k}^0)^2 \left. \prod_{p=0}^{m-1} (b_{q,p} + (-1)^{b_{q,p}} H_\epsilon(\phi_p)) \right\} \right], \tag{3}
\]

for \( i \in [0, m-1] \), where \( H_\epsilon(z) = \frac{1}{2} \left( 1 + \frac{2}{\pi} \tan^{-1} \left( \frac{z}{\epsilon} \right) \right) \) is the smooth version of Heaviside function and
\[ c^0_{i,k} = \text{mean}(I) \text{ in } (x,y,z) : \begin{cases} \phi_p > 0, & \text{if } b_{q,p} = 1, \\ \phi_p < 0, & \text{if } b_{q,p} = 0 \end{cases} \]

with \( q = 2k - k \mod 2^i, \forall p \in [0, m-1] \), \( i \geq 0 \), \( i,k \geq 0 \) in (4)

\[ c^1_{i,k} = \text{mean}(I) \text{ in } (x,y,z) : \begin{cases} \phi_p > 0, & \text{if } b_{q,p} = 1, \\ \phi_p < 0, & \text{if } b_{q,p} = 0 \end{cases} \]

with \( q = 2k + 2^i - k \mod 2^i, \forall p \in [0, m-1] \), and \( b_{q,p} = p^{th} \text{ bit } (\in \{0, 1\}) \) in binary representation of \( q \) (either from (4) or (5)).

Consider \( q \in Z^+ \) such that its \( i^{th} \) bit is 1 for \( I^+_{i} \), 0 for \( I^-_{i} \), and 0 if \( i > (m-1) \). In this way, \( q \) spans the \( n \) possible regions induced by \( \Phi \). In (3), \( c^1_{i,k} \) (or \( c^0_{i,k} \)) represents the average of intensity values of \( I \) in \( I^+_{i} \) (or \( I^-_{i} \)) where the other bits determine regions inside or outside for rest of the implicit surfaces. To generate the index for a region, we enumerate the possible \( 2^{m-1} \) values and insert a 1 or a 0 at the \( i^{th} \) bit. Alternative expressions for \( q \) in (4) and (5) are

\[
q = (k - (k \land (2^i - 1))) \ll 1 + k \land (2^i - 1), \text{ and (6)}
\]

\[
q = (k - (k \land (2^i - 1))) \ll 1 + 2^i + k \land (2^i - 1), \text{ and (7)}
\]

respectively. We find that the binary indexing for keeping track of various regions of \( \Phi \) is not only compact but also efficient for implementation on a constrained compute environment like the GPU.

### 4 Streaming Solver Architecture

In this section the streaming architecture of our GPU level set solver is presented. The solver operates on three-dimensional volumes and therefore, makes extensive use of 3D textures available to current GPUs. The complexity of the solver is increased by the fact that \( m \) simultaneous PDEs need to be solved to arrive at a solution to (3). The framework is based on the NVIDIA CUDA compute framework [9]. The solver has a very small memory footprint on the GPU compared to the actual volume that it can handle. On the CPU side, memory requirement is of the order of the number of implicit functions. Our GPU computational setup consists of a host memory manager to handle data streaming and a set of CUDA kernels to operate on parts of the data fetched to the device by the host.

#### 4.1 Compute Unified Device Architecture (CUDA)

Volume processing is a computationally intensive task since every voxel needs to be updated. Further, solving a differential equation on a regular grid by a numerical update method demands for even higher computational resources both in terms of clock cycles and memory. Today’s graphics processing processing units (GPU) outperform any CPU in terms of raw computing power by factors of 10 and more. NVIDIA provides a general purpose GPU computing framework known as the Compute Unified Device Architecture (CUDA).
A GPU (also called a device) is a set of Streaming Multiprocessors (SMs) employing a new SIMT (Single-Instruction Multiple-Threads) multi-processor computational architecture. Each SM consists of eight Scalar Processor (SP) cores. A device acts as a coprocessor to the CPU (also called a host) capable of running tens of thousands of threads at once (512 threads per block, with a maximum of 65535 blocks). Every SP has its own registers while every SM has access to small (16 KB) but high speed shared memory. The whole device has access to device memory, constant memory, and texture memory. The constant and texture memory are cached while the device memory is not. In fact, the shared memory can be viewed as a user managed cache.

The logical CUDA programming model exposes three basic concepts: a hierarchy of thread groups, shared memories, and barrier synchronization. C for CUDA is an extension to C with a minimal set of additions. A CUDA computation on the device takes place in the form of functions called kernels executed by every thread.

4.2 Memory Manager

The solver is designed keeping in mind large volumes of data and therefore we stick to the streaming paradigm for processing. Central to our GPU computational setup is the memory manager to handle data streaming between the host and the device. Fig. 1 shows a schematic diagram of the memory manager. It is responsible for the following

- Maintaining a memory hierarchy.
- Managing memory transfers between the host and the device.

**Memory Hierarchy.** The data on the device is handled in manageable chunks of a 3D sub-volume called a computational volume. The memory manager splits the entire volume into the minimum possible number of sub-volumes of size of the computational volume. The size of such a computational volume is chosen such that:

- computations are performed by one thread per voxel.
- the computational volume fits into the device memory.

The computational volume is further divided into the CUDA grid and block for thread invocation. CUDA allows for a 3D block of threads, but not a 3D grid of blocks. The memory manager builds a logical 3D grid that is mapped to a 1D grid of blocks, assigning each voxel with a thread to process it. For a $128 \times 128 \times 128$ computational volume, a typical grid size could be $16 \times 16 \times 16$ blocks with each block consisting $8 \times 8 \times 8$ threads. Note that the number of threads in a block cannot exceed 512 with the version 2.0 of CUDA. The hierarchy of thread blocks, grid and the computational volume is shown in Fig. 2.
Fig. 1. The memory manager.

Fig. 2. Volume hierarchy.
Memory Transfers. For operations that involve accessing voxel neighbours (e.g., finite differencing or convolution filtering), the memory manager appropriately pads the computational volume to enable the required number of shared voxels around the border of the computational volume. This effectively reduces the size of the volume to incorporate neighbours along the border of the volume. Further, the full volume might not be an exact multiple of the computational volume, therefore the memory manager pads the computational volume on the boundary with null values in the empty space.

Memory copies between device and host are performed in size of the computational volume. Special care is taken while copying data along the border of the full volume. The memory manager also dynamically allocates and frees the device memory if required by a kernel. Transferred computational volumes exist as 3D textures on the device. Individual kernels then operate on these sub-volumes and the results are stored on the GPU global memory, which are then transferred back to the host volume(s).

4.3 Solver Kernels

For solving the level set equation, the solver performs a series of operations. In the order of operation, these are

1. Interface initialization,
2. Signed distance field computation,
3. Average values computation
4. Cubic coefficients computation, and
5. PDE time stepping.

To achieve maximum performance, we use hybrid CPU-GPU computations. Kernel specific details of the solver are explained next for the above mentioned operations. The majority of these steps can be executed in a streaming fashion with an exception of average value computation.

Interface Initialization. The level set interface $\Gamma$ is initialized to a bounding box or to a super-ellipsoid with center $(c_x, c_y, c_z)$ and radii $(r_x, r_y, r_z)$

$$\left(\frac{x-c_x}{r_x}\right)^n + \left(\frac{y-c_y}{r_y}\right)^n + \left(\frac{z-c_z}{r_z}\right)^n = 1,$$  \hspace{1cm} (8)

for a two domain segmentation. Multi-domain initialization should ensure that all the possible classes occupy non-zero regions in space at the start so that all the domains have scope of evolution. The domain $\Omega$ is partitioned into smaller sub-domains and each sub-domain is assigned small super-ellipsoids (with randomized centers) that form the interface $I'$ for each implicit function. Vese and Chan [14] observe a better and faster convergence in a 2D case for such an initialization. Our tests confirm this observation for the 3D case.
The kernel module for interface initialization computes the implicit function \( \Phi \) such that:

\[
\phi_i(x,y,z) = \begin{cases} 
  k, & \text{if } (x,y,z) \in \Gamma^+ \\
  -k, & \text{otherwise,}
\end{cases}
\]

where \( i \in [0, m - 1] \), and \( k \in \mathbb{R}^+ \) is a constant.

**Signed Distance Field.** The solver adopts a narrow band approach to constructing a signed distance field on GPU using a \( d \)-pass approach to compute the distance field in \( d \) layers where \( 2d \) is the integer width of the narrow band. This is a streaming algorithm to create a Chamfer distance field \cite{5} that uses optimal values of coefficients for distance multipliers to minimize accuracy error with an actual distance field (see Algorithm 1). The algorithm has complexity \( O(dN) \), where \( N \) is the total number of voxels in the volume.

**Algorithm 1**: Computation of signed distance field

Every voxel is updated based on the values of the neighbours. The resulting layer has distance values that are locally Euclidean. The kernel to compute a signed distance layer operates on the computational volume that has a shared 1-voxel border.
Average Values. In a two-domain segmentation, the zero level set of $\phi$ divides $\Omega$ into two regions. Average values $c_1$ and $c_2$ can be easily computed over these regions.

Multi-domain segmentation creates more than two regions corresponding to every class of segmentation. We use binary indexing (as explained in section 3) to keep track of inside and outside in every implicit function. Thus, for any $q \in [0, n]$, we can compute the average value of image intensity.

Computation of average values is a serial operation and requires parsing all the values in the dataset. Reduction algorithms do exist for a parallel computation of sum like operations [4]. A CUDA implementation of the same exists as the CUDPP library by Harris and Sengupta [7]. With CUDPP, however, it is difficult to sum up datasets that cannot fit into the device memory, thus requiring some sort of data slicing. In our experience, the overhead of data slicing, setting up the prefix sum and computing average values turns out to be more expensive than a CPU computation of the averages. Therefore, the average values are computed on the host.

Cubic Coefficients. Higher order level set requires a spline representation of the implicit function $\Phi$. The cubic coefficients are computed for a set of data values sampled along any direction. Tri-cubic spline coefficients can be derived from these coefficients by the computing cubic coefficients along each direction sequentially. The coefficients are derived along an axis in planar sections orthogonal to one of the co-ordinate axes (see Figure 3). The computation takes place in three steps, sequentially along each coordinate direction.

The memory manager determines the largest possible sub-volume, the GPU slice, that can fit into the available device memory. The full volume is then processed in sizes of the GPU slice. For every plane in the GPU slice, cubic coefficients are computed along the segments parallel to one of the coordinate axes. The resulting coefficients are written to device array of same size as the GPU slice and copied back to the host array holding the coefficients. The CUDA
kernel for computing coefficients works on a linear section per thread. The intermediate sequences are not stored, but recomputed during the recursive process to evaluate $c_i$.

**PDE Time Stepping.** The two-phase scalar equation is solved in a similar fashion except that it is trivial to compute terms for this equation. Equation (3) is solved by discretizing the lower order term and by computing the spline derivatives for the higher order curvature term. Each PDE has a single higher order term, and $2^{m-1}$ terms involving average values. Binary indexing is used again to enumerate the lower order terms in every PDE.

Since all the PDEs must be updated simultaneously (i.e., every voxel in all implicit functions must be updated in parallel), cubic coefficients for all the implicit functions in the device memory are required at all times. However, CUDA does not allow dynamically creating texture references. Furthermore, a 4D texture (array of 3D textures) is also not possible in CUDA. Therefore, we simulate a 4D texture by a large 3D texture containing computational volume sized coefficient sub-volumes for all implicit functions.

There is a possibility of wasting some device memory here for a very large number of implicit functions, since for some $m (>16)$ it might not be possible to have a rectangular 3D array. This is because the largest 3D texture in CUDA can be of size $2^{11} \times 2^{11} \times 2^{11}$, and we hit this limit along one dimension for a computational volume of size $128 \times 128 \times 128$ and $m > 16$. In such a case, the unusable device memory locked in the texture can be minimized by computing optimal number of three positive factors $m_x, m_y$ and $m_z$ for a number $\hat{m} \geq m$ such that $(\hat{m} - m)$ is minimized and $\hat{m} = m_x \times m_y \times m_z$. The three factors are the number of computational volumes along the coordinate axes. In doing so, $m_x$ is made as large as possible, followed by a similar heuristic for $m_y$.

With sub-volumes of $\Phi, I$ and the coefficients cached on the GPU, the PDE is updated for every voxel and for all the implicit functions.

5 Results

We present results of multi-domain segmentation on CT volume of the human thoracic cage, followed by GPU performance statistics. The tests are produced on an NVIDIA Tesla C870 machine with Dual-Core AMD Opteron™ processor 2218 running at clock speed of 2.6 GHz, and a physical memory of 2 GB. The GPU has 16 multiprocessors (128 processor cores) running at clock speed of 1.35 GHz and an onboard memory of 1.6 GB.

Figure 4 shows a computed tomography of the human thoracic cage. The Volume has a size of $256 \times 256 \times 256$ voxels. Segmentation parameters for this volume are: $\lambda_1 = \lambda_2 = 1$, $\mu = .000005 \times 255 \times 255$, $\epsilon = 1$, and $m = 3$. A time stepping $\Delta t = 0.01$ is used. The interface is initialized to a super-ellipsoid of power $p = 2$ with a bounding box offset of 5 voxels from all sides. A total of 60 solver iterations produced the segmentation shown in Figure 5. The segmentation
yields four prominent classes. These classes separate regions of ribs, spinal chord, lungs and bronchioles as shown in Figure 5(a), (b), (c), and (d) respectively.

Fig. 4. Volume rendering of computed Tomography (CT) image of the human thoracic cage.

5.1 Reconstruction Accuracy

We analyze the reconstruction accuracy with a synthetic example of a phantom volume of size $128 \times 128 \times 128$ consisting of a CSG object formed by the union of a cuboid and a sphere as shown in Figure 6. Noise of specified standard deviation is added to the clean volume (see Fig. 7) and the object is reconstructed. We measure the reconstruction error by means of the symmetrical Hausdorff distance which is a good measure of the distance between two meshes (see Aspert et al. [1]).

Symmetrical Hausdorff distance, $d_H$, between two surfaces $M_0$ and $M_1$ is given by

$$d_H(M_0, M_1) = \max \left\{ \sup_{x_0 \in M_0} \inf_{x_1 \in M_1} d(x_0, x_1), \sup_{x_1 \in M_1} \inf_{x_0 \in M_0} d(x_0, x_1) \right\} , \quad (10)$$

where $d(\cdot, \cdot)$ is an appropriate metric for measuring distance between two points in a metric space. $d_H(M_0, M_1)$ measures the maximum possible distance that will be required to travel from surface $M_0$ to $M_1$. We compute this metric and use it to quantify the error in reconstruction of a surface from noisy volume. Table 1 shows $d_H$ for increasing noise in the volumes with standard deviations
Fig. 5. Multi-domain segmentation of the human thoracic cage

(a) Ribs  (b) Spinal chord

(c) Lungs  (d) Bronchioles

Fig. 6. Phantom object
of 6, 12, 25, and 50 respectively. From the results, the accuracy of reconstruction (considering mean and rms $d_H$) decreases with increasing noise, nevertheless it always remains below a relative $d_H$ of 0.05% of the bounding box diagonal of the object. For the extreme case of $\sigma$ being 50, the reconstruction error goes high. The results presented here are produced without any filtering on the synthetic volumes. For very high noise content in the images, a preprocessing stage such as median filtering or anisotropic filtering is generally recommended that smears out the noise.

Table 1. Reconstruction error in noisy volumes

<table>
<thead>
<tr>
<th>$d_H$</th>
<th>Noise standard deviation $\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td>Max</td>
<td>Absolute</td>
</tr>
<tr>
<td></td>
<td>BBox %</td>
</tr>
<tr>
<td>Mean</td>
<td>Absolute</td>
</tr>
<tr>
<td></td>
<td>BBox %</td>
</tr>
<tr>
<td>RMS</td>
<td>Absolute</td>
</tr>
<tr>
<td></td>
<td>BBox %</td>
</tr>
</tbody>
</table>
5.2 Speedup

Interface initialization CUDA kernel has low arithmetic intensity, therefore a very high speedup of about 24x to 25x is achieved. On the other hand, PDE updates are expensive in terms of arithmetic operations, thus giving a speedup of about 3x with tri-linear update and that of 11x with tri-cubic update. It should be noted that the tri-cubic PDE update is faster than the tri-linear one since the later uses finite differencing to compute double derivatives, while the earlier uses texture lookups and fewer computations. Cubic coefficients are expensive to compute yielding a speedup of about 3x.

Performance speedups of GPU computations compared to CPU ones are shown in Figures 8 and 9. The speedups show a general trend (non-linear) of increase in performance with increase in size of volume. However, for very large volumes (e.g. 1602 × 1125 × 195 volume), both the host and the device computations slow down by a large extent due to excessive memory paging. Since the device computation reads the volume in small chunks of memory, we surmise that this access pattern hits the performance even more by increasing the number of page faults.

![Graph](image.png)

Fig. 8. GPU speedup for tri-linear segmentation

6 Conclusions

In this work we presented a framework for streaming computations on the GPU. The framework is employed for efficient computation of multi-domain, higher order level set method applied to the Mumford-Shah energy functional. The presented framework is generic and can be easily used with other energy functional as well. We show results of the segmentation on CT image along with performance speedups obtained with the solver. The overall performance gain obtained is 20x for the two-domain segmentation and 10x for the multi-domain segmentation.
Acknowledgements. This work was done while Ojaswa Sharma was visiting Chandrajit Bajaj at UT-CVC. His visit was supported by the Technical University of Denmark. The research of Qin Zhang and Chandrajit Bajaj was supported in part by NSF grant CNS-0540033 and NIH contracts R01-EB00487, R01-GM074258, R01-GM07308.

References